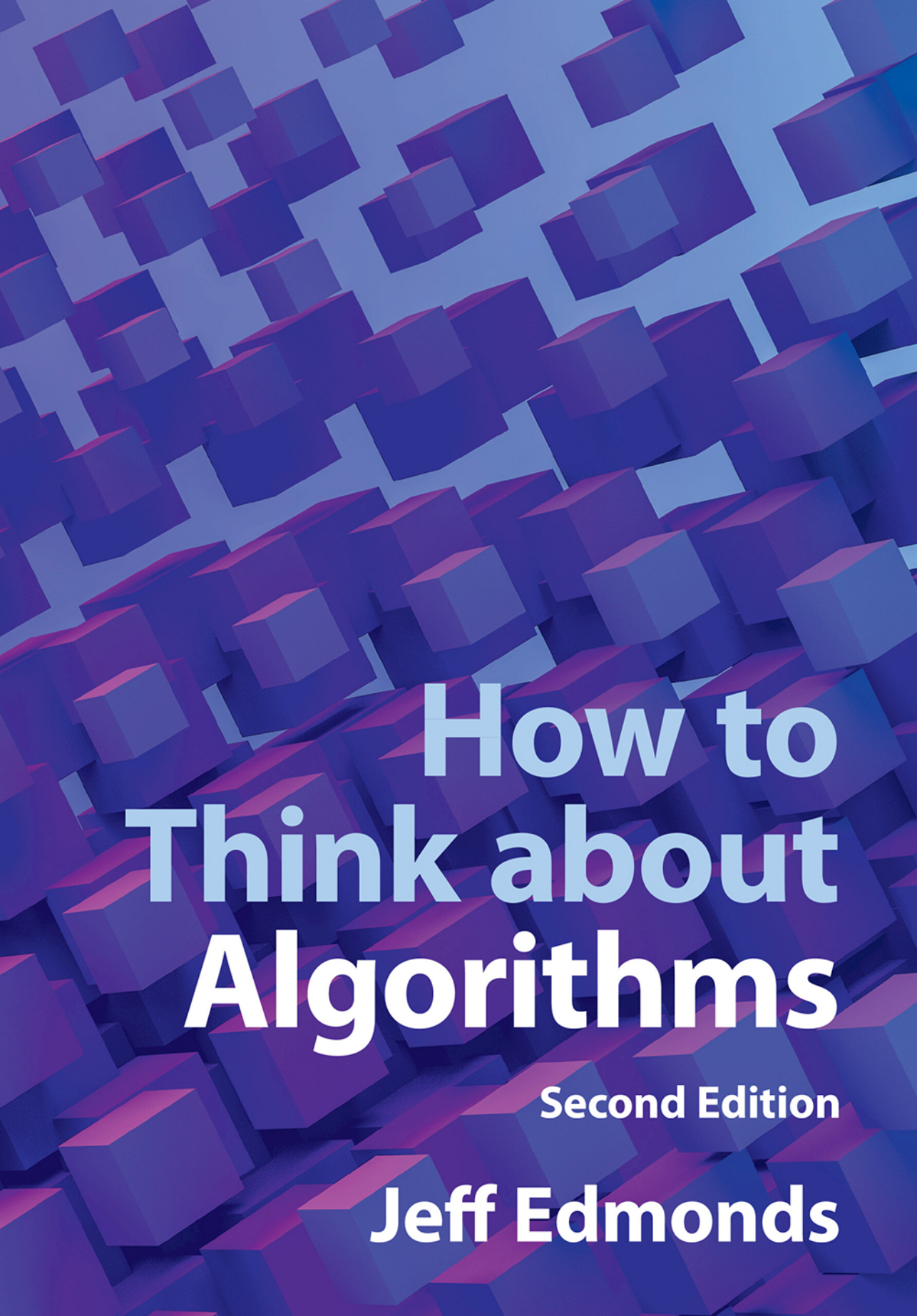


The background of the book cover is an abstract pattern of numerous 3D cubes. The cubes are rendered in various shades of purple, blue, and magenta, creating a sense of depth and perspective. They are arranged in a way that suggests a complex, three-dimensional grid or a stack of blocks, with some cubes appearing more prominent than others. The lighting is soft, highlighting the edges of the cubes and giving them a slightly glossy appearance.

How to Think about Algorithms

Second Edition

Jeff Edmonds

How to Think about Algorithms

Second Edition

Understand algorithms and their design with this revised student-friendly textbook. Unlike other algorithms books, this one is approachable, the methods it explains are straightforward, and the insights it provides are numerous and valuable. Without grinding through lots of formal proof, students will benefit from step-by-step methods for developing algorithms, expert guidance on common pitfalls, and an appreciation of the bigger picture. Revised and updated, this second edition includes a new chapter on machine learning algorithms, and concise key concept summaries at the end of each part for quick reference. Also new to this edition are more than 150 new exercises: selected solutions are included to let students check their progress, while a full solutions manual is available online for instructors. No other text explains complex topics such as loop invariants as clearly, helping students to think abstractly and preparing them for creating their own innovative ways to solve problems.

Jeff Edmonds is Professor in the Department of Electrical Engineering and Computer Science at York University, Canada.

“Jeff Edmonds’ *How to Think about Algorithms* offers a fresh perspective, placing methodical but intuitive design principles (pre- and post-conditions, invariants, ‘transparent’ correctness) as the bedrock on which to build and practice algorithmic thinking. The book reads like an epic guided meditation on the vast universe of algorithms, directing the reader’s attention to the core of each insight, while stimulating the mind through well-paced examples, playful but concise analogies, and thought-provoking exercises.”

Nathan Chenette, *Rose-Hulman Institute of Technology*

“With a good book like this in your hands, learning about algorithms and getting programs to work well will be fun and empowering. Anybody who wants to be a good programmer will get a great deal from this surprisingly readable book. Its approach makes it perfect for reading on your own if you want to enjoy learning about algorithms without being distracted by heavy maths. It has lots of exercises that are worth doing. Most importantly, *How to Think about Algorithms* does just that: it shows you how to think about algorithms and become a better programmer. Knowing how to think about algorithms gives you the insights and skills to make computers do anything more reliably and faster. The book is also ideal for any taught university course, because it is self-contained and systematically sets out the essential material, but most importantly because it empowers students to think for themselves.”

Harold Thimbleby, *Swansea University*

How to Think about Algorithms

Second Edition

JEFF EDMONDS

York University, Toronto





CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment,
a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of
education, learning and research at the highest international levels of excellence.

www.cambridge.org

Information on this title: www.cambridge.org/highereducation/isbn/9781009302142

DOI: [10.1017/9781009302180](https://doi.org/10.1017/9781009302180)

© Jeff Edmonds 2008, 2024

This publication is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without the written
permission of Cambridge University Press & Assessment.

First edition published 2008

Second edition published 2024

Printed in the United Kingdom by TJ Books Limited, Padstow, Cornwall, 2024

A catalogue record for this publication is available from the British Library

A Cataloging-in-Publication data record for this book is available from the Library of Congress

ISBN 978-1-009-30214-2 Hardback

ISBN 978-1-009-30213-5 Paperback

Additional resources for this publication at www.cambridge.org/Edmonds2e

Cambridge University Press & Assessment has no responsibility for the persistence
or accuracy of URLs for external or third-party internet websites referred to in this
publication and does not guarantee that any content on such websites is, or will
remain, accurate or appropriate.



Dedicated to my siblings, Jennifer, Martin, Alex, and Laura, and to my children, Josh and Micah.

May the love and the mathematics continue to flow between the generations.



Problem Solving
 Out of the Box Leaping
 Deep Thinking
 Creative Abstracting
 Logical Deducing
 with Friends Working
 Fun Having
 Fumbling and Bumbling
 Bravely Persevering
 Joyfully Succeeding



Contents

Preface	<i>page xiii</i>
Introduction	1
Part I Iterative Algorithms and Loop Invariants	
1 Iterative Algorithms: Measures of Progress and Loop Invariants	5
1.1 A Paradigm Shift: A Sequence of Actions vs. a Sequence of Assertions	5
1.2 The Steps to Develop an Iterative Algorithm	9
1.3 More about the Steps	13
1.4 Different Types of Iterative Algorithms	21
1.5 Code from Loop Invariants	28
1.6 Typical Errors	31
1.7 Exercises	32
2 Examples Using More-of-the-Input Loop Invariants	33
2.1 Coloring the Plane	33
2.2 Deterministic Finite Automaton	35
2.3 More of the Input vs. More of the Output	42
3 Abstract Data Types	47
3.1 Specifications and Hints at Implementations	47
3.2 Link List Implementation	55
3.3 Merging with a Queue	61
3.4 Parsing with a Stack	62
4 Narrowing the Search Space: Binary Search	64
4.1 Binary Search Trees	64
4.2 Magic Sevens	66
4.3 VLSI Chip Testing	68
4.4 Exercises	72
5 Iterative Sorting Algorithms	74
5.1 Bucket Sort by Hand	74

5.2	Counting Sort (a Stable Sort)	75
5.3	Radix Sort	78
6	More Iterative Algorithms	80
6.1	Euclid's GCD Algorithm	80
6.2	Multiplying	84
7	The Loop Invariant for Lower Bounds	88
8	Key Concepts Summary: Loop Invariants and Iterative Algorithms	97
8.1	Loop Invariants and Iterative Algorithms	97
8.2	System Invariants	99
9	Additional Exercises: Part I	102
10	Partial Solutions to Additional Exercises: Part I	124
Part II Recursion		
11	Abstractions, Techniques, and Theory	133
11.1	Thinking about Recursion	133
11.2	Looking Forward vs. Backward	134
11.3	With a Little Help from Your Friends	135
11.4	The Towers of Hanoi	138
11.5	Checklist for Recursive Algorithms	139
11.6	The Stack Frame	144
11.7	Proving Correctness with Strong Induction	146
12	Some Simple Examples of Recursive Algorithms	149
12.1	Sorting and Selecting Algorithms	149
12.2	Operations on Integers	157
12.3	Ackermann's Function	162
12.4	Fast Fourier Transformations	163
12.5	Exercise	168
13	Recursion on Trees	169
13.1	Tree Traversals	174
13.2	Simple Examples	177
13.3	Heap Sort and Priority Queues	180
13.4	Representing Expressions with Trees	187
14	Recursive Images	192
14.1	Drawing a Recursive Image from a Fixed Recursive and a Base Case Image	192
14.2	Randomly Generating a Maze	195

15	Parsing with Context-Free Grammars	198
16	Key Concepts Summary: Recursion	208
17	Additional Exercises: Part II	211
18	Partial Solutions to Additional Exercises: Part II	230
 Part III Optimization Problems		
19	Definition of Optimization Problems	241
20	Graph Search Algorithms	243
20.1	A Generic Search Algorithm	243
20.2	Breadth-First Search for Shortest Paths	248
20.3	Dijkstra's Shortest-Weighted-Path Algorithm	253
20.4	Depth-First Search	259
20.5	Recursive Depth-First Search	263
20.6	Linear Ordering of a Partial Order	264
20.7	Exercise	267
21	Network Flows and Linear Programming	268
21.1	A Hill-Climbing Algorithm with a Small Local Maximum	270
21.2	The Primal–Dual Hill-Climbing Method	276
21.3	The Steepest-Ascent Hill-Climbing Algorithm	284
21.4	Linear Programming	288
21.5	Exercises	293
22	Greedy Algorithms	294
22.1	Abstractions, Techniques, and Theory	294
22.2	Examples of Greedy Algorithms	307
22.3	Exercises	320
23	Recursive Backtracking	321
23.1	Recursive Backtracking Algorithms	321
23.2	The Steps in Developing a Recursive Backtracking	325
23.3	Pruning Branches	329
23.4	Satisfiability	331
23.5	Exercises	334
24	Dynamic Programming Algorithms	336
24.1	Start by Developing a Recursive Backtracking Algorithm	336
24.2	The Steps in Developing a Dynamic Programming Algorithm	340
24.3	Subtle Points	346

24.4	The Longest-Common-Subsequence Problem	364
24.5	Dynamic Programs as More-of-the-Input Iterative Loop Invariant Algorithms	368
24.6	A Greedy Dynamic Program: The Weighted Job/Event Scheduling Problem	371
25	Designing Dynamic Programming Algorithms via Reductions	375
26	The Game of Life	380
26.1	Graph G from Computation	380
26.2	The Graph of Life	382
26.3	Examples of the Graph of Life	385
27	Solution Is a Tree	390
27.1	The Solution Viewed as a Tree: Chains of Matrix Multiplications	390
27.2	Generalizing the Problem Solved: Best AVL Tree	395
27.3	All Pairs Using Matrix Multiplication	397
27.4	Parsing with Context-Free Grammars	398
28	Reductions and NP-Completeness	402
28.1	Satisfiability Is at Least as Hard as Any Optimization Problem	404
28.2	Steps to Prove NP-Completeness	407
28.3	Example: 3-Coloring Is NP-Complete	415
28.4	An Algorithm for Bipartite Matching Using the Network Flow Algorithm	419
29	Randomized Algorithms	423
29.1	Using Randomness to Hide the Worst Cases	423
29.2	Solutions of Optimization Problems with a Random Structure	427
30	Machine Learning	431
31	Key Concepts Summary: Greedy Algorithms and Dynamic Programming	439
31.1	Greedy Algorithms	439
31.2	Dynamic Programming	444
32	Additional Exercises: Part III	454
32.1	Graph Algorithms	454
32.2	Greedy Algorithms	457
32.3	Dynamic Programming	465
32.4	Reductions and NP-Completeness	476
33	Partial Solutions to Additional Exercises: Part III	482
33.1	Graph Algorithms	482

33.2	Greedy Algorithms	482
33.3	Dynamic Programming	485
33.4	Reductions and NP-Completeness	492
Part IV Additional Topics		
34	Existential and Universal Quantifiers	499
35	Time Complexity	508
35.1	The Time (and Space) Complexity of an Algorithm	508
35.2	The Time Complexity of a Computational Problem	513
36	Logarithms and Exponentials	515
37	Asymptotic Growth	518
37.1	Steps to Classify a Function	519
37.2	More about Asymptotic Notation	525
38	Adding-Made-Easy Approximations	529
38.1	The Technique	530
38.2	Some Proofs for the Adding-Made-Easy Technique	534
39	Recurrence Relations	540
39.1	The Technique	540
39.2	Some Proofs	543
40	A Formal Proof of Correctness	549
41	Additional Exercises: Part IV	551
41.1	Existential and Universal Quantifiers	551
41.2	Time Complexity	553
41.3	Asymptotic Growth	554
41.4	Adding Made-Easy Approximations	554
42	Partial Solutions to Additional Exercises: Part IV	556
42.1	Existential and Universal Quantifiers	556
42.2	Time Complexity	560
Exercise Solutions		561
Conclusion		588
Index		589

Preface

This book is designed to be used in a twelve-week, third-year algorithms course. The goal is to teach students to think abstractly about algorithms and about the key algorithmic techniques used to develop them.

Meta-Algorithms: Students must learn so many algorithms that they are sometimes overwhelmed. In order to facilitate their understanding, most textbooks cover the standard themes of iterative algorithms, recursion, greedy algorithms, and dynamic programming. Generally, however, when it comes to presenting the algorithms themselves and their proofs of correctness, the concepts are hidden within optimized code and slick proofs. One goal of this book is to present a uniform and clean way of thinking about algorithms. We do this by focusing on the structure and proof of correctness of *iterative* and *recursive* meta-algorithms, and within these the *greedy* and *dynamic programming* meta-algorithms. By learning these and their proofs of correctness, most actual algorithms can be easily understood. The challenge is that thinking about meta-algorithms requires a great deal of abstract thinking.

Abstract Thinking: Students are very good at learning how to apply a concrete code to a concrete input instance. They tend, however, to find it difficult to think abstractly about the algorithms. I maintain that the more abstractions a person has from which to view the problem, the deeper their understanding of it will be, the more tools they will have at their disposal, and the better prepared they will be to design their own innovative ways to solve new problems. Hence, I present a number of different notations, analogies, and paradigms within which to develop and to think about algorithms.

Levels: The psychological profiling of a successful person is mostly the ability to shift levels of abstraction.

To understand the detailed workings.

To understand the big picture.

To understand complex things in simple ways.

Way of Thinking: People who develop algorithms have various ways of thinking and intuition that tend not to get taught. The assumption, I suppose, is that these cannot

be taught but must be figured out on one's own. This text attempts to teach students to think like a designer of algorithms.

Not a Reference Book: My intention is not to teach a specific selection of algorithms for specific purposes. Hence, the book is not organized according to the application of the algorithms, but according to the techniques and abstractions used to develop them.

Developing Algorithms: The goal is not to present completed algorithms in a nice clean package, but to go slowly through every step of the development. Many false starts have been added. The hope is that this will help students learn to develop algorithms on their own. The difference is a bit like the difference between studying carpentry by looking at houses and by looking at hammers.

Proof of Correctness: Our philosophy is not to follow an algorithm with a formal proof that it is correct. Instead, this text is about learning how to think about, develop, and describe algorithms in such way that their correctness is transparent.

Big Picture vs. Small Steps: For each topic, I attempt both to give the big picture and to break it down into easily understood steps.

Level of Presentation: This material is difficult. There is no getting around that. I have tried to figure out where confusion may arise and to cover these points in more detail. I try to balance the succinct clarity that comes with mathematical formalism against the personified analogies and metaphors that help to provide both intuition and humor.

Point Form: The text is organized into blocks, each containing a title and a single thought. Hopefully, this will make the text easier to lecture and study from.

Prerequisites: The text assumes that the students have completed a first-year programming course and have a general mathematical maturity. The [Part IV](#), Additional Topics, covers much of the mathematics that will be needed.

Homework Questions: A few additional questions are included. I am hoping to develop many more, along with their solutions. Contributions are welcome.

Read Ahead: The student is expected to read the material *before* attending lectures or classes. This will facilitate productive discussion during class.

Explaining: To be able to prove yourself on a test or on the job, you need to be able to explain the material well. In addition, explaining it to someone else is the best way to learn it yourself. Hence, I highly recommend spending a lot of time explaining the material over and over again out loud to yourself, to each other, and to your stuffed bear.

Dreaming: I would like to emphasize the importance of thinking, even daydreaming, about the material. This can be done while going through your day – while swimming, showering, cooking, or lying in bed. Ask questions. Why is it done this way and not that way? Invent other algorithms for solving a problem. Then look for input instances for which your algorithm gives the wrong answer. Mathematics is not all linear thinking. If the essence of the material, what the questions are really asking, is allowed to seep down into your subconscious then with time little thoughts will begin to percolate up. Pursue these ideas. Sometimes even flashes of inspiration appear.

Acknowledgments

I would like to thank Andy Mirzaian, Franck van Breugel, James Elder, Suprakash Datta, Eric Ruppert, Russell Impagliazzo, Toniann Pitassi, and Kirk Pruhs, with whom I co-taught and co-researched algorithms for many years. I would like to thank Jennifer Wolfe, Lauren Cowles, Julie Lancashire, Anna Scriven, Rachel Norridge, and Beth Morel for their fantastic editing jobs. All of these people were a tremendous support for this work.

Introduction

From determining the cheapest way to make a hot dog to monitoring the workings of a factory, there are many complex *computational problems* to be solved. Before executable *code* can be produced, computer scientists need to be able to design the *algorithms* that lie behind the code, be able to understand and describe such algorithms abstractly, and be confident that they work correctly and efficiently. These are the goals of computer scientists.

A Computational Problem: A specification of a computational problem uses *preconditions* and *postconditions* to describe for each legal *input instance* that the computation might receive, what the required output or actions are. This may be a function mapping each input instance to the required output. It may be an optimization problem that requires a solution to be outputted that is “optimal” from among a huge set of possible solutions for the given input instance. It may also be an ongoing system or data structure that responds appropriately to a constant stream of input.

Example: The *sorting* problem is defined as follows:

Preconditions: The input is a list of n values, including possible repetitions.

Postconditions: The output is a list consisting of the same n values in nondecreasing order.

An Algorithm: An *algorithm* is a step-by-step procedure that, starting with an input instance, produces a suitable output. It is described at the level of detail and abstraction best suited to the human audience who must understand it. In contrast, *code* is an implementation of an algorithm that can be executed by a computer. *Pseudocode* lies between these two.

An Abstract Data Type: Computers use zeros and ones, ANDs and ORs, IFs and GOROs. This does not mean that *we* have to. The description of an algorithm may talk of *abstract objects* such as integers, reals, strings, sets, stacks, graphs, and trees; *abstract operations* such as “sort the list,” “pop the stack,” or “trace a path”; and *abstract relationships* such as greater than, prefix, subset, connected, and child. To be useful, the nature of these objects and the effect of these operations need to be understood. However, in order to hide details that are tedious or irrelevant, the precise implementations of these data structure and algorithms do not need to be specified. For more on this see [Chapter 3](#).

Correctness: An algorithm for the problem is *correct* if for every legal input instance, the required output is produced. Though a certain amount of logical thinking is required, the goal of this text is to teach how to think about, develop, and describe algorithms in such way that their correctness is transparent. See [Chapter 40](#) for the formal steps required to prove correctness, and [Chapter 34](#) for a discussion of *forall* and *exist* statements that are essential for making formal statements.

Running Time: It is not enough for a computation to eventually get the correct answer. It must also do so using a reasonable amount of time and memory space. The *running time* of an algorithm is a function from the *size* n of the input instance given to a bound on the number of *operations* the computation must do. (See [Chapter 35](#).) The algorithm is said to be *feasible* if this function is a *polynomial* like $\text{Time}(n) = \Theta(n^2)$, and is said to be *infeasible* if this function is an *exponential* like $\text{Time}(n) = \Theta(2^n)$. (See [Chapters 36](#) and [37](#) for more on the asymptotics of functions.) To be able to compute the running time, one needs to be able to add up the times taken in each iteration of a loop and to solve the recurrence relation defining the time of a recursive program. (See [Chapter 38](#) for an understanding of $\sum_{i=1}^n i = \Theta(n^2)$, and [Chapter 39](#) for an understanding of $T(n) = 2T(\frac{n}{2}) + n = \Theta(n \log n)$.)

Meta-algorithms: Most algorithms are best described as being either *iterative* or *recursive*. An iterative algorithm ([Part I](#)) takes one step at a time, ensuring that each step makes *progress* while maintaining the *loop invariant*. A recursive algorithm ([Part II](#)) breaks its instance into smaller instances, which it gets a *friend* to solve, and then combines their solutions into one of its own.

Optimization problems ([Part III](#)) form an important class of computational problems. The key algorithms for them are the following. *Greedy* algorithms ([Chapter 22](#)) keep grabbing the next object that looks best. *Recursive backtracking* algorithms ([Chapter 23](#)) try things and, if they don't work, backtrack and try something else. *Dynamic programming* ([Chapter 24](#)) solves a sequence of larger and larger instances, reusing the previously saved solutions for the smaller instances, until a solution is obtained for the given instance. *Reductions* ([Chapter 28](#)) use an algorithm for one problem to solve another. *Randomized* algorithms ([Chapter 29](#)) flip coins to help them decide what actions to take. Finally, *lower bounds* ([Chapter 7](#)) prove that there are no faster algorithms.

Part I

Iterative Algorithms and Loop Invariants

1 Iterative Algorithms: Measures of Progress and Loop Invariants

Using an *iterative algorithm* to solve a computational problem is a bit like following a road, possibly long and difficult, from your start location to your destination. With each iteration, you have a method that takes you a single step closer. To ensure that you move forward, you need to have a *measure of progress* telling you how far you are either from your starting location or to your destination. You cannot expect to know exactly where the algorithm will go and you need to expect some weaving and winding. On the other hand, you do not want to have to know how to handle every ditch and dead end in the world. A compromise between these two is to have a *loop invariant*, which defines a road (or region) that you may not leave. As you travel, worry about one step at a time. You must know how to get onto the road from any start location. From every place along the road, you must know what actions you will take in order to step forward while not leaving the road. Finally, when sufficient progress has been made along the road, you must know how to exit and reach your destination. Hopefully, this happens in a reasonable amount of time.

1.1 A Paradigm Shift: A Sequence of Actions vs. a Sequence of Assertions

Understanding iterative algorithms requires understanding the difference between a *loop invariant*, which is an *assertion* or picture of the computation at a particular point in time, and the actions that are required to maintain such a loop invariant.

One of the first important paradigm shifts that programmers struggle to make is from viewing an algorithm as a sequence of actions to viewing it as a sequence of snapshots of the state of the computer. Programmers tend to fixate on the first view, because code is a sequence of instructions for action and a computation is a sequence of actions. Though this is an important view, there is another. Imagine stopping time at key points during the computation and taking still pictures of the state of the computer. Then a computation can equally be viewed as a sequence of such snapshots. Having two ways of viewing the same thing gives one both more tools to handle it and a deeper understanding of it. An example of viewing a computation as an alternation between assertions about the current state of the computation and blocks of actions that bring the state of the computation to the next state is shown here.

```

Max(a, b, c)
  PreCond: Input has 3 numbers.
  m = a
  assert: m is max in {a}.
  if(b > m)
    m = b
  end if
  assert: m is max in {a, b}.
  if(c > m)
    m = c
  end if
  assert: m is max in {a, b, c}.
  return(m)
  PostCond: return max in {a, b, c}.
end algorithm

```

The Challenge of the Sequence-of-Actions View Suppose one is designing a new algorithm or explaining an algorithm to a friend. If one is thinking of it as sequence of actions, then one will likely start at the beginning: Do this. Do that. Do this. Shortly one can get lost and not know where one is. To follow this, one needs to keep track of how the state of the computer changes with each new action. In order to know what action to take next, one needs to have a global plan of where the computation is to go. To make it worse, the computation has many `ifs` and `loops` so one has to consider all the various paths that the computation may take.

The Advantages of the Sequence of Snapshots View: This new paradigm is a useful one from which one can think about, explain, or develop an algorithm.

Pre- and Postconditions: Before one can consider an algorithm, one needs to carefully define the computational problem being solved by it. This is done with pre- and postconditions by providing the initial picture, or *assertion*, about the input instance and a corresponding picture or assertion about required output.

Proof of Correctness of the Algorithm: It is important that you *know* that your code works. The proof might be a formal mathematical one or it might be informal hand waving. Either way, the formal statement of what needs to be proved is as follows.

$$PreCond \ \& \ code \Rightarrow \ PostCond$$

This states that if the input instance happens to meet the precondition and the code is executed, then in the end the output will meet the postcondition. On the other hand, if the precondition was not met, then except for being polite, there are no requirements on what your code does.

Check Each Computation Path: The computation from pre- to postcondition will follow one of possibly many paths. For each such path, you need to be confident that the postcondition will be met in the end. Suppose the above `Max(a, b, c)` code had not

two *if then else* statements but a sequence of r of them. There would then be 2^r paths though which the computation might follow. Checking each of these would be too much work. Instead, we will break the computation into subcomputations.

One Step at a Time: A main theme of this book is to implore you not to think about the entire computation all at once. As recommended in Alcoholics Anonymous, worry about one step at a time.

Start in the Middle: Instead of starting with the first line code, an alternative way to design an algorithm is to jump into the middle of the computation and to draw a static picture, or assertion, about the state we would like the computation to be in at this time. This picture does not need to state the exact value of each variable. Instead, it gives general properties and relationships between the various data structures that are key to understanding the algorithm. If this *assertion* is sufficiently general, it will capture not just this one point during the computation, but many similar points. Then it might become a part of a loop.

Sequence of Snapshots: Once one builds up a sequence of assertions in this way, one can see the entire path of the computation laid out before one.

Fill in the Actions: These assertions are just static snapshots of the computation with time stopped. No actions have been considered yet. The final step is to fill in actions (code) between consecutive assertions.

One Step at a Time: Each such block of actions can be executed completely independently of the others. It is much easier to consider them one at a time than to worry about the entire computation at once. In fact, one can complete these blocks in any order one wants and modify one block without worrying about the effect on the others.

Fly In from Mars: This is how you should fill in the code between the i th and the $i + 1^{\text{st}}$ assertions. Suppose you have just flown in from Mars, and absolutely the only thing you know about the current state of your computation is that the i th *assertion* holds. The computation might actually be in a state that is completely impossible to arrive at, given the algorithm that has been designed so far. It is allowing this that provides independence between these blocks of actions.

Take One Step: Being in a state in which the i th assertion holds, your task is simply to write some simple code to do a few simple actions, that change the state of the computation so that the $i + 1^{\text{st}}$ assertion holds.

Differentiating between Values: Define x_i to be the value of x when the computation is at the i th assertion and x_{i+1} to be that after the computation has gone around the loop and is at the $i + 1^{\text{st}}$. You may prefer the notation x' and x'' .

Code vs. Math Assertions: Denote as code_i the assertion of the result of the code between these assertions. Code $x = x + 1$, though a fine action, is incorrect as a statement. If you want to prove things you need to translate this into the mathematical

assertion $code_i : x_{i+1} = x_i + 1$. Similarly translate the actions $x = x + 1$; $y = y + x$; $x = x \times y$ into the assertion $x_{t+1} = (x_t + 1)(y_t + x_t + 1)$; $y_{t+1} = y_t + x_t + 1$.

Proof of Correctness of Each Step: The proof that your algorithm works can be done one block at a time. The formal statement of what needs to be proved is as follows:

$$\langle i^{th} - assertion \rangle \& code_i \Rightarrow \langle i + 1^{st} - assertion \rangle.$$

For example, a statement $\langle i^{th} - assertion \rangle : x_i \text{ is odd}$ assures us that the state of the computation is such that i^{th} assertion holds for the values x_i . The statement $code_i : x_{i+1} = x_i + 1$ assures us that if you restart the code just long enough to execute the next block of code, the stated relationship holds between the x_i values and the x_{i+1} values. Our goal is to combine these two facts to prove that the statement $\langle i + 1^{st} - assertion \rangle : x_{i+1} \text{ is even}$ holds for the x_{i+1} values.

Proof by Case: If you can prove: 1) There are only two cases, 2) If the first case is true, then our assertion is true, 3) If the second case is true, then our assertion is true, then we can conclude that our assertion is true.

In the above code, let's assume that computation has paused at the line *assert: m_i is max in $\{a, b\}$* and that this assertion is true. From here, there are two paths that the computation might follow. If the first path is followed, we are assured that $c > m_i$ and that $m_{i+1} = c$. From these three facts, we can conclude *assert: m_{i+1} is max in $\{a, b, c\}$* . On the other hand, if the second path is followed, we are assured that $c \leq m_i$ and that $m_{i+1} = m_i$. From these three facts, we can also conclude *assert: m_{i+1} is max in $\{a, b, c\}$* .

Proof by Transitivity: If we prove $PreCond \Rightarrow assert_1$, $assert_1 \Rightarrow assert_2$, $assert_2 \Rightarrow assert_3$, and $assert_3 \Rightarrow PostCond$, then by transitively we can conclude that $PreCond \Rightarrow PostCond$.

Loops: Once you see the pattern in the above $Max(a, b, c)$ code, we can generalize each of its steps into a generic step and put that into a loop giving the following code.

```

loop
    assert: m is max in {L[1], L[2], ..., L[j]}.
    j = j + 1
    if( L[j] > m )
        m = L[j]
    end if
end loop

```

We prove this is correct just as we did before. The only difference is that we will be hitting the same assertion twice in a row. The natural thing is to assume that the computation has paused at the top of the loop after having iterated some t number of times. We would differentiate between values by defining j_t and m_t to be the current value of these variables. The problem with this is that it might make you tempted to assume something based on what has come before. Instead, suppose you have just flown in from Mars and absolutely the only thing you know is that you are at the top of the loop, j' and m' are the current value of these variables, and the assertion that m' is max

in $\{L[1], L[2], \dots, L[j']\}$ is true. As before, the *if* gives two cases. When the *if* condition is true, then the assertion from the code is $j'' = j' + 1$; $L[j''] > m'$; and $m'' = L[j'']$. From this you must prove that when the computation gets back to the top of the loop, the assertion will again be true, namely, that m'' is max in $\{L[1], L[2], \dots, L[j'']\}$. You could prove this formally or informally. This proves that once the assertion at the top of the loop has been established, the code will maintain it for as many iterations as needed. We will refer to such assertions as *loop invariants*.

Correctness: Together these techniques formally prove that your code works. Equally important, understanding these steps will help you write correct code in the first place.

Game of Life: Don't worry so much. Taking one step at a time is a lower stress way to live your life.

```
begin routine Life(me)
     $\langle pre - cond \rangle$ : I am born
    Hopefully my parents help
    loop
         $\langle loop - invariant \rangle$ : I am well and reasonably on track
        exit when (I die)
        Maintain the loop invariant
        Make a little progress
        Make the world a little better
    end loop
     $\langle post - cond \rangle$ : It was a well spent and good life
end routine
```

1.2 The Steps to Develop an Iterative Algorithm

Iterative Algorithms: A good way to structure many computer programs is to store the key information you currently know in some data structure and then have each iteration of the main loop take a step toward your destination by making a simple change to this data.

Loop Invariant: A *loop invariant* expresses important relationships among the variables that must be true at the start of every iteration and when the loop terminates. If it is true, then the computation is still on the road. If it is false, then the algorithm has failed.

The Code Structure: The basic structure of the code is as follows.

```
begin routine
     $\langle pre-cond \rangle$ 
     $code_{pre-loop}$  % Establish loop invariant
    loop
         $\langle loop-invariant \rangle$ 
```



```
exit when exit-cond ›
  codeloop % Make progress while maintaining the loop invariant
end loop
codepost-loop % Clean up loose ends
⟨post-cond⟩
end routine
```

Proof of Correctness: Naturally, you want to be sure your algorithm will work on all specified inputs and give the correct answer.

Running Time: You also want to be sure that your algorithm completes in a reasonable amount of time.

The Most Important Steps: If you need to design an algorithm, do not start by typing in code without really knowing how or why the algorithm works. Instead, I recommend first accomplishing the following tasks. See Figure 1.1. These tasks need to fit together in very subtle ways. You may have to cycle through them a number of times, adjusting what you have done, until they all fit together as required.

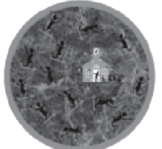
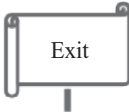
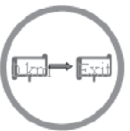
Define Problem 	Define Loop Invariants 	Define Measure of Progress 
Define Step 	Define Exit Condition 	Maintain Loop Inv 
Make Progress 	Initial Conditions  	Ending  

Fig. 1.1 The requirements of an iterative algorithm.

- 1) **Specifications:** What problem are you solving? What are its pre- and postconditions—i.e., where are you starting and where is your destination?
- 2) **Basic Steps:** What basic steps will head you more or less in the correct direction?
- 3) **Measure of Progress:** You must define a measure of progress: where are the distance markers along the road?

4) The Loop Invariant: You must define a loop invariant that will give a picture of the state of your computation when it is at the top of the main loop, in other words, define the road that you will stay on.

5) Main Steps: For every location on the road, you must write the pseudocode *code_{loop}* to take a single step. You do not need to start with the first location. I recommend first considering a typical step to be taken during the middle of the computation.

6) Make Progress: Each iteration of your main step must make progress according to your measure of progress.

7) Maintain Loop Invariant: Each iteration of your main step must ensure that the loop invariant is true again when the computation gets back to the top of the loop. (Induction will then prove that it remains true always.)

8) Establishing the Loop Invariant: Now that you have an idea of where you are going, you have a better idea about how to begin. You must write the pseudocode *code_{pre-loop}* to initially establish the loop invariant. How do you get from your house onto the correct road?

9) Exit Condition: You must write the condition (*exit-cond*) that causes the computation to break out of the loop.

10) Ending: How does the exit condition together with the invariant ensure that the problem is solved? When at the end of the road but still on it, how do you produce the required output? You must write the pseudocode *code_{pre-loop}* to clean up loose ends and to return the required output.

11) Termination and Running Time: How much progress do you need to make before you know you will reach this exit? This is an estimate of the running time of your algorithm.

12) Special Cases: When first attempting to design an algorithm, you should only consider one general type of input instances. Later, you must cycle through the steps again considering other types of instances and special cases. Similarly, test your algorithm by hand on a number of different examples.

13) Coding and Implementation Details: Now you are ready to put all the pieces together and produce pseudocode for the algorithm. It may be necessary at this point to provide extra implementation details.

14) Formal Proof: If the above pieces fit together as required, then your algorithm works.

Example 1.2.1 (The Find-Max Two-Finger Algorithm to Illustrate These Ideas)

1) Specifications: An input instance consists of a list $L(1..n)$ of elements. The output consists of an index i such that $L(i)$ has maximum value. If there are multiple entries with this same value, then any one of them is returned.

2) Basic Steps: You decide on the two-finger method. Your right finger runs down the list.

3) Measure of Progress: The measure of progress is how far along the list your right finger is.

4) The Loop Invariant: The loop invariant states that your left finger points to one of the largest entries encountered so far by your right finger.

5) Main Steps: Each iteration, you move your right finger down one entry in the list. If your right finger is now pointing at an entry that is larger than the left finger's entry, then move your left finger to be with your right finger.

6) Make Progress: You make progress because your right finger moves one entry.

7) Maintain Loop Invariant: You know that the loop invariant has been maintained as follows. For each step, the new left finger element is $\text{Max}(\text{old left finger element}, \text{new element})$. By the loop invariant, this is $\text{Max}(\text{Max}(\text{shorter list}), \text{new element})$. Mathematically, this is $\text{Max}(\text{longer list})$.

8) Establishing the Loop Invariant: You initially establish the loop invariant by pointing both fingers to the first element.

9) Exit Condition: You are done when your right finger has finished traversing the list.

10) Ending: In the end, we know the problem is solved as follows. By the exit condition, your right finger has encountered all of the entries. By the loop invariant, your left finger points at the maximum of these. Return this entry.

11) Termination and Running Time: The time required is some constant times the length of the list.

12) Special Cases: Check what happens when there are multiple entries with the same value or when $n = 0$ or $n = 1$.

13) Coding and Implementation Details:

algorithm *FindMax*(L)

(pre-cond): L is an array of n values.

(post-cond): Returns an index with maximum value.

begin

$i = 1; j = 1$

loop

(loop-invariant): $L[i]$ is max in $L[1..j]$.

exit when $(j \geq n)$

% Make progress while maintaining the loop invariant

$j = j + 1$

if $(L[i] < L[j])$ then $i = j$

end loop

```

    return(i)
end algorithm

```

14) Formal Proof: The correctness of the algorithm follows from the above steps.

A New Way of Thinking: You may be tempted to believe that measures of progress and loop invariants are theoretical irrelevancies. But industry, after many expensive mistakes, has a deeper appreciation for the need for correctness. Our philosophy is to learn how to think about, develop, and describe algorithms in such a way that their correctness is transparent. For this, measures of progress and loop invariants are essential. The description of the preceding algorithms and their proofs of correctness are wrapped up into one.

Keeping Grounded: Loop invariants constitute a life philosophy. They lead to feeling grounded. Most of the code I mark as a teacher makes me feel ungrounded. It cycles, but I don't know what the variables mean, how they fit together, where the algorithm is going, or how to start thinking about it. Loop invariants mean starting my day at home, where I know what is true and what things mean. From there, I have enough confidence to venture out into the unknown. However, loop invariants also mean returning full circle to my safe home at the end of my day.

Exercise 1.2.1 What are the formal mathematical things involving loop invariants that must be proved, to prove that if your program exits then it obtains the postcondition?

1.3 More about the Steps

In this section I give more details about the steps for developing an iterative algorithm.

1) Specifications: Before we can design an iterative algorithm, we need to know precisely what it is supposed to do.

Preconditions: What are the legal input instances? Any assertions that are promised to be true about the input instance are referred to as *preconditions*.

Postconditions: What is the required output for each legal instance? Any assertions that must be true about the output are referred to as *postconditions*.

Correctness: An algorithm for the problem is *correct* if for every legal input instance, the required output is produced. If the input instance does not meet the preconditions, then all bets are off. Formally, we express this as

$$\langle pre-cond \rangle \ \& \ code_{alg} \Rightarrow \langle post-cond \rangle.$$

This correctness is only with respect to the specifications.

Example: The *sorting* problem is defined as follows:

Preconditions: The input is a list of n values, including possible repetitions.

Postconditions: The output is a list consisting of the same n values in nondecreasing order.

The Contract: Pre and postconditions are, in a sense, the contract between the implementer and the user (or invoker) of the coded algorithm.

Implementer: When you are writing a subroutine, you can assume the input comes to your program in the correct form, satisfying all the preconditions. You must write the subroutine so that it ensures that the postconditions hold after execution.

User: When you are using the subroutine, you must ensure that the input you provide meets the preconditions of the subroutine. Then you can trust that the output meets its postconditions.

2) Basic Steps: As a preliminary to designing the algorithm it can be helpful to consider what basic steps or operations might be performed in order to make progress toward solving this problem. Take a few of these steps on a simple input instance in order to get some intuition as to where the computation might go. How might the information gained narrow down the computation problem?

3) Measure of Progress: You need to define a function that, when given the current state of the computation, returns an integer value measuring either how much progress the computation has already made or how much progress still needs to be made. This is referred to either as a *measure of progress* or as a *potential function*. It must be such that the total progress required to solve the problem is not infinite and that at each iteration, the computation makes progress. Beyond this, you have complete freedom to define this measure as you like. For example, your measure might state the amount of the output produced, the amount of the input considered, the extent to which the search space has been narrowed, some more creative function of the work done so far, or how many cases have been tried. [Section 1.4](#) outlines how these different measures lead to different types of iterative algorithms.

4) The Loop Invariant: Often, coming up with the loop invariant is the hardest part of designing an algorithm. It requires practice, perseverance, creativity, and insight. However, from it the rest of the algorithm often follows easily. Here are a few helpful pointers.

Definition: A *loop invariant* is an assertion that is placed at the top of a loop and that must hold true every time the computation returns to the top of the loop.

Assertions: More generally, an *assertion* is a statement made at some particular point during the execution of an algorithm about the current state of the computation's data structures that is either true or false. If it is false, then something has gone wrong in the logic of the algorithm. Pre- and postconditions

are special cases of assertions that provide clean boundaries between systems, subsystems, routines, and subroutines. Within such a part, assertions can also provide checkpoints along the path of the computation to allow everyone to know what should have been accomplished so far. *Invariants* are the same, except they apply either to a loop that is executed many times or to an object-oriented data structure that has an ongoing life.

Designing, Understanding, and Proving Correct: Generally, assertions are not tasks for the algorithm to perform, but are only comments that are added to assist the designer, the implementer, and the reader in understanding the algorithm and its correctness.

Debugging: Some languages allow you to insert assertions as lines of code. If during the execution such an assertion is false, then the program automatically stops with a useful error message. This is helpful both when debugging and after the code is complete. It is what is occurring when an error box pops up during the execution of a program telling you to contact the vendor if the error persists. Not all interesting assertions, however, can be tested feasibly within the computation itself.

Picture from the Middle: A loop invariant should describe what you would like the data structure to look like when the computation is at the beginning of an iteration. Your description should leave your reader with a visual image. Draw a picture if you like.

Don't Be Frightened: A loop invariant need not consist of formal mathematical mumbo jumbo if an informal description gets the idea across better. On the other hand, English is sometimes misleading, and hence a more mathematical language sometimes helps. Say things twice if necessary. I recommend pretending that you are describing the algorithm to a first-year student.

On the Road: A loop invariant must ensure that the computation is still on the road toward the destination and has not fallen into a ditch or landed in a tree.

A Wide Road: Given a fixed algorithm on a fixed input, the computation will follow one fixed line. When the algorithm designer knows exactly where this line will go, they can use a very tight loop invariant to define a very narrow road. On the other hand, because your algorithm must work for an infinite number of input instances and because you may pass many obstacles along the way, it can be difficult to predict where the computation might be in the middle of its execution. In such cases, using a very loose loop invariant to define a very wide road is completely acceptable. The line actually followed by the computation might weave and wind, but as long as it stays within the boundaries of the road and continues to make progress, all is well. An advantage of a wide road is that it gives more flexibility in how the main loop is implemented. A disadvantage is that there are then more places where the computation might be, and for each the algorithm must define how to take a step.

Example: As an example of a loose loop invariant, in the find-max two-finger algorithm, the loop invariant does not completely dictate which entry your left finger should point at when there are a number of entries with the same maximum value.

Meaningful and Achievable: You want a loop invariant that is *meaningful*, meaning it is strong enough that, with an appropriate exit condition, it will guarantee the postcondition. You also want the loop invariant to be *achievable*, meaning you can establish and maintain it.

Know What a Loop Invariant Is: Be clear about what a loop invariant is. It is not code, a precondition, a postcondition, or some other inappropriate piece of information. For example, stating something that is always true, such as “ $1 + 1 = 2$ ” or “The root is the max of any heap,” may be useful information for the answer to the problem, but should not be a part of the loop invariant.

Flow Smoothly: The loop invariant should flow smoothly from the beginning to the end of the algorithm.

- At the beginning, it should follow easily from the preconditions.
- It should progress in small, natural steps.
- Once the exit condition has been met, the postconditions should easily follow.

Ask for 100%: A good philosophy in life is to ask for 100% of what you want, but not to assume that you will get it.

Dream: Do not be shy. What would you like to be true in the middle of your computation? This may be a reasonable loop invariant, and it may not be.

Pretend: Pretend that a genie has granted your wish. You are now in the middle of your computation, and your dream loop invariant is true.

Maintain the Loop Invariant: From here, are you able to take some computational steps that will make progress while maintaining the loop invariant? If so, great. If not, there are two common reasons.

Too Weak: If your loop invariant is too weak, then the genie has not provided you with everything you need to move on.

Too Strong: If your loop invariant is too strong, then you will not be able to establish it initially or maintain it.

No Unstated Assumptions: You don’t want loop invariants that lack detail or are too weak to proceed to the next step. Don’t make assumptions that you don’t state. As a check, pretend that you are a Martian who has jumped into the top of the loop knowing *nothing* that is not stated in the loop invariant.

Example: In the find-max two-finger algorithm, the loop invariant does make some unstated assumptions. It assumes that the numbers above your right finger have been encountered by your right finger and those below it have not. Perhaps more importantly for ± 1 errors is whether or not the

number currently being pointed at has been encountered already. The loop invariant also assumes that the numbers in the list have not changed from their original values.

The Starry Night: How did van Gogh come up with his famous painting, *The Starry Night*? There's no easy answer. In the same way, coming up with loop invariants and algorithms is an art form.

Use This Process: Don't come up with the loop invariant after the fact. Use it to design your algorithm.

5) Main Steps: The pseudocode $code_{loop}$ must be defined so that it can be taken not just from where you think the computation might be, but from any state of the data structure for which the loop invariant is true and the exit condition has not yet been met.

Worry about *one step at a time*. Don't get pulled into the strong desire to understand the entire computation at once. Generally, this only brings fear and unhappiness. I repeat the wisdom taught by both the Buddhists and the 12-step programs: Today you may feel like like you were dropped off in a strange city without knowing how you got there. Do not worry about the past or the future. Be reassured that you are somewhere along the correct road. Your goal is only to take one step so that you make progress and stay on the road. Another analogy is to imagine you are part of a relay race. A teammate hands you the baton. Your job is only to carry it once around the track and hand it to the next teammate.

6) Make Progress: You must prove that progress of at least one unit of your measure is made every time the algorithm goes around the loop. Sometimes there are odd situations in which the algorithm can iterate without making any measurable progress. This is not acceptable. The danger is that the algorithm will loop forever. You must either define another measure that better shows how you are making progress during such iterations or change the step taken in the main loop so that progress is made. The formal proof of this is similar to that for maintaining the loop invariant.

7) Maintain the Loop Invariant: You must prove that the loop invariant is maintained in each iteration.

The Formal Statement: Whether or not you want to prove it formally, the formal statement that must be true is

$$\langle loop\text{-invariant}' \rangle \ \& \ not \langle exit\text{-cond} \rangle \ \& \ code_{loop} \Rightarrow \langle loop\text{-invariant}'' \rangle.$$

Proof Technique

- Assume that the computation is at the top of the loop.
- Assume that the loop invariant is satisfied; otherwise, the program would have already failed. Refer back to the picture that you drew to see what this tells you about the current state of the data structure.

- You can also assume that the exit condition is not satisfied, because otherwise the loop would exit.
- Execute the pseudocode $code_{loop}$, for one iteration of the loop. How does this change the data structure? Or equivalently translate the code $code_{loop} : x = x + 1$ into a math statement $code_{loop} : x'' = x' + 1$ that assures us of some relationship between value x' of x at the beginning of the iteration and that x'' that after going around the loop one more time. You may prefer the notation x_t and x_{t+1} .
- Prove that when you get back to the top of the loop again, the requirements set by the loop invariant are met once more.

Different Situations: Many subtleties can arise from the huge number of different input instances and the huge number of different places the computation might find itself in.

- I recommend first designing the pseudocode $code_{loop}$ to work for a general middle iteration when given a large and general input instance. Is the loop invariant maintained in this case?
- Then try the first and last couple of iterations.
- Also try special case input instances. Before writing separate code for these, check whether the code you already have happens to handle these cases. If you are forced to change the code, be sure to check that the previously handled cases still are handled.
- To prove that the loop invariant is true in all situations, pretend that you are at the top of the loop, but you do not know how you got there. You may have dropped in from Mars. Besides knowing that the loop invariant is true and the exit condition is not, you know nothing about the state of the data structure. Make no other assumptions. Then go around the loop and prove that the loop invariant is maintained.

8) Establishing the Loop Invariant: You must prove that the initial code establishes the loop invariant.

The Formal Statement: The formal statement that must be true is

$$\langle pre-cond \rangle \& code_{pre-loop} \Rightarrow \langle loop-invariant \rangle.$$

Proof Technique

- Assume that you are just beginning the computation.
- You can assume that the input instance satisfies the precondition; otherwise you are not expected to solve the problem.
- Execute the code $code_{pre-loop}$ before the loop.
- Prove that when you first get to the top of the loop, the requirements set by the loop invariant are met.

Easiest Way: Establish the loop invariant in the easiest way possible. For example, if you need to construct a set such that all the dragons within it are

purple, the easiest way to do it is to construct the empty set. Note that all the dragons in this set are purple, because it contains no dragons that are not purple.

Careful: Sometimes it is difficult to know how to set the variables to make the loop invariant initially true. In such cases, try setting them to ensure that it is true after the first iteration. For example, what is the maximum value within an empty list of values? One might think 0 or ∞ . However, a better answer is $-\infty$. When adding a new value, one uses the code $newMax = \max(oldMax, newValue)$. Starting with $oldMax = -\infty$, gives the correct answer when the first value is added.

9) Exit Condition: Generally you exit the loop when you have completed the task.

Stuck: Sometimes, however, though your intuition is that your algorithm designed so far is making progress at each iteration, you have no clue whether, heading in this direction, the algorithm will ever solve the problem or how you would know it if it happens. Because the algorithm cannot make progress forever, there must be situations in which your algorithm gets stuck. For such situations, you must either think of other ways for your algorithm to make progress or have it exit. A good first step is to exit. In step 10, you will have to prove that when your algorithm exits, you actually are able to solve the problem. If you are unable to do this, then you will have to go back and redesign your algorithm.

Loop While vs. Exit When: The following are equivalent:

while(A and B)	loop
...	$\langle loop\text{-invariant} \rangle$
end while	exit when (not A or not B)
	...
	end loop

The second is more useful here because it focuses on the conditions needed to exit the loop, while the first focuses on the conditions needed to continue. Another advantage of the second is that it also allows you to slip in the loop invariant between the top of the loop and the exit condition.

10) Ending: In this step, you must ensure that once the loop has exited you will be able to solve the problem.

The Formal Statement: The formal statement that must be true is

$$\langle loop\text{-invariant}' \rangle \& exit\text{-}cond \& code_{post\text{-}loop} \Rightarrow \langle post\text{-}cond \rangle.$$

Proof Technique

- Assume that you have just broken out of the loop.
- You can assume that the loop invariant is true, because you have maintained that it is always true.

- You can also assume that the exit condition is true by the fact that the loop has exited.
- Execute the code *code_{post-loop}* after the loop to give a few last touches toward solving the problem and to return the result.
- From these facts alone, you must be able to deduce that the problem has been solved correctly, namely, that the postcondition has been established.

11) Termination and Running Time: You must prove that the algorithm does not loop forever. This is done by proving that if the measure of progress meets some stated amount, then the exit condition has definitely been met. (If it exits earlier than this, all the better.) The number of iterations needed is then bounded by this stated amount of progress divided by the amount of progress made each iteration. The running time is estimated by adding up the time required for each of these iterations. For some applications, space bounds (i.e., the amount of memory used) may also be important. We discuss important concepts related to running time in [Chapters 35–38](#): time and space complexity, the useful ideas of logarithms and exponentials, BigOh (O) and Theta (Θ) notation and several handy approximations.

12) Special Cases: When designing an algorithm, you do not want to worry about every possible type of input instance at the same time. Instead, first get the algorithm to work for one general type, then another and another. Though the next type of input instances may require separate code, start by tracing out what the algorithm that you have already designed would do given such an input. Often this algorithm will just happen to handle a lot of these cases automatically without requiring separate code. When adding code to handle a special case, be sure to check that the previously handled cases still are handled.

13) Coding and Implementation Details: Even after the basic algorithm is outlined, there can be many little details to consider. Many of these implementation details can be hidden in abstract data types (see [Chapter 3](#)). If a detail does not really make a difference to an algorithm, it is best to keep all possibilities open, giving extra flexibility to the implementer. For many details, it does not matter which choice you make, but bugs can be introduced if you are not consistent and clear as to what you have chosen. This text does not focus on coding details. This does not mean that they are not important.

14) Formal Proof: Steps 1–11 are enough to ensure that your iterative algorithm works, that is, that it gives the correct answer on all specified inputs. Consider some instance which meets the preconditions. By step 8 we establish the loop invariant the first time the computation is at the top of the loop, and by step 7 we maintain it each iteration. Hence by way of induction, we know that the loop invariant is true every time the computation is at the top of the loop. (See the following discussion.) Hence, by step 5, the step taken in the main loop is always defined and executes without crashing until the loop exits. Moreover, by step 6 each such iteration makes progress of at least one. Hence, by step 11, the exit condition is eventually met. Step 10 then gives that the postcondition is achieved, so that the algorithm works in this instance.

Mathematical Induction: Induction is an extremely important mathematical technique for proving universal statements and is the cornerstone of iterative algorithms. Hence, we will consider it in more detail.

Induction Hypothesis: For each $n \geq 0$, let $S(n)$ be the statement “If the loop has not yet exited, then the loop invariant is true when you are at the top of the loop after going around n times.”

Goal: The goal is to prove that $\forall n \geq 0, S(n)$, namely, “As long as the loop has not yet exited, the loop invariant is always true when you are at the top of the loop.”

Proof Outline: Proof by induction on n .

Base Case: Proving $S(0)$ involves proving that the loop invariant is true when the algorithm first gets to the top of the loop. This is achieved by proving the statement $\langle pre-cond \rangle \ \& \ code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$.

Induction Step: Proving $S(n-1) \Rightarrow S(n)$ involves proving that the loop invariant is maintained. This is achieved by proving the statement $\langle loop-invariant' \rangle \ \& \ \neg \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant'' \rangle$.

Conclusion: By way of induction, we can conclude that $\forall n \geq 0, S(n)$, i.e., that the loop invariant is always true when at the top of the loop.

The Process of Induction

$S(0)$ is true (by base case)

$S(0) \Rightarrow S(1)$ (by induction step, $n = 1$)

hence, $S(1)$ is true

$S(1) \Rightarrow S(2)$ (by induction step, $n = 2$)

hence, $S(2)$ is true

$S(2) \Rightarrow S(3)$ (by induction step, $n = 3$)

hence, $S(3)$ is true ...

Other Proof Techniques: Other formal steps for proving correctness are described in [Chapter 40](#).

Faith in the Method: Convince yourself that these steps are sufficient to define an algorithm so that you do not have to convince yourself every time you need to design an algorithm.

1.4 Different Types of Iterative Algorithms

To help you design a measure of progress and a loop invariant for your algorithm, here are a few classic types, followed by examples of each.

More of the Output: If the solution is a structure composed of many pieces (e.g., an array of integers, a set, or a path), a natural thing to try is to construct the solution one piece at a time.

Measure of Progress: The amount of the output constructed.

Loop Invariant: The output constructed so far is correct.

More of the Input: Suppose the input consists of n objects (e.g., an array of n integers or a graph with n nodes). It would be reasonable for the algorithm to read them in one at a time.

Measure of Progress: The amount of the input considered.

Loop Invariant: Pretending that this prefix of the input is the entire input, I have a complete solution.

Examples: After i iterations of the preceding find-max two-finger algorithm, the left finger points at the highest score within the prefix of the list seen so far. After i iterations of one version of insertion sort, the first i elements of the input are sorted. See [Figure 1.2](#).

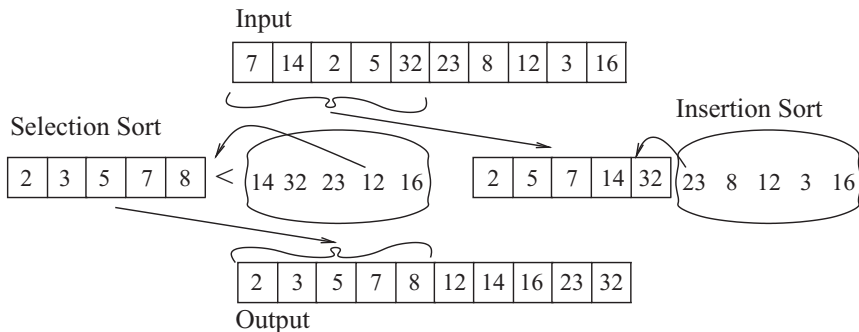


Fig. 1.2 The loop invariants for insertion sort and selection sort are demonstrated.

Bad Loop Invariant: A common mistake is to give the loop invariant “I have handled and have a solution for each of the first i objects in the input.” This is wrong because each object in the input does not need a separate solution; the input as a whole does. For example, in the find-max two-finger algorithm, one cannot know whether one element is the maximum by considering it in isolation from the other elements. An element is only the maximum in comparison with the other elements in the sublist.

Narrowing the Search Space: If you are searching for something, try narrowing the search space, maybe decreasing it by one or, even better, cutting it in half.

Measure of Progress: The size of the space in which you have narrowed the search.

Loop Invariant: If the thing being searched for is anywhere, then it is in this narrowed sublist.

Example: Binary search.

Work Done: The measure of progress might also be some other, more creative function of the work done so far.

Example: Bubble sort measures its progress by how many pairs of elements are out of order.

Case Analysis: Try the obvious thing. For which input instances does it work, and for which does it not work? Now you only need to find an algorithm that works for those later cases. A measure of progress might include which cases you have tried.

We will now give a simple example of each of these. Though you likely know these algorithms already, use them to understand these different types of iterative algorithms and to review the required steps.

Example 1.4.1 (More of the Output—Selection Sort)

1) Specifications: The goal is to rearrange a list of n values in nondecreasing order.

2) Basic Steps: We will repeatedly select the smallest unselected element.

3) Measure of Progress: The measure of progress is the number k of elements selected.

4) The Loop Invariant: The loop invariant states that the selected elements are the k smallest of the elements and that these have been sorted. The larger elements are in a set on the side.

5) Main Steps: The main step is to find the smallest element from among those in the remaining set of larger elements and to add this newly selected element to the end of the sorted list of elements.

6) Make Progress: Progress is made because k increases.

7) Maintain Loop Invariant: We must prove that $\langle \text{loop-invariant}' \rangle$ & $\text{not } \langle \text{exit-cond} \rangle \& \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. By the previous loop invariant, the newly selected element is at least the size of the previously selected elements. By the step, it is no bigger than the elements on the side. It follows that it must be the $k + 1^{\text{st}}$ element in the list. Hence, moving this element from the set on the side to the end of the sorted list ensures that the selected elements in the new list are the $k + 1$ smallest and are sorted.

8) Establishing the Loop Invariant: We must prove that $\langle \text{pre-cond} \rangle \& \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$. Initially, $k = 0$ are sorted and all the elements are set aside.

9) Exit Condition: Stop when $k = n$.

10) Ending: We must prove $\langle \text{loop-invariant} \rangle \ \& \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$. By the exit condition, all the elements have been selected, and by the loop invariant these selected elements have been sorted.

11) Termination and Running Time: We have not considered how long it takes to find the next smallest element or to handle the data structures.

Example 1.4.2 (More of the Input—Insertion Sort)

1) Specifications: Again the goal is to rearrange a list of n values in nondecreasing order.

2) Basic Steps: This time we will repeatedly insert some element where it belongs.

3) Measure of Progress: The measure of progress is the number k of elements inserted.

4) The Loop Invariant: The loop invariant states that the k inserted elements are sorted within a list and that, as before, the remaining elements are off to the side somewhere.

5) Main Steps: The main step is to take any of the elements that are off to the side and *insert* it into the sorted list where it belongs.

6) Make Progress: Progress is made because k increases.

7) Maintain Loop Invariant: $\langle \text{loop-invariant}' \rangle \ \& \ \text{not} \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. You know that the loop invariant has been maintained because the new element is inserted in the correct place in the previously sorted list.

8) Establishing the Loop Invariant: Initially, with $k = 1$, think of the first element in the array as a sorted list of length one.

9) Exit Condition: Stop when $k = n$.

10) Ending: $\langle \text{loop-invariant} \rangle \ \& \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$. By the exit condition, all the elements have been inserted, and by the loop invariant, these inserted elements have been sorted.

11) Termination and Running Time: We have not considered how long it takes to insert the element or to handle the data structures.

Example 1.4.3 (Narrowing the Search Space—Binary Search)

1) Specifications: An input instance consists of a sorted list $A[1 \dots n]$ of elements and a key to be searched for. Elements may be repeated. If the key is in the list, then the output consists of an index i such that $A[i] = \text{key}$. If the key is not in the list, then the output reports this.

2) Basic Steps: Continue to cut the search space in which the key might be in half.

3) Measure of Progress: The measure of progress is the number of elements in our sublist, namely, $j - i + 1$.

4) The Loop Invariant: The algorithm maintains a sublist $A[i \dots j]$ such that if the key is contained in the original list $A[1 \dots n]$, then it is contained in this narrowed sublist. (If the element is repeated, then it might also be outside this sublist.)

5) Main Steps: Each iteration compares the key with the element at the center of the sublist. This determines which half of the sublist the key is not in and hence which half to keep. More formally, let mid index the element in the middle of our current sublist $A[i \dots j]$. If $key \leq A[mid]$, then the sublist is narrowed to $A[i \dots mid]$. Otherwise, it is narrowed to $A[mid + 1 \dots j]$.

6) Make Progress: The size of the sublist decreases by a factor of two.

7) Maintain Loop Invariant: $\langle loop\text{-}invariant' \rangle \ \& \ not \ \langle exit\text{-}cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop\text{-}invariant'' \rangle$. The previous loop invariant gives that the search has been narrowed to the sublist $A[i \dots j]$. If $key > A[mid]$, then because the list is sorted, we know that key is not in $A[1 \dots mid]$ and hence these elements can be thrown away, narrowing the search to $A[mid + 1 \dots j]$. Similarly, if $key < A[mid]$. If $key = A[mid]$, then we could report that the key has been found. However, the loop invariant is also maintained by narrowing the search down to $A[i \dots mid]$.

8) Establishing the Loop Invariant: $\langle pre\text{-}cond \rangle \ \& \ code_{pre\text{-}loop} \Rightarrow \langle loop\text{-}invariant \rangle$. Initially, you obtain the loop invariant by considering the entire list as the sublist. It trivially follows that if the key is in the entire list, then it is also in this sublist.

9) Exit Condition: We exit when the sublist contains one (or zero) elements.

10) Ending: $\langle loop\text{-}invariant \rangle \ \& \ \langle exit\text{-}cond \rangle \ \& \ code_{post\text{-}loop} \Rightarrow \langle post\text{-}cond \rangle$. By the exit condition, our sublist contains at most one element, and by the loop invariant, if the key is contained in the original list, then the key is contained in this sublist, i.e., it must be this one element. Hence, the final code tests to see if this one element is the key. If it is, then its index is returned. If it is not, then the algorithm reports that the key is not in the list.

11) Termination and Running Time: The sizes of the sublists are approximately $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \dots, 8, 4, 2, 1$. Hence, only $\Theta(\log n)$ splits are needed. Each split takes $O(1)$ time. Hence, the total time is $\Theta(\log n)$.

12) Special Cases: A special case to consider is when the key is not contained in the original list $A[1 \dots n]$. Note that the loop invariant carefully takes this case into account. The algorithm will narrow the sublist down to one (or zero) elements. The counter positive of the loop invariant then gives that if the key is not contained in this narrowed sublist, then the key is not contained in the original list $A[1 \dots n]$.

13) Coding and Implementation Details: In addition to testing whether $key \leq A[mid]$, each iteration could test to see if $A[mid]$ is the key. Though finding the key in this way would allow you to stop early, extensive testing shows that this extra comparison slows down the computation.

Example 1.4.4 (Work Done—Bubble Sort)

1) Specifications: The goal is to rearrange a list of n values in nondecreasing order.

2) Basic Steps: Swap elements that are out of order.

3) Measure of Progress: An *involution* is a pair of elements that are out of order, i.e., a pair i, j where $1 \leq i < j \leq n$, $A[i] > A[j]$. Our measure of progress will be the number of involutions in our current ordering of the elements. For example, in $[1, 2, 5, 4, 3, 6]$, there are three involutions.

4) The Loop Invariant: The loop invariant is relatively weak, stating only that we have a permutation of the original input elements.

5) Main Steps: The main step is to find two adjacent elements that are out of order and to swap them.

6) Make Progress: Such a step decreases the number of involutions by one.

7) Maintain Loop Invariant: $\langle \text{loop-invariant}' \rangle \ \& \ \text{not} \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. By the previous loop invariant we had a permutation of the elements. Swapping a pair of elements does not change this.

8) Establishing the Loop Invariant: $\langle \text{pre-cond} \rangle \ \& \ \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$. Initially, we have a permutation of the elements.

9) Exit Condition: Stop when we have a sorted list of elements.

10) Ending: $\langle \text{loop-invariant} \rangle \ \& \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$. By the loop invariant, we have a permutation of the original elements, and by the exit condition these are sorted.

11) Termination and Running Time: Initially, the measure of progress cannot be higher than $n(n-1)/2$ because this is the number of pairs of elements there are. In each iteration, this measure decreases by one. Hence, after at most $n(n-1)/2$ iterations, the measure of progress has decreased to zero. At this point the list has been sorted and the exit condition has been met. We have not considered how long it takes to find two adjacent elements that are out of order.

Example 1.4.5 (Conservation of Energy) Given the specs of a roller coaster, suppose you want to determine how fast the car will be traveling at the end of its ride. Knowing the location and velocity of the car at any instant in time, Newton's laws tell you its location and velocity in the next instant. But this is a lot of complicated computation. Instead, let's define a loop invariant. Newton tells us that mass and energy are conserved. The car's initial energy comes purely from its initial speed and height. The car's final energy, which by conservation will be the same, determines its final speed.

begin routine Physics(rollercoaster)

$\langle \text{pre-cond} \rangle$: Input is a roller coaster with car starting high.

$E_0 = \frac{1}{2}mv_0^2 + h_0m$

loop

```

    ⟨loop – invariant⟩:  $E_t = E_0$ 
    exit when (car at end)
     $\langle \vec{x}_{t+\delta t}, \vec{v}_{t+\delta t} \rangle = \text{Newton}(\langle \vec{x}_t, \vec{v}_t \rangle)$ 
end loop
 $v_{\text{final}} = (2(E_0 - h_{\text{final}})/m)^{\frac{1}{2}}$ 
return( $v_{\text{final}}$ )
⟨post – cond⟩: Speed of car at end is returned.
end routine

```

Example 1.4.6 (Differential Equations) Consider a block bouncing on a spring. We will solve the differential equation expressing its movement. Let $x(t)$ denote its distance from rest at time t , $v(t)$ is velocity, and $a(t)$ its acceleration. We set $x(0) = x_{\text{max}}$ and $v(0) = 0$.

Newton: Newton says that if you push something, it accelerates with $\text{acceleration}(t) = \text{force}(t)/\text{mass}$ and that the farther $x(t)$ you pull the string, the harder it pulls back in the opposite direction with $\text{force}(t) = -k'x(t)$. Let's simplify this with $x''(t) = a(t) = -kx(t)$. This means that if we know how far the spring is pulled, then we know its acceleration.

Calculus: If you took a course on differential equations, you would know that given the three conditions $x''(t) = -kx(t)$, $x(0) = x_{\text{max}}$, and $x'(0) = 0$, the unique solution is the function $x(t) = x_{\text{max}}\cos(\sqrt{k}t)$. If you change the initial conditions to $x(0) = 0$ and $x'(0) = x_{\text{max}}$, then the unique solution changes to $x(t) = x_{\text{max}}\sin(\sqrt{k}t)$. Now let $\vec{x}(t) = \langle x(t), y(t) \rangle$ specify where the block is within a 2-dimensional Euclidean plane. We still have $\vec{x}''(t) = -k\vec{x}(t)$, but now these are all vectors and the spring constantly accelerates the block towards the origin $\langle 0, 0 \rangle$. Suppose the initial conditions are that the block is pulled out a distance r in the x -dimension and is moving at a velocity r up in the y -dimension perpendicular to its location, i.e., $\vec{x}(0) = \langle r, 0 \rangle$ and $\vec{x}'(0) = \langle 0, r \rangle$. The unique solution will be that the block orbits the circle of radius r , namely, $\vec{x}(t) = \langle r \cos(\sqrt{k}t), r \sin(\sqrt{k}t) \rangle$.

Unique Solution: We will prove that these are the unique solutions as follows: We will trace out the function $x(t)$ using loop invariants and in doing so prove that the entire function $x(t)$ is uniquely determined from these conditions, $x''(t) = -kx(t)$, $x(0) = x_{\text{max}}$, and $x'(0) = 0$. Note that each of these conditions holds for the solution $x(t) = x_{\text{max}}\cos(\sqrt{k}t)$. And hence this must be the unique solution. The same is true for the other two examples.

Cycle of Movement: We intuitively know that the block is pulled up and down.

Top: Time $t = 0$. When the block reaches its maximum height $x(t) = x_{\text{max}}$, it stops moving $v(t) = 0$. Because the spring is pulling it, it is accelerating down $a(t) = -a_{\text{max}}$.

Middle: When the block moves down to the center $x(t) = 0$, it is moving its maximum speed down $v(t) = -v_{\text{max}}$. Because the spring is not pulling, it is not accelerating $a(t) = 0$.

Bottom: When the block reaches its lowest height $x(t) = -x_{max}$, it again stops moving $v(t) = 0$. Because the spring is pulling it, it is accelerating up $a(t) = a_{max}$.

Middle: When the block moves up to the center $x(t) = 0$, it is moving its maximum speed up $v(t) = v_{max}$. Because the spring is not pulling, it is not accelerating $a(t) = 0$.

Loop Invariants: Our loop invariant will be that we have plotted distance $x(t)$ from $t = 0$ to T and know $x(T)$ and $v(T)$.

Discrete Values: One problem is that time t is a real value and you can't do induction on the reals! Instead, we will break time into multiples of some infinitesimal ϵ , $T = 0\epsilon, 1\epsilon, 2\epsilon, 3\epsilon, \dots$ and then we will pretend that the solution we obtain is true for the reals in between.

Loop Code: Our task now is to learn $x(T + \epsilon)$ and $v(T + \epsilon)$. We will estimate the plot from T to $T + \epsilon$ with a straight line.

When designing equations, it is helpful to balance units.

Location: If I travel speed $v(T)$ [cm/sec] for time ϵ [sec], then the distance $\Delta x(T)$ I travel is $v(T) \times \epsilon$ [cm]. If I am located at $x(T)$ [cm] and I travel this distance, then where I end up is $x(T + \epsilon)$ [cm] = $x(T)$ [cm] + $v(T)$ [cm/sec] $\times \epsilon$ [sec].

Acceleration: We have that $a(T) = -kx(T)$.

Velocity: If I accelerate $a(T)$ [cm/sec²] for time ϵ [sec], then my increase $\Delta v(T)$ in velocity is $a(T) \times \epsilon$ [cm/sec]. If I current velocity is $v(T)$ [cm/sec] and I increase it by this amount, then my new velocity will be $v(T + \epsilon)$ [cm/sec] = $v(T)$ [cm/sec] + $a(T)$ [cm/sec²] $\times \epsilon$ [sec].

End: Completing the algorithm, we can have the block bounce forever.

Exercise 1.4.1 (See solution in Exercise Solutions.) Give the implementation details and the running times for selection sort.

Exercise 1.4.2 (See solution in Exercise Solutions.) Give the implementation details and the running times for insertion sort. Does using binary search to find the smallest element or to find where to insert help? Does it make a difference whether the elements are stored in an array or in a linked list?

Exercise 1.4.3 (See solution in Exercise Solutions.) Give the implementation details and the running times for bubble sort: Use another loop invariant to prove that the total number of comparisons needed is $O(n^2)$.

1.5 Code from Loop Invariants

It is hard to design an algorithm (or even worse write its code), before you understand the proof of why the algorithm works. It is hard to prove if the algorithm works, before

you know the code. We will first design the loop invariant. From it, we will follow mechanical steps, i.e., minimal thinking, and the code will fall out. We will do an easy example, but the same ideas can help with many problems.

Precond: The input is an integer I and objects $\langle x[1], \dots, x[I] \rangle$.

Postcond: Required output is $S = x[1] + x[2] + x[3] + \dots + x[I]$.

Possible Loop Invariants: You may be tempted to say that the loop invariant is “Each iteration adds in a new object”; however, this is an action, not a picture. Ask yourself what you want to be true in the middle of the computation. The postcondition is that a set of things has been added together. In the middle of the computation, some prefix of them has been added together. Variable i could denote the index of the next object to add in so that the number added so far would be $i - 1$, or i could denote the number added so that the next to add in would be the $i + 1^{\text{st}}$. Though it changes everything, it is a matter of aesthetics. All that matters is that you are consistent! The $\langle i, s \rangle$ is our data structure. The loop invariant tells us a relationship between i and s .

Loop Invariants: $s = x[1] + x[2] + x[3] + \dots + x[i]$.

Starting and Exiting: You might be tempted to worry whether the initial value of i should be 0 or 1. Similarly, does the loop exit when i is $I - 1$, I , or $I + 1$? I implored you to not worry about the entire computation as it is the most common reason for panicking and drowning.

Middle: Jump into the middle of the computation. Say $i = 5$. The fairy godmother assures you the loop invariant is true. Your job is only to make progress and maintain the loop invariant.

Measure of Progress: Clearly our measure of progress is number i of objects added together.

Head in the Right Direction & Return to Path: Because it is hard to know how to make progress and maintain the loop invariant at the same time, my advice is to do them as two separate steps.

Make Progress: Start by heading in the direction that “makes progress” in the easiest way possible. It sounds silly, but if i is the measure of progress, then to make progress the computation only needs to increase i . This gives that $code_{loop}$ contains the code $i = i + 1$.

Reestablishing the Loop Invariant: A Zen master will tell that if your mind wanders while meditating, don’t beat yourself up, simply return to the path. What remains for our computation is to fix the rest of the data structure in the minimal way so that the loop invariant is true again. This is how we determine the rest of the actions $code_{loop}$.

One Step: We must make the loop invariant true. It is easy at this point to try to solve the entire algorithm and to panic! Do not. You must use the fact that the loop invariant had been true in the previous iteration.

Maintaining Loop Invariant: We need to design $code_{loop}$ in a way to make the following true:

$$\langle loop - invariant_t \rangle \ \& \ not \ \langle exit - cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop - invariant_{t+1} \rangle.$$

Let i_t and s_t be the values of i and s at the beginning of the iteration and let i_{t+1} and s_{t+1} be their values after going around again.

Let's work forward from our assumptions and backward from what we want to prove.

Have $\langle loop - invariant_t \rangle$: Knowing that the loop invariant is true for the original values, we can just copy down this statement with these values substituted in, i.e., we know that $s_t = x[1] + x[2] + x[3] + \dots + x[i_t]$.

Have $code_{loop}$: All we know so far is that $i_{t+1} = i_t + 1$.

Want $\langle loop - invariant_{t+1} \rangle$: Wanting that the loop invariant be true for the new values, we can just copy down this statement at the end of our proof, i.e., we need to prove that $s_{t+1} = x[1] + x[2] + x[3] + \dots + x[i_{t+1}]$.

LHS vs. RHS: A typical way to prove that the left hand and right hand sides of an equation are equal is to start at both ends and try to work your way to the middle. We will use both the previous loop invariant and the loop code.

$$\begin{aligned} LHS &= s_{t+1}; \\ RHS &= x[1] + x[2] + x[3] + \dots + x[i_{t+1}], \\ &= [x[1] + x[2] + x[3] + \dots + x[i_{t+1} - 1]] + x[i_{t+1}], \\ &= [x[1] + x[2] + x[3] + \dots + x[i_t]] + x[i_{t+1}], \\ &= [s_t] + x[i_{t+1}]. \end{aligned}$$

Combining LHS and RHS gives that what we want is that $s_{t+1} = s_t + x[i_{t+1}]$. This will complete the proof.

Code: The way that we make the $s_{t+1} = s_t + x[i_{t+1}]$ true is to convert it from being a math statement into being the code $s = s + x[i]$.

Careful: It is tempting to make the code in our loop be $s = s + x[i]$; $i = i + 1$. However, a closer examination shows that this is wrong. The loop invariant clearly states that the last object added was $x[i_t]$. Instead of adding on $x[i_{t+1}]$ next, this code adds $x[i_t]$ a second time. We can fix this by making the code be $s = s + x[i + 1]$; $i = i + 1$ or $i = i + 1$; $s = s + x[i]$.

Ending with Postcondition: We need to design $\langle exit - condition \rangle$ and $code_{post-loop}$ in a way to make the following true: $\langle loop - invariant \rangle \ \& \ \langle exit - cond \rangle \ \& \ code_{post-loop} \Rightarrow \langle post - cond \rangle$. Again let's work forward from our assumptions and backwards from what we want to prove.

Have $\langle loop - invariant \rangle$: We know that $s = x[1] + x[2] + x[3] + \dots + x[i]$.

Want $\langle post - cond \rangle$: We need to prove that $S = x[1] + x[2] + x[3] + \dots + x[I]$.

Clearly, these are the same if $s = S$ and $i = I$.

Exit Condition: We can get $i = I$ to be true when we exit by making our exit condition (*exit* – *cond*) be this.

Post Loop Code: s is the value we are computing. S is the value returned. We can make $s = S$ true with the *code_{post-loop}* by converting the math statement $s = S$ into the code $S = s$.

Establish Loop Invariant: We now must design our pre-loop code so that (*pre* – *cond*) & *code_{pre-loop}* $\Rightarrow$ (*loop* – *invariant*) is true. Winding the code backwards we see that if $i = 1$, we have that $s = x[1]$ and if $i = 0$, we have $s = 0$. Personally, I really like the number zero because after figuring out what the zero-statement means, it is usually easy to make it true. Our *code_{pre-loop}* will be $i = 0; s = 0$.

Harder Example: [Exercise I.9](#) in [Chapter 9](#) will use these same steps to develop both the elementary and the high school algorithm for multiplying integers $X \times Y$.

1.6 Typical Errors

In a study, a group of experienced programmers was asked to code binary search. Easy, yes? But 80% got it wrong! My guess is that if they had used loop invariants, they all would have got it correct.

Be Clear: The code specifies the current subinterval $A[i \dots j]$ with two integers i and j . Clearly document whether the sublist includes the end points i and j or not. It does not matter which, but you must be consistent. Confusion in details like this is the cause of many bugs.

Math Details: Small math operations like computing the index of the middle element of the subinterval $A[i, j]$ are prone to bugs. Check for yourself that the answer is $mid = \lfloor \frac{i+j}{2} \rfloor$.

6) Make Progress: Be sure that in each iteration progress is made in every special case. For example, in binary search, when the current sublist has even length, it is reasonable (as done above) to let *mid* be the element just to the left of center. It is also reasonable to include the middle element in the right half of the sublist. However, together these cause a bug. Given the sublist $A[i \dots j] = A[3, 4]$, the middle will be the element indexed with 3, and the right sublist will be still be $A[mid \dots j] = A[3, 4]$. If this sublist is kept, no progress will be made, and the algorithm will loop forever.

7) Maintain Loop Invariant: Be sure that the loop invariant is maintained in every special case. For example, in binary search, it is reasonable to test whether $key < A[mid]$ or $key \geq A[mid]$. It is also reasonable for it to cut the sublist $A[i \dots j]$ into $A[i \dots mid]$ and $A[mid + 1 \dots j]$. However, together these cause a bug. When key and $A[mid]$ are equal, the test $key < A[mid]$ will fail, causing the algorithm to think the key is bigger and to keep the right half $A[mid + 1 \dots j]$. However, this skips over the key.

Simple Loop: Code like “ $i = 1$; while $(i \leq n) A[i] = 0; i = i + 1$; end while” is surprisingly prone to the error of being off by one. The loop invariant “When at the top of the loop, i indexes the next element to handle” helps a lot.

Exercise 1.6.1 (See solution in Exercise Solutions.) You are now the professor. Which of the steps to develop an iterative algorithm did the student fail to do correctly in the following code? How? How would you fix it?

algorithm $Eg(I)$

⟨pre-cond⟩: I is an integer.

⟨post-cond⟩: Outputs $\sum_{j=1}^I j$.

begin

$s = 0$

$i = 1$

while $(i \leq I)$

⟨loop-invariant⟩: Each iteration adds the next term giving that $s = \sum_{j=1}^i j$.

$s = s + i$

$i = i + 1$

end loop

return (s)

end algorithm

1.7 Exercises

Exercise 1.7.1 You are in the middle of a lake of radius 1. You can swim at a speed of 1 and can run infinitely fast. There is a smart monster on the shore who can’t go in the water but can run at a speed of 4. Your goal is to swim to shore, arriving at a spot where the monster is not, and then run away. If you swim directly to shore, it will take you 1 time unit. In this time, the monster will run the distance $\Pi < 4$ around to where you land and eat you. Your better strategy is to maintain the most obvious loop invariant while increasing the most obvious measure of progress for as long as possible and then swim for it. Describe how this works.

Exercise 1.7.2 Given an undirected graph G such that each node has at most $d + 1$ neighbors, color each node with one of $d + 1$ colors so that for each edge the two nodes have different colors. Hint: Don’t think too hard. Just color the nodes. What loop invariant do you need?

2 Examples Using More-of-the-Input Loop Invariants

We are now ready to look at more examples of iterative algorithms. For each example, look for the key steps of the loop invariant paradigm. What is the loop invariant? How is it obtained and maintained? What is the measure of progress? How is the correct final answer ensured?

In this chapter, we will encounter some of those algorithms that use the more-of-the-input type loop invariant. The algorithm reads the n objects making up the input one at a time. After reading the first i of them, the algorithm temporarily pretends that this prefix of the input is in fact the entire input. The loop invariant is “I currently have a solution for the input consisting solely of these first i objects (and maybe some additional information).” In [Section 2.3](#), we also encounter some algorithms that use the more-of-the-output type of loop invariant.

2.1 Coloring the Plane

See [Figure 2.1](#).

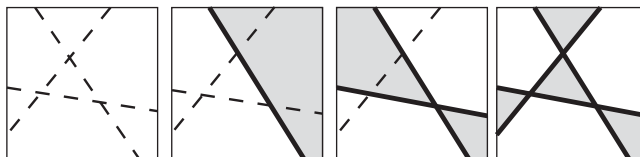


Fig. 2.1 An example of coloring the plane.

1) Specifications: An input instance consists of a set of n (infinitely long) lines. These lines form a subdivision of the plane, that is, they partition the plane into a finite number of regions (some of them unbounded). The output consists of a coloring of each region with either black or white so that any two regions with a common boundary have different colors. An algorithm for this problem proves the theorem that such a coloring exists for any such subdivision of the plane.

2) Basic Steps: When an instance consists of a set of objects, a common technique is to consider them one at a time, incrementally solving the problem for those objects considered so far.

3) Measure of Progress: The measure of progress is the number of lines, i , that have been considered.

4) The Loop Invariant: We have considered the first i lines. Here C is a proper coloring of the plane subdivided by these lines.

5) Main Steps: We have a proper coloring C for the first i lines. Line $i + 1$ cuts the plane in half, cutting many regions in half. Each of these halves needs a different color. Then when we change the color of one region, its neighbors must change color too. We will accomplish this by keeping all the colors on one side of line $i + 1$ the same and flipping those on the other side from white to black and from black to white.

6) Make Progress: Each iteration increases i by one.

7) Maintain Loop Invariant: $\langle \text{loop-invariant}' \rangle \ \& \ \text{not}(\text{exit-cond}) \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. We need to check each boundary to make sure that the regions on either side have opposite colors. Boundaries formed by one of the first i lines had opposite colors before the change. The colors of the regions on either side were either neither flipped or both flipped. Hence, they still have opposite colors. Boundaries formed by line $i + 1$ had the same colors before the change. One of these colors was flipped, the other not. Hence, they now have opposite colors.

8) Establishing the Loop Invariant: $\langle \text{pre-cond} \rangle \ \& \ \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$. With $i = 0$ lines, the plane is all one region. The coloring that makes the entire plane white works.

9) Exit Condition: Exit when all n lines have been considered, i.e., $i = n$.

10) Ending: $\langle \text{loop-invariant} \rangle \ \& \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$. If C is a proper coloring given the first i lines and $i = n$, then clearly C is a proper coloring given all of the lines.

11) Termination and Running Time: Clearly, only n iterations are needed.

12) Special Cases: There are no special cases we need to consider.

13) Coding and Implementation Details

algorithm *ColoringPlane*(*lines*)

$\langle \text{pre-cond} \rangle$: *lines* specifies n (infinitely long) lines.

$\langle \text{post-cond} \rangle$: C is a proper coloring of the plane subdivided by the lines.

begin

C = “the coloring that colors the entire plane white.”

$i = 0$

loop

$\langle \text{loop-invariant} \rangle$: C is a proper coloring of the plane subdivided by the first i lines.

exit when ($i = n$)

% Make progress while maintaining the loop invariant

```

    Line  $i + 1$  cuts the plane in half.
    On one half, the new coloring  $C'$  is the same as the old one  $C$ .
    On the other half, the new coloring  $C'$  is the same as the old one  $C$ ,
        except white is switched to black and black to white.
     $i = i + 1$  &  $C = C'$ 
end loop
return ( $C$ )
end algorithm

```

2.2 Deterministic Finite Automaton

One large class of problems that can be solved using an iterative algorithm with the help of a loop invariant is the class of *regular languages*. You may have learned that this is the class of languages that can be decided by a deterministic finite automata (DFA) or described using a regular expression.

Applications: This class is useful for modeling:

- simple iterative algorithms,
- simple mechanical or electronic devices like elevators and calculators,
- simple processes like the job queue of an operating system, and
- simple patterns within strings of characters.

Features: All of these have the following similar features:

Input Stream: They receive a stream of information to which they must react. For example, the stream of input for a simple algorithm consists of the characters read from input; for a calculator, it is the sequence of buttons pushed; for the job queue, it is the stream of jobs arriving; and for the pattern within a string, one scans the string once from left to right.

Read-Once Input: Once a token of the information has arrived, it cannot be requested again.

Bounded Memory: The algorithm, device, process, or pattern matcher has limited memory with which to remember the information that it has seen so far. Though the amount of memory can be any fixed amount, this amount cannot grow even if the input instance becomes really big.

Example 2.2.1 (A Simple DFA)

1) **Specifications:** Given a string α as the input instance, determine whether it is contained in the set (*language*)

$$L = \{\alpha \in \{0, 1\}^* \mid \alpha \text{ has length at most three and the number of 1s is odd}\}.$$

In most, but not all, DFAs, the computation's task is to either accept or reject the input.

2) Basic Steps: The characters of the input instance are read one at a time. Because the computation will never be able to read a character again, it must remember what it needs about what it has read so far.

3) Measure of Progress: The measure of progress is the number of characters read so far.

4) The Loop Invariant: Let ω denote the prefix of the input instance read so far. The loop invariant states what information is remembered about it. Its length and the number of 1s read so far cannot be remembered with a bounded amount of memory, because these counts would grow arbitrarily large were the input instance to grow arbitrarily long. Luckily, the language is only concerned with this length up to three and whether the number of 1s is even or odd. This can be accommodated with two variables: length, $l \in \{0, 1, 2, 3, \text{more}\}$, and parity, $r \in \{\text{even}, \text{odd}\}$. This requires only a fixed amount of memory.

5) Main Steps: Read a character, and update what we know about the prefix.

6) Make Progress: Progress increases because the number of characters read so far increases by one.

7) Maintaining the Loop Invariant: After reading another character c , the prefix read is now ωc . We know that the length of ωc is one more than that of ω and that the number of 1s is either one more mod 2 or the same, depending on whether or not the new character c is a 1.

8) Establishing the Loop Invariant: At the beginning of the computation, the prefix that has been read so far is the empty string $\omega = \epsilon$, whose length is $l = 0$, and the number of 1s is $r = \text{even}$.

9) Exit Condition: We exit when the entire input instance has been read.

10) Ending: When the input instance has been completely read in, the knowledge that the loop invariant states what we know is sufficient for us to compute the final answer. We accept if the instance has length at most three and the number of 1s is odd.

11) Termination and Running Time: The number of iterations is clearly the length n of the input instance.

12) Special Cases: There are no special cases we need to consider.

13) Coding and Implementation Details:

algorithm *DFA*()

(pre-cond): The input instance α will be read in one character at a time.

(post-cond): The instance will be *accepted* if it has length at most three and the number of 1s is odd.

```

begin
   $l = 0$  and  $r = \text{even}$ 
  loop
    ⟨loop-invariant⟩: When the iterative program has read in some
      prefix  $\omega$  of the input instance  $\alpha$ , the bounded memory of the
      machine remembers the length  $l \in \{0, 1, 2, 3, \text{more}\}$  of this prefix
      and whether the number of 1's in it is  $r \in \{\text{even}, \text{odd}\}$ .
    exit when end of input
    get( $c$ )  % Reads next character of input
    if( $l < 4$ ) then  $l = l + 1$ 
    if( $c = 1$ ) then  $r = (r + 1) \bmod 2$ 
  end loop
  if( $l < 4$  AND  $r = \text{odd}$ ) then
    accept
  else
    reject
  end if
end algorithm

```

Mechanically Compiling an Iterative Program into a DFA: Any iterative program with bounded memory and an input stream can be mechanically compiled into a DFA that solves the same problem. This provides another model or notation for understanding the algorithm. A DFA is specified by $M = \langle \Sigma, Q, \delta, s, F \rangle$.

Alphabet Σ —Precondition: The precondition of the problem provides an alphabet Σ of characters and specifies that any string of these characters is a valid input instance. This may be $\{a, b\}$, $\{a, b, \dots, z\}$, ASCII, or any other finite set of tokens that the program may input. In [Example 2.2.1](#), which we are continuing, $\Sigma = \{0, 1\}$.

Set of States, Q —The Loop Invariant: The loop invariant states what information is remembered about the prefix ω read so far. A discrete way of stating this loop invariant is by constructing the set Q of different states that this remembering iterative program might be in when at the top of the loop. In this example, these states are $Q = \{q(l=0, r=\text{even}), q(l=0, r=\text{odd}), \dots, q(l=\text{more}, r=\text{odd})\}$, because at each point in time the computation remembers both the length $l \in \{0, 1, 2, 3, \text{more}\}$ and parity $r \in \{\text{even}, \text{odd}\}$ of the prefix read.

Recall that a restriction that we are imposing on DFAs is that the amount of memory that they use is fixed and cannot grow even if the input instance becomes really big. The consequence of this is that the number $|Q|$ of states that the DFA might be in is fixed to some finite number.

Each state $q \in Q$ of the DFA specifies a value for each of the program's variables. If the variables are allocated in total r bits of memory, then there are $|Q| = 2^r$ different states that these variable might be in. Conversely, with $|Q|$

states, a DFA can remember $r = \log_2 |Q|$ bits of information. If the algorithm has two variables, one with $|Q_1|$ different states and one with $|Q_2|$, then the algorithm can be in $|Q| = |Q_1| \times |Q_2|$ different states.

In our example, $|Q| = 5 \times 2 = 10$.

Be sure to assign meaningful names to the states, i.e., not $q_0, q_1, \dots, q_{|Q|}$, as I have often seen.

Sometimes, when tightening up the algorithm, some of these states can be collapsed into one, if there is no need for the algorithm to differentiate between them. In our example, there are three states that can be collapsed into a dead state from which the final answer is known to be *reject*. Also, the state $q_{(l=0, r=odd)}$ should be deleted because it is impossible to be in it.

Graphical Representation: Because the number of states that the computation might be in is fixed to some finite number, the DFA can be represented graphically by having one node for each state, as in Figure 2.2.

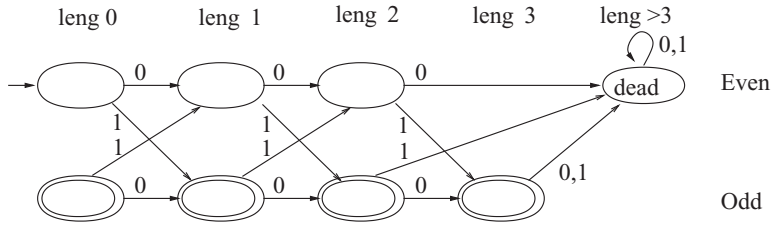


Fig. 2.2 Graphical representation of the DFA.

Transition Function δ —Maintain Loop Invariant: Suppose that the computation has read the prefix ω and is at top of the loop. By the loop invariant, the DFA will be remembering something about this prefix ω and a result will be in some state $q \in Q$. After reading another character c , the prefix read is now ωc . We maintain the loop invariant by putting the DFA into the state q' corresponding to what it is to remember about this new prefix ωc . Because the DFA does not know anything about the present prefix ω other than the fact that it is in state q , the next state that the DFA will be in can depend only on its present state q and on the next character c read.

The DFA's transition function δ defines how the machine transitions from state to state. Formally, it is a function $\delta : Q \times \Sigma \rightarrow Q$. If the DFA's current state is $q \in Q$ and the next input character is $c \in \Sigma$, then the next state of the DFA is given by $q' = \delta(q, c)$. Consider some state $q \in Q$ and some character $c \in \Sigma$. Set the program's variables to the values corresponding to state q , assume the character read is c , and execute the code once around the loop. The new state $q' = \delta(q, c)$ of the DFA is defined to be the state corresponding to the values of the program's variables when the computation has reached the top of the loop again.

In a graphical representation of a DFA, for each state q and character c , there is an edge labeled c from node q to node $q' = \delta(q, c)$.

The Start State s —Establishing the Loop Invariant: The start state s of the DFA M is the state in Q corresponding to the initial values that the program

assigns to its variables before reading any input characters. In Figure 2.2, the corresponding node has an arrow pointing to it.

Accept States F —Ending: When the input instance has been completely read in, the DFA might be in any one of the states $q \in Q$. Because the DFA does not know anything about the input instance other than the fact that it is in state q , the result of the computation can only depend on this state. If the task of the DFA is only to either accept or reject the input instance, then the set of states Q must be partitioned into *accept* and *reject* states. If the DFA is in an accept state when the instance ends, then the instance is accepted. Otherwise, it is rejected. When the DFA is specified by $M = \langle \Sigma, Q, \delta, s, F \rangle$, F denotes the set of these accept states. In Figure 2.2, these nodes are denoted by double circles.

Example 2.2.2 (Addition) In the standard elementary school algorithm for addition, the input consists of two integers x and y represented as strings of digits. The output is the sum z , also represented as a string of digits. The input can be viewed as a stream if the algorithm is first given the lowest digits of x and of y , then the second lowest, and so on. The algorithm outputs the characters of z as it proceeds. The only memory required is a single bit to store the carry bit. Because of these features, the algorithm can be modeled as a DFA.

algorithm *Adding*()

⟨pre-cond⟩: The digits of two integers x and y are read in backward in parallel.

⟨post-cond⟩: The digits of their sum will be outputted backward.

```

begin
  allocate carry  $\in \{0, 1\}$ 
  carry = 0
  loop
    ⟨loop-invariant⟩: If the low-order  $i$  digits of  $x$  and of  $y$  have been
      read, then the low-order  $i$  digits of the sum  $z = x + y$  have
      been outputted. The bounded memory of the machine
      remembers the carry.
    exit when end of input
    get( $\langle x_i, y_i \rangle$ )
     $s = x_i + y_i + \textit{carry}$ 
     $z_i = \text{low order digit of } s$ 
     $\textit{carry} = \text{high order digit of } s$ 
    put( $z_i$ )
  end loop
  if(carry = 1) then
    put(carry)
  end if
end algorithm

```

The DFA is as follows:

Set of States: $Q = \{q_{\langle \textit{carry}=0 \rangle}, q_{\langle \textit{carry}=1 \rangle}\}$.

Alphabet: $\Sigma = \{\langle x_i, y_i \rangle \mid x_i, y_i \in [0 \dots 9]\}$.

Start State: $s = q_{\langle \text{carry}=0 \rangle}$.

Transition Function: $\delta(q_{\langle \text{carry}=c \rangle}, \langle x_i, y_i \rangle) = \langle q_{\text{carry}=c'}, z_i \rangle$

where c' is the high-order digit and z_i is the low order digit of $x_i + y_i + c$.

Example 2.2.3 (Division) Dividing an integer by seven requires a fairly complex algorithm. Surprisingly, it can be done by a DFA. The input consists of an integer x read in one digit at a time, starting with the high-order digit. Simultaneously, the digits of the output $\lfloor \frac{x}{7} \rfloor$ are outputted. In the end, the remainder is provided. Try computing $\lfloor \frac{39591}{7} \rfloor = 5655$. After the prefix $\omega = 395$ has been read, the loop invariant states that the answer $z = \lfloor \frac{395}{7} \rfloor = 56$ has been outputted, and its remainder $r = 395 \bmod 7 = 3 \in \{0, 1, \dots, 6\}$ has been remembered. When the next character $x_i = 9$ is read, we must do the same for $\omega 9 = 3959$. The new answer is

$$\begin{aligned} z' &= \left\lfloor \frac{3959}{7} \right\rfloor = \left\lfloor \frac{395 \times 10 + 9}{7} \right\rfloor = \left\lfloor \frac{(\lfloor \frac{395}{7} \rfloor \times 7 + r) \times 10 + 9}{7} \right\rfloor \\ &= \left\lfloor \frac{395}{7} \right\rfloor \times 10 + \left\lfloor \frac{r \times 10 + 9}{7} \right\rfloor = z \times 10 + \left\lfloor \frac{3 \times 10 + 9}{7} \right\rfloor \\ &= 56 \times 10 + 5 = 565. \end{aligned}$$

In general, for $r \in \{0, 1, \dots, 6\}$ and $x_i \in \{0, 1, \dots, 9\}$, the value $z_i = \lfloor \frac{r \times 10 + x_i}{7} \rfloor$ is a single digit that is easy to compute. This gives that z' as a string is z concatenated with this new digit z_i . Given that z has already been outputted, what remains is to output z_i . Similarly, the new remainder $r' = 3959 \bmod 7 = (395 \times 10 + 9) \bmod 7 = ((395 \bmod 7) \times 10 + (9 \bmod 7)) \bmod 7 = ((3) \times 10 + (2)) \bmod 7 = 32 \bmod 7 = 4$. More generally, $r' = r \times 10 + c \bmod 7$. Initially, the prefix read so far is the empty string ω representing 0, giving $z = \lfloor \frac{0}{7} \rfloor = 0 =$ empty string and $r = 0 \bmod 7 = 0$. In the end, $z = \lfloor \frac{x}{7} \rfloor$ has been outputted, and what remains is to output its remainder r . The DFA to compute this will have seven states $q_0, \dots, q_6$. The transition function is $\delta(q_r, c) = q_{\langle r \cdot 10 + c \bmod 7 \rangle}$.

Example 2.2.4 (Calculator) Invariants can be used to understand a computer system that, instead of simply computing one function, continues dynamically to take in inputs and produce outputs. In our simple calculator, the keys are limited to $\Sigma = \{0, 1, 2, \dots, 9, +, clr\}$. You can enter a number. As you do so, it appears on the screen. The $+$ key adds the number on the screen to the accumulated sum and displays the sum on the screen. The clr key resets both the screen and the accumulator to zero. The machine only can store positive integers from zero to 99999999. Additions are done mod 10^8 .

Set of States: $Q = \{q_{\langle \text{acc}, \text{cur}, \text{scr} \rangle} \mid \text{acc}, \text{cur} \in \{0 \dots 10^8 - 1\} \text{ and } \text{scr} \in \{\text{showA}, \text{showC}\}\}$.

There are $10^8 \times 10^8 \times 2$ states in this set, so you would not want to draw the diagram.

Alphabet: $\Sigma = \{0, 1, 2, \dots, 9, +, clr\}$.

Start State: $s = q_{\langle 0, 0, \text{showC} \rangle}$.

Transition Function

- For $c \in \{0 \dots 9\}$, $\delta(q_{\langle \text{acc}, \text{cur}, \text{scr} \rangle}, c) = q_{\langle \text{acc}, 10 \times \text{cur} + c, \text{showC} \rangle}$.
- $\delta(q_{\langle \text{acc}, \text{cur}, \text{scr} \rangle}, +) = q_{\langle \text{acc} + \text{cur}, \text{cur}, \text{showA} \rangle}$.
- $\delta(q_{\langle \text{acc}, \text{cur}, \text{scr} \rangle}, \text{clr}) = q_{\langle 0, 0, \text{showC} \rangle}$.

Example 2.2.5 (Longest Block of Ones)

Suppose that the input consists of a sequence $A[1 \dots n]$ of zeros and ones, and we want to find a longest contiguous block $A[p, q]$ of ones. For example, on input $A[1 \dots n] = [1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0]$, the block $A[5 \dots 7]$ of length 3 is a suitable solution, and so is the block $A[1 \dots 12]$. Here are some things we must consider when designing the loop invariant.

Nonfinite Memory: Both the size of the longest block and the indices of its beginning and end are integers in $[1 \dots n]$. These require $O(\log n)$ bits to remember. Hence, this algorithm will not be a deterministic *finite* automaton.

Remember the Solution for the Prefix: After reading the prefix $A[1 \dots i]$, it is clear that you need to remember the longest block. Is this enough for a loop invariant? How would you maintain this loop invariant when reading in only the next character $A[i + 1]$? For example, if $A[1 \dots i] = [0, 1, 1, 0, 0, 1, 1]$, the loop invariant may give us only the block $A[2 \dots 3]$ length 2. Then if we read $A[i + 1] = 1$, then the longest block of $A[1 \dots i + 1] = [0, 1, 1, 0, 0, 1, 1, 1]$ becomes $A[6 \dots 8]$ of length 3. How would your program know about this block?

Remember the Longest Current Block: You also must keep a pointer to the beginning of the current block being worked on, i.e., the longest one ending in the value $A[i]$, and its size. With this, the algorithm can know whether the current increasing contiguous subsequence gets to be longer than the previous one. This needs to be included in the loop invariant.

Maintaining the Loop Invariant: If you have this information about $A[1 \dots i]$, then you can learn it about $A[1 \dots i + 1]$ as follows. If $A[i + 1] = 1$, then the longest block of ones ending in the current value increases in length by one. Otherwise, it shrinks to being the empty string. If this block increases to be longer than our previous longest, then it replaces the previous longest. In the end, we know the longest block of ones.

Empty Blocks: $A[3 \dots 3]$ is a block of length 1, and $A[4 \dots 3]$ is a block of length zero ending in $A[3]$. This is why initially, with $i = 0$, the blocks are set to $A[1 \dots 0]$, and when the current block ending in $A[i + 1]$ becomes empty, it is set to $A[i + 2 \dots i + 1]$.

Dynamic Programming: Dynamic programming, covered in [Chapter 24](#), is a very powerful technique for solving optimization problems. Many of these amount to reading the elements of the input instance $A[1 \dots n]$ one at a time and, when at $A[i]$,

saving the optimal solution for the prefix $A[1 \dots i]$ and its cost. This amounts to a deterministic nonfinite automaton. The maximum-block-of-ones problem is a trivial example of this. The solutions to the following two problems and more problems can be found in [Section 24.5](#).

Longest Increasing Contiguous Subsequence: The input consists of a sequence $A[1 \dots n]$ of integers, and we want to find the longest contiguous subsequence $A[k_1 \dots k_2]$ such that the elements are monotonically increasing. For example, the optimal solution for $[5, 3, 1, 3, 7, 9, 8]$ is $[1, 3, 7, 9]$.

Longest Increasing Subsequence: This is harder problem. Again the input consists of a sequence A of integers of size n . However, now we want to find the longest (not necessarily contiguous) subsequence $S \subseteq [1 \dots n]$ such that the elements, in the order that they appear in A , are monotonically increasing. For example, an optimal solution for $[5, 1, 5, 7, 2, 4, 9, 8]$ is $[1, 5, 7, 9]$, and so is $[1, 2, 4, 8]$.

Exercise 2.2.1 (See solution in Exercise Solutions.) Give the code for these examples:

1. Divide.
2. Calculator.
3. Longest block of ones.

Exercise 2.2.2 For the longest block of ones, what are Σ , Q , δ , s , and F ?

Exercise 2.2.3 For each of the following examples, give the code, and either give the DFA or, if necessary, give a deterministic nonfinite automaton as done in [Example 2.2.5](#).

1. $L = \{0^n 1^n | n \geq 0\} = \{\alpha \in \{0, 1\}^* | \alpha \text{ has zero or more zeros followed by the same number of ones}\}$.
2. $L = \{\alpha \in \{0, 1\}^* | \text{every third character of } \alpha \text{ is a } 1\}$. e.g., $1010011110110 \in L$, $\epsilon \in L$, $0 \in L$, and $100 \notin L$.
3. $L_{OR} = \{\alpha \in \{0, 1\}^* | \alpha \text{ has length at most three OR the number of } 1\text{'s is odd}\}$.
4. $L = \{\alpha \in \{0, 1\}^* | \alpha \text{ contains the substring } 0101\}$. For example, $\alpha = 1110101101 \in L$, because it contains the substring, namely, $\alpha = 111 \underline{0101} 101$.

2.3 More of the Input vs. More of the Output

Sometimes it is not clear at first whether to use more-of-the-input or more-of-the-output loop invariants. This section gives two similar problems, of which the first works better for one and in which the second works better for the other.

Example 2.3.1 (Tournament) A *tournament* is a directed graph (see [Section 3.1](#)) formed by taking the complete undirected graph and assigning arbitrary directions to the edges, i.e., a graph $G = (V, E)$ such that for each $u, v \in V$, exactly one of $\langle u, v \rangle$ or $\langle v, u \rangle$ is in E . A *Hamiltonian path* is a path through a graph that can start and finish

anywhere but must visit every node exactly once each. Design an algorithm that, given any tournament, finds a Hamiltonian path through it. Because it finds a Hamiltonian path for any tournament, this algorithm, in itself, acts as proof that every tournament has a Hamiltonian path.

More of the Output: It is natural to want to push forward and find the required path through a graph. The measure of progress would be the amount of the path outputted and the loop invariant would say “I have the first i nodes (or edges) in the final path.” Maintaining this loop invariant would require extending the path constructed so far by one more node. The problem, however, is that the algorithm might get stuck when the path constructed so far has no edges from the last node to a node that has not yet been visited. This makes the loop invariant as stated false.

Recursive Backtracking: One is then tempted to have the algorithm *backtrack* when it gets stuck, trying a different direction for the path to go. This results in a fine algorithm. See the recursive backtracking algorithms in [Chapter 23](#). However, unless one is really careful, such algorithms tend to require exponential time.

More of the Input: Instead, try solving this problem using a more-of-the-input loop invariant. Assume the nodes are numbered 1 to n in an arbitrary way. The algorithm temporarily pretends that the subgraph on the first i of the nodes is the entire input instance. The loop invariant is “I currently have a solution for this subinstance.” Such a solution is a Hamiltonian path $u_1, \dots, u_i$ that visits each of the first i nodes exactly once and that itself is simply a permutation the first i nodes. Maintaining this loop invariant requires constructing a path for the first $i + 1$ nodes. There is no requirement that this new path resemble the previous path. For this problem, however, it can be accomplished by finding a place to insert the $i + 1^{\text{st}}$ node within the permutation of the first i nodes. In this way, the algorithm looks a lot like insertion sort.

Case Analysis: When developing an algorithm, a good technique is to see for which input instances the obvious thing works and then try to design another algorithm for the remaining cases ([Figure 2.3](#)):

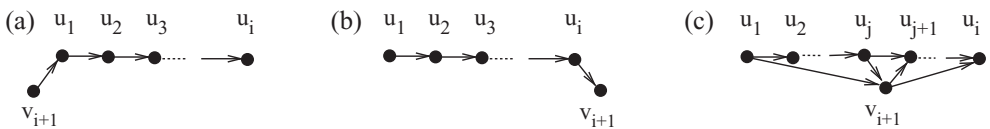


Fig. 2.3 Case analysis for [Example 2.3.1](#).

(a) If $\langle v_{i+1}, u_1 \rangle$ is an edge, then the extended path is easily $v_{i+1}, u_1, \dots, u_i$.

(b) Similarly, if $\langle u_i, v_{i+1} \rangle$ is an edge, then the extended path is easily $u_1, \dots, u_i, v_{i+1}$.

(c) Otherwise, because the graph is a tournament, both $\langle u_1, v_{i+1} \rangle$ and $\langle v_{i+1}, u_i \rangle$ are edges. Color each node u_j red if $\langle u_j, v_{i+1} \rangle$ is an edge, and blue if $\langle v_{i+1}, u_j \rangle$ is. Because u_1 is red and u_i is blue, there must some place u_j to u_{j+1} in the path where it changes

color from red to blue. Because both $\langle u_j, v_{i+1} \rangle$ and $\langle v_{i+1}, u_{j+i} \rangle$ are edges, we can form the extended path $u_1, \dots, u_j, v_{i+1}, u_{j+i}, \dots, u_i$.

Example 2.3.2 (Euler Cycle) An *Eulerian cycle* in an undirected graph is a cycle that passes through each edge exactly once. A graph contains an Eulerian cycle if it is connected and the degree of each vertex is even. Given such a graph, find such a cycle.

More of the Output: We will again start by attempting to solve the problem using the more-of-the-output technique, namely, start at any node and build the output path one edge at a time. Not having any real insight into which edge should be taken next, we will choose them in a blind or greedy way (see [Chapter 22](#)). The loop invariant is that after i steps you have some path through i different edges from some node s to some node v .

Getting Stuck: The next step in designing this algorithm is to determine when, if ever, this simple blind algorithm gets stuck, and either to figure out how to avoid this situation or to fix it.

Making Progress: If $s \neq v$, then the end node v must be adjacent to an odd number of edges that are in the path. See [Figure 2.4.a](#). This is because there is the last edge in the path, and for every edge in the path coming into the node there is one leaving. Hence, because v has even degree, it follows that v is adjacent to at least one edge that is not in the path. Follow this edge, extending the path by one edge. This maintains the loop invariant while making progress. This process can get stuck only when the path happens to cycle back to the starting node, giving $s = v$. In such a case, join the path here to form a cycle.

Ending: If the cycle created covers all of the edges, then we are done.

Getting Unstuck: If the cycle we have created from our chosen node s back to s does not cover all the edges, then we look for a node u within this cycle that is adjacent to an edge not in the cycle. See [Figure 2.4.b](#). Change s to be this new node u . We break the cycle at u , giving us a path from u back to u . The difference with this path is that we can extend it past u along the unvisited edge. Again the loop invariant has been maintained while making progress.

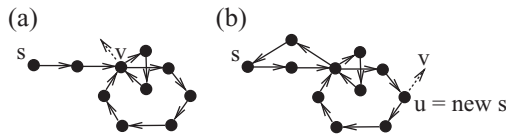


Fig. 2.4 Path constructed thus far by the Euler algorithm within the undirected graph.

u Exists: The only thing remaining to prove is that when v comes around to meet s again and we are not done, then there is in fact a node u in the path that is adjacent to an edge not in the path. Because we are not done, there is an edge e in the graph that is not in our path. Because the graph is connected, there must be a path in the graph

from e to our constructed path. The node u at which this connecting path meets our constructed path must be as required, because the last edge $\{u, w\}$ in the connecting path is not in our constructed path.

Extended Loop Invariant: To avoid having to find such a node u when it is needed, we extend the loop invariant to state that in addition to the path, the algorithm remembers some node u other than s and v that is in the path and is adjacent to an edge not in the path.

Exercise 2.3.1 (See solution in Exercise Solutions.) Iterative cake cutting: The famous algorithm for fairly cutting a cake in two is for one person to cut the cake in the place that they believe is half and for the other person to choose which “half” they like. One player may value the icing and while the other the cake more, but it does not matter. The second player is guaranteed to get a piece that they consider to be worth at least a half, because they choose between two pieces whose sum worth for them is at least one. Because the first person cut it in half according to their own criteria, they are happy which ever piece is left for them. Our goal is write an iterative algorithm that solves this same problem for n players.

To make our life easier, we view a cake not as a three-dimensional thing, but as the line from zero to one. Different players value different subintervals of the cake differently. To express this, each player assigns some numerical value to each subinterval. For example, if player p_i 's name is written on the subinterval $[\frac{i-1}{2n}, \frac{i}{2n}]$ of the cake, then they might allocate a higher value to it, say, $\frac{1}{2}$. The only requirement is that the total value of the cake is one.

Your algorithm is only allowed the following two operations. In an evaluation query, $v = \text{Eval}(p, [a, b])$, the algorithm asks a player p how much (v) they value a particular subinterval $[a, b]$ of the whole cake $[0, 1]$. In a cut query, $b = \text{Cut}(p, a, v)$, the protocol asks the player p to identify the shortest subinterval $[a, b]$, starting at a given left endpoint a , with a given value v . In the above example, $\text{Eval}(p_i, [\frac{i-1}{2n}, \frac{i}{2n}])$ returns $\frac{1}{2}$ and $\text{Cut}(p_i, \frac{i-1}{2n}, \frac{1}{2})$ returns $\frac{i}{2n}$. Using these, the two-player algorithm is as follows:

algorithm *Partition2*($\{p_1, p_2\}, [a, b]$)

<pre-cond>: p_1 and p_2 are players.

$[a, b] \subseteq [0, 1]$ is a subinterval of the whole cake.

<post-cond>: Returns a partitioning of $[a, b]$ into two disjoint pieces $[a_1, b_1]$ and $[a_2, b_2]$ so that player p_i values $[a_i, b_i]$ at least half as much as he values $[a, b]$.

begin

$v_1 = \text{Eval}(p_1, [a, b])$

$c = \text{Cut}(p_1, a, \frac{v_1}{2})$

if ($\text{Eval}(p_2, [a, c]) \leq \text{Eval}(p_2, [c, b])$) *then*

$[a_1, b_1] = [a, c]$ and $[a_2, b_2] = [c, b]$

else

$[a_1, b_1] = [c, b]$ and $[a_2, b_2] = [a, c]$

end if

return ($[a_1, b_1]$ and $[a_2, b_2]$)
end algorithm

The problem that you must solve is the following:

algorithm *Partition*(n, P)

(pre-cond): P is a set of n players.

Each player in P *values the whole cake* $[0, 1]$ *by at least* 1 .

(post-cond): *Returns a partitioning of* $[0, 1]$ *into* n *disjoint pieces* $[a_i, b_i]$ *so that for each* $i \in P$, *the player* p_i *values* $[a_i, b_i]$ *by at least* $\frac{1}{n}$.

begin

...

end algorithm

1. Can you cut off n pieces of cake, each of *size* strictly bigger than $\frac{1}{n}$, and have cake left over? Is it sometimes possible to have allocated a disjoint piece to each player, each worth much more than $\frac{1}{n}$, to the receiving player, and for there to still be cake left? Explain.
2. As a big hint to designing an iterative algorithm, I will tell you what the first iteration accomplishes. (Later iterations may do slightly modified things.) Each player specifies where they would cut if they were to cut off the first fraction $\frac{1}{n}$ of the $[a, b]$ cake. The player who wants the smaller amount of this first part of the cake is given this piece of the cake. The code for this is as follows:

loop $i \in P$
 $c_i = \text{Cut}(p_i, 0, \frac{1}{n})$
end loop
 $i_{\min} = \text{the } i \in P \text{ that minimizes } c_i$
 $[a_{i_{\min}}, b_{i_{\min}}] = [0, c_{i_{\min}}]$

As your first step in designing the algorithm, what is your loop invariant? It should include:

- (a) how the cake has been cut so far,
 - (b) who has been given cake, and how they feel about it, and
 - (c) how the remaining players feel about the remaining cake.
3. Give the iterative pseudocode.
 4. Formally prove that the loop invariant is established.
 5. Formally prove that the loop invariant is maintained.
 6. Formally prove that the postcondition is established.
 7. What is the running time of this algorithm?
 8. Is this a more-of-the-input or a more-of-the-output loop invariant?

3 Abstract Data Types

Abstract data types (ADTs) provide both a language for talking about and tools for operating on complex data structures. Each is defined by the types of objects that it can store and the operations that can be performed. Unlike a function that takes an input and produces an output, an ADT is more dynamic, periodically receiving information and commands to which it must react in a way that reflects its history. In an object-oriented language, these are implemented with objects, each of which has its own internal variables and operations. A user of an ADT has no access to its internal structure except through the operations provided. This is referred to as *information hiding* and provides a clean boundary between the user and the ADT. One person can use the ADT to develop other algorithms without being concerned with how it is implemented or worrying about accidentally messing up the data structure. Another can implement and modify the ADT without knowing how it is used or worrying about unexpected effects on the rest of the code. A general purpose ADT—not just the code, but also the understanding and the mathematical theory—can be reused in many applications. Having a limited set of operations guides the implementer to use techniques that are efficient for these operations yet may be slow for the operations excluded. Conversely, using an ADT such as a stack in your algorithm automatically tells someone attempting to understand your algorithm a great deal about the purpose of this data structure. Generally, the running time of an operation is not a part of the description of an ADT, but is tied to a particular implementation. However, it is useful for the user to know the relative expense of using operations so that they can make their own choices about which ADTs and which operations to use.

This chapter will treat the following ADTs: lists, stacks, queues, priority queues, graphs, trees, and sets. From the user's perspective, these consist of a data structure and a set of operations with which to access the data. From the perspective of the data structure itself, it is an ongoing system that continues to receive a stream of commands to which it must react dynamically. ADTs have a set of invariants or integrity constraints (both public and hidden) that must be true every time the system is entered or left. Imagining a big loop around the system allows us to regard them as a kind of loop invariant.

3.1 Specifications and Hints at Implementations

The following are examples of frequently used ADTs.

Simple Types: Integers, floating point numbers, strings, arrays, and records are abstract data types provided by all programming languages.

The List ADT

Specification: A list consists of an ordered sequence of elements. Unlike arrays, they contain no empty positions. Elements can be *inserted*, *deleted*, *read*, *modified*, and *searched for*.

Array Implementations: There are different implementations that have trade-offs in the running time, memory requirements, and difficulty of implementing. The obvious implementation of a list is to put the elements in an array. If the elements are packed one after the other, then the i th element can be accessed in $\Theta(1)$ time, but inserting or deleting an element requires $\Theta(n)$ time because all the elements need to be shifted. Alternatively, blank spaces could be left between the elements. This leaves room to insert or delete elements in $\Theta(1)$ time, but finding the i th element might now take $\Theta(n)$ time.

Linked List Implementations: A problem with the array implementation is that the array needs to be allocated some fixed size of memory when initialized. An alternative implementation, which can be expanded or shrunk in size as needed, uses a linked list. This implementation has the disadvantage of requiring $\Theta(n)$ time to access a particular element. See [Section 3.2](#).

Tree Implementations: A nice balance between the advantages of array and the linked list implementations is data structure called a *heap*. Heaps can do every operation in $\Theta(\log n)$ time. See [Section 3.4](#). Adelson-Velsky–Landis (AVL) trees and red–black trees have similar properties.

The Stack ADT

Specification: A stack ADT is the same as a list ADT, except its operations are limited. It is analogous to a stack of plates. A *push* is the operation of adding a new element to the top of the stack. A *pop* is the operation of removing the top element from the stack. The rest of the stack is hidden from view. This order is referred to as *last in, first out* (LIFO).

Use: Stacks are the key data structure for recursion and parsing. Having the operations limited means that all operations can be implemented easily and performed in constant time.

Array Implementation: The hidden invariants in an array implementation of a stack are that the elements in the stack are stored in an array starting with the bottom of the stack and that a variable *top* indexes the entry of the array containing the top element. It is not difficult to implement push and pop so that they maintain these invariants. The stack grows to the right as elements are pushed and shrinks to the left as elements are popped. For the code, see [Exercise 3.1.1](#).

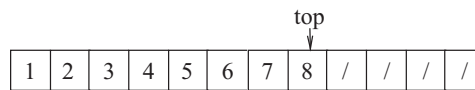


Fig. 3.1 Array.

Linked List Implementation: As with lists, stacks are often implemented using linked lists. See [Section 3.2](#).

The Queue ADT

Specification: The queue ADT is also the same as a list ADT, except with a different limited set of operations. A queue is analogous to a line-up for movie tickets. One is able to *insert* an element at the *rear* and *remove* the element that is at the *front*. This order is, *first in first out* (FIFO).

Queue Use: An operating system will have a queue of jobs to run and a *network hub* will have a queue of packets to transmit. Again, all operations can be implemented easily to run in constant time.

Array Implementation

Trying Small Steps: If the front element is always stored at index 1 of the array, then when the current front is removed, all the remaining elements would need to shift by one to take its place. To save time, once an element is placed in the array, we do not want to move it until it is removed. The effect is that the rear moves to the right as elements arrive, and the front moves to the right as elements are removed. We use two different variables, *front* and *rear*, to index their locations. As the queue migrates to the right, eventually it will reach the end of the array. To avoid getting stuck, we will treat the array as a circle, indexing modulo the size of the array. This allows the queue to migrate around and around as elements arrive and leave.

Hidden Invariants: The elements are stored in order from the entry indexed by *front* to that indexed by *rear* possibly wrapping around the end of the array.

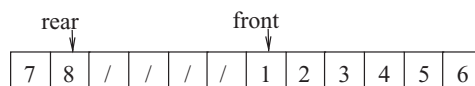


Fig. 3.2 Elements stored in order.

Extremes: It turns out that the cases of a completely empty and a completely full queue are indistinguishable, because with both *front* will be one to the left of *rear*. The easiest solution is not to let the queue get completely full.

Code: See [Exercises 3.1.2](#) and [3.1.3](#).

Linked List Implementation: Again, see [Section 3.2](#).

The Priority Queue ADT

Specification: A priority queue is still analogous to a line-up for movie tickets. However, in these queues the more important elements are allowed to move to the front of the line. When *inserting* an element, its priority must be specified. This priority can later be changed. When *removing*, the element with the highest priority in the queue is removed and returned. Ties are broken arbitrarily.

Tree Implementations: Heaps, AVL trees, and red–black trees can do each operation in $\Theta(\log n)$ time. See [Sections 4.1](#), [13.2](#), and [13.4](#).

The Set ADT

Specification: A *set* is basically a bag within which you can put any elements that you like. It is the same as a list, except that the elements cannot be repeated or ordered.

Indicator Vector Implementation: If the universe of possible elements is sufficiently small, then a good data structure is to have a Boolean array indexed with each of these possible elements. An entry being true will indicate that the corresponding element is in the set. All set operations can be done in constant time, i.e., in a time independent of the number of items in the set.

Hash Table Implementation: Surprisingly, even if the universe of possible elements is infinite, a similar trick can be done, using a data structure called a *hash table*. A pseudorandom function H is chosen that maps possible elements of the set to the entries $[1, N]$ in the table. It is a deterministic function in that it is easy to compute and always maps an element to the same entry. It is pseudorandom in that it appears to map each element into a random place. Hopefully, all the elements that are in your set happen to be placed into different entries in the table. In this case, one can determine whether or not an element is contained in the set, ask for an arbitrary element from the set, determine the number of elements in the set, iterate through all the elements, and add and delete elements—all in constant time, i.e., independently of the number of items in the set. If collisions occur, meaning that two of your set elements get mapped to the same entry, then there are a number of possible methods to rehash them somewhere else. Surprisingly insert, look up, and delete can each be done in constant time, i.e., $\theta(1)$ time.

The Set System ADT

Specification: A set system allows you to have a set (or list) of sets. Operations allow the *creation*, *union*, *intersection*, *complementation*, and *subtraction* of sets. The *find* operator determines which set a given element is contained in.

List-of-Indicator-Vectors or Hash-Table Implementations: One way to implement these is to have a list of elements implemented using an array or a

linked list where each of these elements is an implementation of a set. What remains is to implement operations that operate on multiple sets. Generally, these operations take $\Theta(n)$ time.

Union-Find Set System Implementation: Another quite surprising result is that on disjoint sets, the union and find operations can be done on average in a constant amount of time for all practical purposes. See the end of this section.

The Dictionary ADT: A dictionary associates a meaning with each word. Similarly, a dictionary ADT associates data with each *key*, generally as a list or as a Hash Table.

Graphs

Specification: A *graph* is set of nodes with edges between them. They can represent networks of roads between cities or friendships between people. The key information stored is which pairs of nodes are connected by an edge. Sometimes data, such as weight, cost, or length, can be associated with each edge or with each node. Though a drawing implicitly places each node at some location on the page, a key abstraction of a graph is that the location of a node is not specified. The basic operations are to determine whether an edge is in a graph, to add or delete an edge, and to iterate through the neighbors of a node. There is a huge literature of more complex operations that one might want to do. For example, one might want to determine which nodes have paths between them or to find the shortest path between two nodes. See [Chapter 20](#).

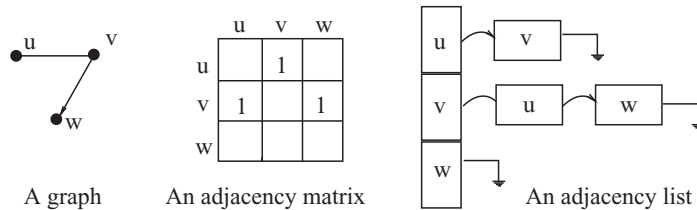


Fig. 3.3 Two graph data structures.

Adjacency Matrix Implementation: This consists of an $n \times n$ matrix with $M(u, v) = 1$ if $\langle u, v \rangle$ is an edge. It requires $\Theta(n^2)$ space (corresponding to the number potential edges) and $\Theta(1)$ time to access a given edge, but $\Theta(n)$ time to find the edges adjacent to a given node, and $\Theta(n^2)$ to iterate through all the nodes. This is only a problem when the graph is large and sparse.

Adjacency List Implementation: It lists for each node the nodes adjacent to it. It requires $\Theta(E)$ space (corresponding to the number of actual edges) and can iterate quickly through the edges adjacent to a give node, but requires time proportional to the degree of a node to access a specific edge.

See [Figure 3.3](#) for an example of a graph, an adjacency matrix, and an adjacency list.

Trees

Specification: Data are often organized into a hierarchy. A person has children, who have children of their own. The boss has people under them, who have people under them. The abstract data type for organizing this data is a *tree*.

Uses: There is a surprisingly large list of applications for trees. For two examples see Figure 3.4 and Section 13.5.

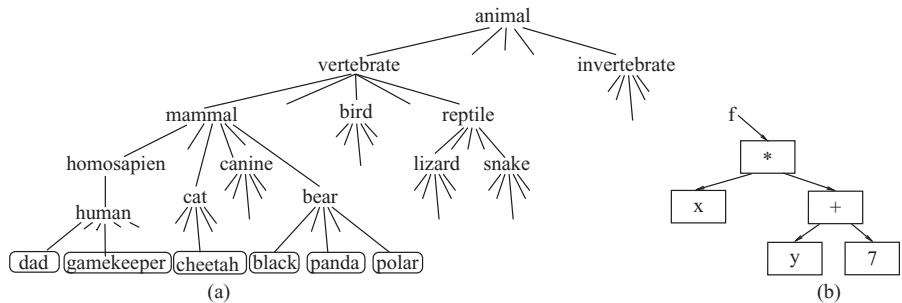


Fig. 3.4 Classification tree of animals and a tree representing the expression $f = x \times (y + 7)$.

Pointer Implementation: Trees are generally implemented by having each node point to each of its children:

Data Structure: Binary trees are usually stored with pointers between node objects. The user has immediate access to a variable *root* that points at the root node. Then each node object has fields *left* and *right* pointing to the node's left and right children. These fields will contain the null pointer if the node is a leaf. Each node also has a field *element* pointing to its data. Sometimes nodes also point back to their parent.

The pointer *root* is often renamed *tree* because it can be thought of in two different ways: on one hand, it can be thought of as the single node it is pointing to, and on the other hand, it can be thought of as a subtree rooted at the node pointed at. This latter is nice because *tree.left* can be thought both as the root node's left child node and as the left subtree of *tree*.

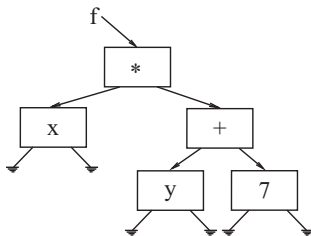


Fig. 3.5 A tree representing the expression $f = x \times (y + 7)$.

Object-oriented programming likes to package data and routines into objects. It will likely have the variable *Tree* be of type tree object that contains the pointer *root*. Using this notation, it is awkward to go from an object *tree* to the object that is that tree's left subtree. For this reason, most recursive programs have a preliminary routine that unwraps the *tree* container and recursively calls on the pointer to the root.

The pseudocode in this book uses *leftSub(tree)* to denote the left subtree of the tree *tree*.

Orders: Imposing rules on how the nodes can be ordered speeds up certain operations.

Binary Search Tree: A binary search tree is a data structure used to store keys along with associated data. The nodes are ordered so that for each node, all the keys in its left subtree are smaller than its key, and all those in the right subtree are larger. Elements can be found in such a tree, using binary search, in $O(\text{height})$ instead of $O(n)$ time. See [Sections 4.1](#) and [13.2](#).

Heaps: A heap requires that the key of each node be bigger than those of both its children. This allows one to find the maximum key in $O(1)$ time. All updates can be done in $O(\log n)$ time. Heaps are useful for a sorting algorithm known as heap sort and for the implementation of priority queues. See [Section 13.3](#).

Balanced Trees: If a binary tree is balanced, it takes less time to traverse down it, because it has height at most $\log_2 n$. It is too much work to maintain a perfectly balanced tree as nodes are added and deleted. There are, however, a number of data structures that are able to add and delete in $O(\log_2 n)$ time while ensuring that the tree remains almost balanced. Here are two.

AVL Trees: Every node has a *balance factor* of -1 , 0 , or 1 , defined as the difference between the heights of its left and right subtrees. As nodes are added or deleted, this invariant is maintained using rotations like [Figure 3.6](#) (see [Exercise 3.1.5](#)).

Red–Black Trees: Every node is either red or black. If a node is red, then both its children are black. Every path from the root to a leaf contains the same number of black nodes. See [Exercise 3.1.6](#).

Balanced Binary Search Tree: By storing the elements in a balanced binary search tree, insertions, deletions, and searches can be done in $\Theta(\log n)$ time.

Union–Find Set System: This data structure maintains a number of disjoint sets of elements.

Operations: (1) *Makeset*(*v*), which creates a new set containing the specified element *v*; (2) *Find*(*v*), which determines the *name* of the set containing a specified element (each set is given a distinct but arbitrary name); and (3) *Union*(*u*, *v*), which merges the sets containing the specified elements *u* and *v*.

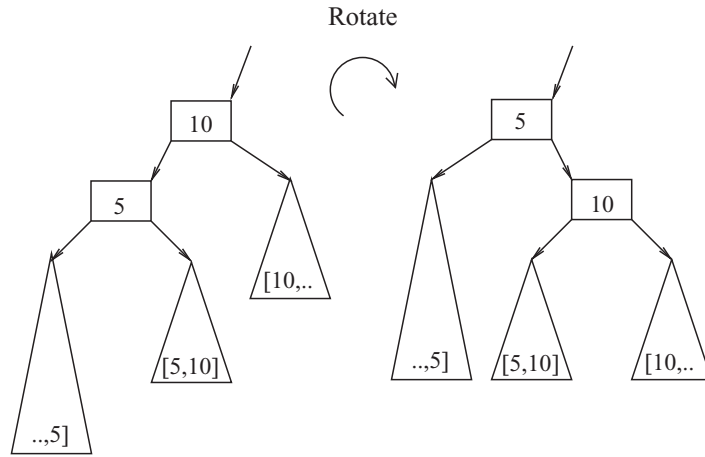


Fig. 3.6 Rotating a tree changing heights while maintain Binary Search Tree properties.

Use: One application of this is in the minimum-spanning-tree algorithm in [Section 22.2.3](#).

Running Time: On average, for all practical purposes, each of these operations can be completed in a constant amount of time. More formally, the total time to do m of these operations on n elements is $\Theta(m\alpha(n))$, where α is the *inverse Ackermann's function*. This function is so slow growing that even if n equals the number of atoms in the universe, then $\alpha(n) \leq 4$. See [Section 12.3](#).

Implementation: The data structure used is a rooted tree for each set, containing a node for each element in the set. The difference is that each node points to its parent instead of to its children. The name of the set is the contents of the root node. *Find*(w) is accomplished by tracing up the tree from w to the root u . *Union*(u, v) is accomplished by having node u point to node v . From then on, *Find*(w) for a node w in u 's tree will trace up and find v instead. What makes this fast on average is that whenever a *Find* operation is done, all nodes that are encountered during the find are changed to point directly to the root of the tree, collapsing the tree into a shorter tree.

Exercise 3.1.1 Implement the push and pop operations on a stack using an array as described in [Section 3.1](#).

Exercise 3.1.2 Implement the insert and remove operations on a queue using an array as described in [Section 3.1](#).

Exercise 3.1.3 When working with arrays, as in [Section 3.1](#), what is the difference between “ $rear = (rear + 1) \bmod \text{MAX}$ ” and “ $rear = (rear \bmod \text{MAX}) + 1$,” and when should each be used?

Exercise 3.1.4 For each of the three pairs of graphs in [Figure 3.8](#), number the nodes in such the way that $\langle i, j \rangle$ is an edge in one if and only if it is an edge in the other.

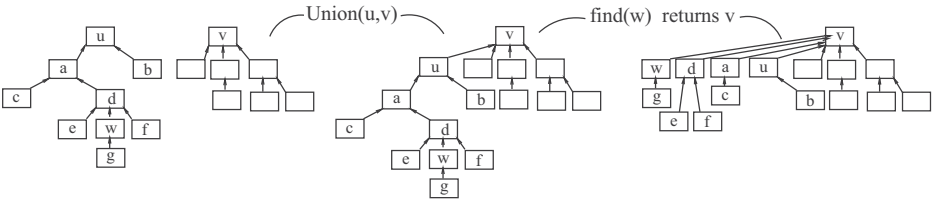


Fig. 3.7 The union-find data structure described above maintains a number of disjoint sets of elements.

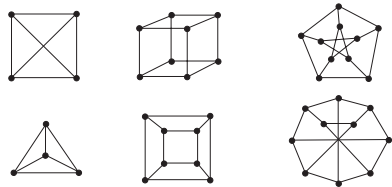


Fig. 3.8 The top row shows three famous graphs: the complete graph on four nodes, the cube, and the Peterson graph. The bottom row shows the same three graphs with their nodes laid out differently.

Exercise 3.1.5 (See solution in Exercise Solutions.) Prove that the height of an AVL tree with n nodes is $\Theta(\log n)$.

Exercise 3.1.6 Prove that the height of a red–black tree with n nodes is $\Theta(\log n)$.

3.2 Link List Implementation

As said, a problem with the array implementation of the list ADT is that the array needs to be allocated some fixed size of memory when it is initialized. A solution to this is to implement these operations using a linked list, which can be expanded in size as needed. This implementation is particularly efficient when the operations are restricted to those of a stack or a queue.

List ADT Specification: A list consists of an ordered sequence of elements. Unlike arrays, it has no empty positions. Elements can be *inserted*, *deleted*, *read*, *modified*, and *searched for*. There are trade-offs in the running time. Arrays can access the i th element in $\Theta(1)$ time, but require $\Theta(n)$ time to insert an element. A *linked list* is an alternative implementation in which the memory allocated can grow and shrink dynamically with the needs of the program. Linked lists allow insertions in $\Theta(1)$ time, but require $\Theta(n)$ time to access the i th element. Heaps can do both in $\Theta(\log n)$ time.

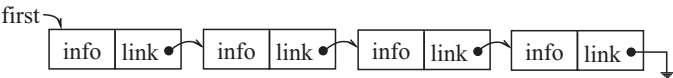


Fig. 3.9 In a linked list, each node points to the next.

Hidden Invariants: In a linked list, each node contains the information for one element and a pointer to the next. The variable *first* points to the first node, and *last* to the *last*. The last node has its pointer variable contain the value *nil*. When the list contains no nodes, *first* and *last* also point to *nil*.

Notation: A pointer, such as *first*, is a variable that is used to store the address of a block of memory. The information stored in the *info* field of such a block is denoted by *first.info* in Java and *first->info* in C. We will adopt the first notation. Similarly, *first.link* denotes the pointer field of the node. Being a pointer itself, *first.link.info* denotes the information stored in the second node of the linked list, and *first.link.link.info* in the third.

Adding a Node to the Front: Given a list ADT and new *Info* to store in an element, this operation is to insert an element with this information into the front the list.

General Case: We need the following steps (with pseudocode given to the right) for a large and general linked list. See Figure 3.10.

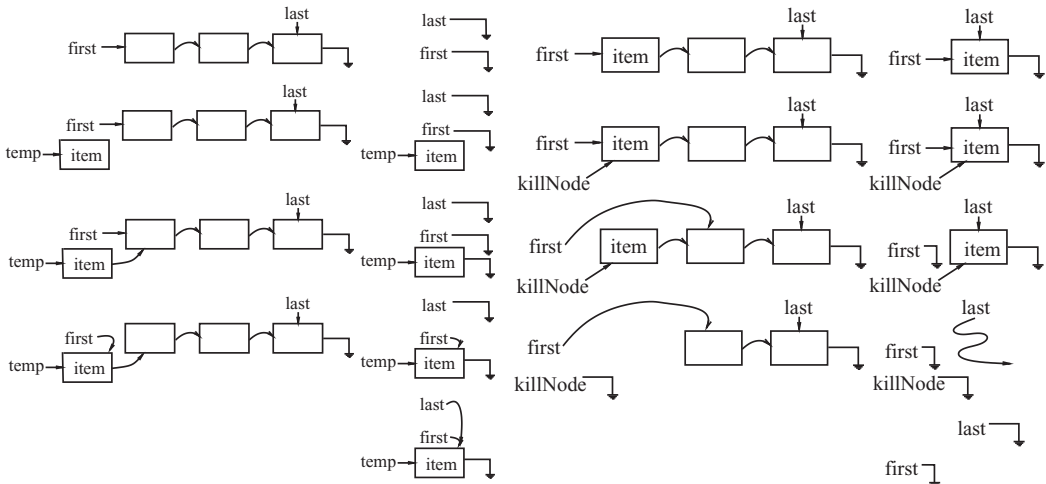


Fig. 3.10 Adding and removing a node from the front of a linked list.

- Allocate space for the new node.
- Store the information for the new element.
- Point the new node at the rest of the list.
- Point *first* at the new node.

New temp
temp.info = *Info*
temp.link = *first*
first = *temp*

Special Case: The main special case is an empty list. Sometimes we are lucky and the code written for the general case also works for such special cases. Inserting a node starting with both *first* and *last* pointing to *nil*, everything works except for *last*. Add the following to the bottom of the code.

- Whenever adding code to handle a special case, be sure to check that the previously handled cases still are handled correctly.

General Case

- Special Cases:** If the list is already empty, a node cannot be removed. The only other special case occurs when there is one node pointed to by both *first* and *last*. At the end of the code, *first* points to nil, which is correct for an empty list. However, *last* still points to the node that has been deleted. This can be solved by adding the following to the bottom of the code:

- Note that the value of *first* and *last* change. If the routine *Pop* passes these parameters in by value, the routine needs to be written to allow this to happen.

Adding Node to End: See [Exercise 3.2.2](#).

Walking Down the Linked List: Now suppose that the elements in the linked lists are sorted by the field *info*. When given an *info* value *newElement*, our task is to point the pointer *next* at the first element in the list with that value. The pointer *prev* is to point to the previous element in the list. This needs to be saved, because if it is needed, there is no back pointer to back up to it. If such an element

does not exist, then *prev* and *next* are to sandwich the location where this element would go. For example, if *newElement* had either the value 6 or the value 8, the result of the search would be as shown in Figure 3.9.

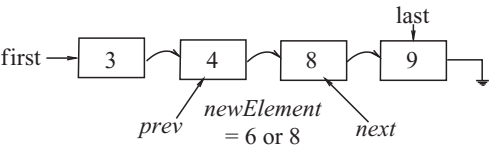


Fig. 3.11 Linked list.

- Walk down the list
 - maintaining the two pointers
 - until the desired location is found
 - pointing *prev* where *next* is pointing
 - and pointing *next* to the next node.
- loop

(loop-inv): *prev* and *next* point to consecutive nodes before or at our desired location.

exit when *next* = *nil*
or *next.info* ≥ *newElement*

prev = *next*
next = *next.link*

end loop

Running Time: This can require $O(n)$ time, where n is the length of the list.

Initialize the Walk: To initially establish the loop invariant, *prev* and *next* must sandwich the location before the first node. We do this as follows (Figure 3.12):

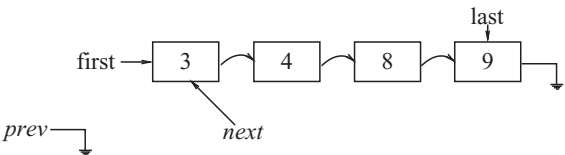


Fig. 3.12 Initialize the walk.

- Sandwich the location before the first node.
- prev* = *nil*
next = *first*

Adding a Node

Into the Middle: The general case to consider first is adding the node into the middle of the list (Figure 3.13).

- Allocate space for the new node.
 - Store the information for the new element.
 - Point the previous node to the new node.
 - Point the new node to the next node.
- new temp*
temp.info = *item*
prev.link = *temp*
temp.link = *next*

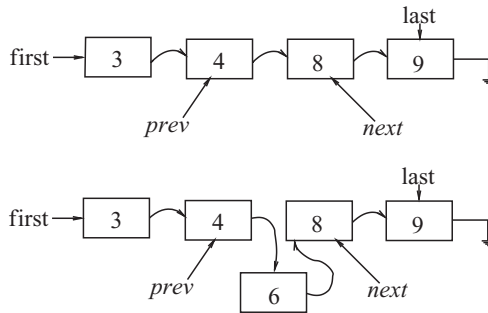


Fig. 3.13 Inserting a new element into a linked list.

At the Beginning: If the new node belongs at the beginning of the list (say, value 2), then $prev.link = temp$ would not work, because $prev$ is not pointing at a node. We will replace this line with the following (see Figure 3.14):

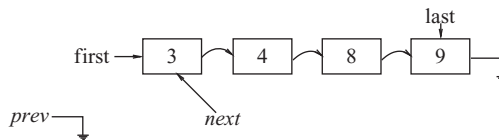


Fig. 3.14 Inserting a new element at the beginning of a linked list.

- If the new node is to be the first node, if $prev = nil$ then
- point $first$ at the new node $first = temp$
- else else
- point the previous node to the new node. $prev.link = temp$
- end if

At the End: Now what if the new node is to be added on the end (e.g. value 12)? The variable $last$ will no longer point at the last node. Adding the following code to the bottom will solve the problem (see Figure 3.15):

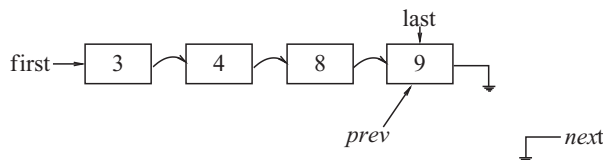


Fig. 3.15 Inserting a new element at the end of a linked list.

- If the new node will be the last node, if $prev = last$ then
- point $last$ at the new node. $last = temp$
- end if

To an Empty List: Another case to consider is when the initial list is empty. In this case, all the variables, $first$, $last$, $prev$, and $next$, will be nil. The new code works in this case as is.

Complete Code for Adding a Node: One needs to put all of these pieces together into one *insert* routine. See [Exercise 3.2.5](#).

Deleting a Node

From the Middle: Again the general case to consider first is deleting the node from the middle of the list. We must maintain the linked list *before* destroying the node. Otherwise, we will drop the list ([Figure 3.16](#)).

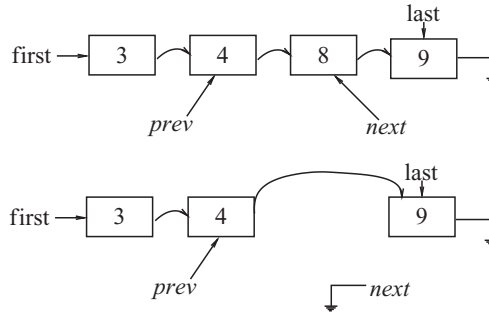


Fig. 3.16 Deleting an element from a linked list.

- Bypass the node being deleted. $prev.link = next.link$
- Deallocate the memory pointed to by $next$. $free\ next$

From the Beginning or the End: As before, you need to consider all the potential special cases. See [Exercise 3.2.6](#) ([Figure 3.17](#)).

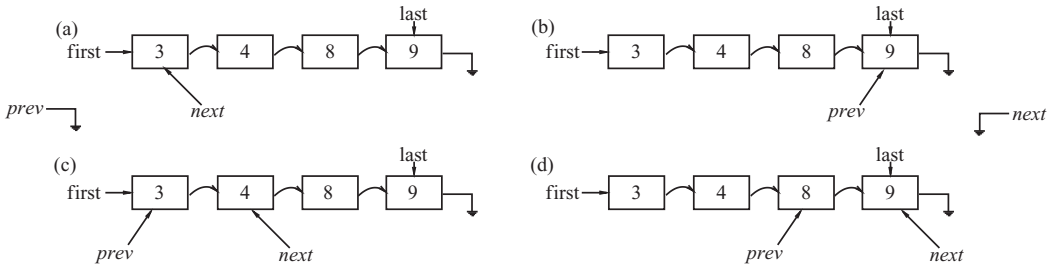


Fig. 3.17 Special cases for deleting an element from a linked list. See [Exercise 3.2.6](#).

Exercise 3.2.1 Implement testing whether a linked list is empty.

Exercise 3.2.2 Implement adding a node to the end of a linked list.

Exercise 3.2.3 Double pointers: Describe how this operation can be done in $\Theta(1)$ time if there are pointers in each node to both the previous and the next node.

Exercise 3.2.4 (See solution in Exercise Solutions.) In the code for walking down the linked list, what effect, if any, would it have if the order of the exit conditions were switched to “exit when $next.info \geq newElement$ or $next = nil$ ”?

Exercise 3.2.5 Implement the complete code `insert` that, when given an info value `newElement`, inserts a new element where it belongs into a sorted linked list. This involves only putting together the pieces just provided.

Exercise 3.2.6 Implement the complete code `Delete` that, when given an info value `newElement`, finds and deletes the first element with this value, if it exists. This involves also considering the four special cases listed for deleting a node from the beginning or the end of a linked list.

3.3 Merging with a Queue

Merging consists of combining two sorted lists, A and B , into one completely sorted list, C . Here A , B , and C are each implemented as queues. The loop invariant maintained is that the k smallest of the elements are sorted in C . (This is a classic more-of-the-output loop invariant. It is identical to that for selection sort.) The larger elements are still in their original lists A and B . The next smallest element will be either the first element in A or the first element in B . Progress is made by removing the smaller of these two first elements and adding it to the back of C . In this way, the algorithm proceeds like two lanes of traffic merging into one. At each iteration, the first car from one of the incoming lanes is chosen to move into the merged lane. This increases k by one. Initially, with $k = 0$, we simply have the given two lists. We stop when $k = n$. At this point, all the elements will be sorted in C . Merging is a key step in the merge sort algorithm presented in [Section 12.1](#).

algorithm *Merge*(*list* : A, B)

⟨*pre-cond*⟩: A and B are two sorted lists.

⟨*post-cond*⟩: C is the sorted list containing the elements of the other two.

begin

 loop

 ⟨*loop-invariant*⟩: The k smallest of the elements are sorted in C .

 The larger elements are still in their original lists A and B .

 exit when A and B are both empty

 if (the first in A is smaller than the first in B or B is empty) then

next Element = Remove first from A

 else

next Element = Remove first from B

 end if

 Add *next Element* to C

 end loop

 return (C)

end algorithm

3.4 Parsing with a Stack

One important use of stack is for parsing.

Specifications

Preconditions: An input instance consists of a string of brackets.

Postconditions: The output indicates whether the brackets match. Moreover, each left bracket is allocated an integer 1, 2, 3, ..., and each right bracket is allocated the integer from its matching left bracket.

Example

Input: ([{ } ()] () { () })
 Output: 1 2 3 3 4 4 2 5 5 6 7 7 6 1

The Loop Invariant: Some prefix of the input instance has been read, and the correct integer allocated to each of these brackets. (Thus, it is a more-of-the-input loop invariant.) The left brackets that have been read and not matched are stored along with their integers in left-to-right order in a stack, with the rightmost on top. The variable c indicates the next integer to be allocated to a left bracket.

Maintaining the Loop Invariant: If the next bracket read is a left bracket, then it is allocated the integer c . Not being matched, it is pushed onto the stack. c is incremented. If the next bracket read is a right bracket, then it must match the rightmost left bracket that has been read in. This will be on the top of the stack. The top bracket on the stack is popped. If it matches the right bracket, i.e., we have $()$, $\{\}$, or $[\]$, then the right bracket is allocated the integer for this left bracket. If not, then an error message is printed.

Initial Conditions: Initially, nothing has been read and the stack is empty.

Ending: If the stack is empty after the last bracket has been read, then the string has been parsed.

Code:

algorithm *Parsing(s)*

<pre-cond> s is a string of brackets.

<post-cond> Prints out a string of integers that indicate how the brackets match.

begin

$i = 0, c = 1$

loop

<loop-invariant> Prefix $s[1, i]$ has been allocated integers, and its left brackets are on the stack.

exit when $i = n$

if($s[i + 1]$ is a left bracket) then

print(c)

```
    push( $\langle s[i + 1], c \rangle$ )
     $c = c + 1$ 
elseif( $s[i + 1] = \text{right bracket}$ ) then
    if(stackempty()) return ("Cannot parse")
     $\langle left, d \rangle = \text{pop}()$ 
    if( $left$  matches  $s[i + 1]$ ) then print ( $d$ )
    else return("Cannot parse")
else
    return("Invalid input character")
end if
 $i = i + 1$ 
end loop
if(stackempty()) return("Parsed") else return("Cannot parse")
end algorithm
```

Parsing only “()”: If you only need to parse one type of brackets and you only want to know whether or not the brackets match, then you do not need the stack in the above algorithm, only an integer storing the number of left brackets in the stack.

Parsing with Context-Free Grammars: To parse more complex sentences, see [Chapter 15](#) and [Chapter 25](#).

4 Narrowing the Search Space: Binary Search

In this chapter, we will consider more binary search algorithms, which use the narrowing-the-search-space type of loop invariant. In this case, if the thing being searched for is anywhere, then it is in the narrowed sublist. We first look at general binary search trees, which are often used in recursive algorithms (see [Section 13.2](#)) and then look at another example of an algorithm that incorporates binary search.

4.1 Binary Search Trees

[Section 3.1](#) defines a *binary search tree* to be a binary tree data structure in which each node stores an element (and some associated data). The nodes are ordered so that for each node all the elements in its left subtree are smaller than that node's element and all those in its right subtree are larger. I will show here how to search quickly for an element with a given tree.

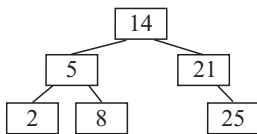


Fig. 4.1 Binary search tree.

1) Specifications: Given a binary search tree and a key, find a node whose element is this key or report that there is no such node.

2) Basic Steps: The restricted search space will be a subtree. Just as in binary search, our goal is to cut the size of this search space in half.

3) Measure of Progress: For binary search [Example 1.4.3](#), the measure of progress was the number of elements in the current sublist. With binary search trees this number is not as predictable when the tree is not balanced. Hence, the measure of progress will be the number of edges in the path from the root of the entire subtree to the root of the current subtree.

4) The Loop Invariant: The loop invariant states that if the key is contained somewhere in the entire binary search tree, then it is contained in our current subtree.

5) Main Steps: In binary search, we compared the key with the element in the middle of the current search space. With an unbalanced binary search tree, we do not know which node is at the exact middle. Instead, we compare the key with the element at the root of the current subtree. If it contains the key, then we are done. If the key is smaller than it, then we know that if the key is anywhere, it is in the left subtree of our current tree, else we know that it is in the right subtree.

6) Make Progress: As the root of our current subtree moves down to either its left or its right subtree, the measure of progress increases by one.

7) Maintain Loop Invariant: $\langle \text{loop-invariant}' \rangle \ \& \ \text{not} \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. The previous loop invariant gives that the search has been narrowed down to the current subtree. The property of binary search trees is that all the elements within this subtree that are smaller than the root of this subtree are in this subtree's left subtree, and all those larger in its right. Hence, the main steps narrow the search space to the subtree that would contain the key.

8) Establishing the Loop Invariant: $\langle \text{pre-cond} \rangle \ \& \ \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$. Initially, you obtain the loop invariant by considering the entire tree of elements as the current subtree.

9) Exit Condition: We exit either when the key is found or when the current subtree becomes empty.

10) Ending: $\langle \text{loop-invariant} \rangle \ \& \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$. By the exit condition, either we have found the key, in which case we are done, or we have narrowed the search space to an empty subtree. The loop invariant says that if the key is contained in the original list, then the key is contained in this empty subtree, which it is not, and hence the algorithm can safely report that the key is not in the original list.

11) Termination and Running Time: The number of iterations of this algorithm is at most the height of the binary search tree, which (if the tree more or less balanced) is $\Theta(\log n)$.

12) Special Cases: There are no special cases here: any input is either equal to, larger than, or smaller than the root of the current subtree.

13) Coding and Implementation Details

algorithm *SearchBST*(*tree*, *keyToFind*)

$\langle \text{pre-cond} \rangle$: *tree* is a binary tree whose nodes contain key and data fields.
keyToFind is a key.

$\langle \text{post-cond} \rangle$: If there is a node with this key in the tree, then the associated data is returned.


```

begin
  subtree = tree
  loop
    (loop-invariant): If the key is contained in tree, then the key is
                      contained in subtree.
    if(subtree = emptyTree) then
      result("key not in tree")
    else if( keyToFind < rootKey(subtree)) then
      subtree = leftSub(subtree)
    else if( keyToFind = rootKey(subtree)) then
      result(rootData(subtree))
    else if(keyToFind > rootKey(subtree)) then
      subtree = rightSub(subtree)
    end if
  end loop
end algorithm

```

4.2 Magic Sevens

My mom gave my son Joshua a book of magic tricks. The book says, “This trick really is magic. It comes right every time you do it, but there is no explanation why.” As it turns out, there is a bug in the way that they explain the trick. Our task is to fix the bug and to counter “there is no explanation why.” The only magic is that of loop invariants. The algorithm is a variant on binary search.

1) Specifications

- Let c , an odd integer, be the number of *columns*. The book uses $c = 3$.
- Let r , an odd integer, be the number of *rows*. The book uses $r = 7$.
- Let $n = c \cdot r$ be the number of cards. The book uses $n = 21$.
- Let t be the number of iterations. The book uses $t = 2$.
- Let f be the final index of the selected card. The book uses $f = 11$.
- Ask someone to select one of the n cards and then shuffle the deck.
- Repeat t times:
 - Spread the cards out as follows. Put c cards in a row left to right face up. Put a second row on top, but shifted down slightly so that you can see what both the first and second row of cards are. Repeat for r rows. This forms c columns of r cards each.
 - Ask which column the selected card is in.
 - Stack the cards in each column. Put the selected column in the middle. (This is why c is odd.)
- Output the f th card.

Our task is to determine for which values c, r, n, t , and f this trick finds the selected card.

Easier Version: Analyzing this trick turns out to be harder than I initially thought. Hence, we'll consider the following easier trick first. Instead of putting the selected column in the middle of the other columns, we put it in front.

2) Basic Steps: At each iteration we gain some information about which card had been selected. The trick seems to be similar to binary search. A difference is that binary search splits the current sublist into two parts, whereas this trick splits the entire pile into c parts. In both algorithms, at each iteration we learn which of these piles the sought-after element is in.

4) Loop Invariant: A good first guess for a loop invariant would be that used by binary search. The loop invariant will state that some subset S_i of the cards contains the selected card. In this easier version, the column containing the card is continually moved to the front of the stack. Hence, let us guess that $S_i = \{1, 2, \dots, s_i\}$ indexes the first s_i cards in the deck. We will later solve a recurrence relation to determine that $s_i = \lceil n/c^i \rceil$.

7) Maintain Loop Invariant: $\langle \text{loop-invariant}' \rangle \ \& \ \text{not} \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. By the previous loop invariant, the selected card is one of the first s_{i-1} in the deck. When the cards are laid out, the first s_{i-1} cards will be spread on the tops of the c columns. Some columns will get $\lceil s_{i-1}/c \rceil$ of these cards, and some will get $\lfloor s_{i-1}/c \rfloor$ of them. When we are told which column the selected card is in, we will know that the selected card is one of the first $\lceil s_{i-1}/c \rceil$ cards in this column. In conclusion,

$$s_i = \left\lceil \frac{s_{i-1}}{c} \right\rceil = \left\lceil \frac{\left\lceil \frac{n}{c^{i-1}} \right\rceil}{c} \right\rceil = \left\lceil \frac{n}{c^i} \right\rceil.$$

8) Establishing the Loop Invariant: Again, as done in binary search, we initially obtain the loop invariant by considering the entire stack of cards, giving $s_0 = \lceil n/c^0 \rceil = n$.

9) Exit Condition: When sufficient rounds have occurred so that $s_t = 1$, the search space has narrowed down to containing only the first card. Hence, the algorithm is able to select card $f = 1$.

11) Running Time: After $t = \lceil \log_c n \rceil$ rounds, $s_t = \lceil n/c^t \rceil = 1$.

For a matching lower bound on the number of iterations needed, see [Chapter 7](#).

The book has $n = 21$, $c = 3$, and $t = 2$. Because $21 = n \not\leq c^t = 3^2 = 9$, the trick in the book does not work. Two rounds is not enough. There must be three.

Original Trick: Consider again the original trick where the selected column is put into the middle.

4) The Loop Invariant: Because the selected column is put into the middle, let us guess that S_i consists of the middle s_i cards. More formally, let $d_i = (n - s_i)/2$. Neither the first nor the last d_i cards will be the selected card. Instead, it will be one of $S_i = \{d_i + 1, \dots, d_i + s_i\}$. Note that both n and s_i need to be odd.

8) Establishing the Loop Invariant: For $i = 0$, we have $s_0 = n$, $d_0 = 0$, and the selected card can be any card in the deck.

7) Maintain Loop Invariant: Suppose that before the i th iteration, the selected card is not one of the first d_{i-1} cards, but is one of the middle s_{i-1} in the deck. Then when the cards are laid out, the first d_{i-1} cards will be spread on the tops of the c columns. Some columns will get $\lceil d_{i-1}/c \rceil$ of these cards, and some will get $\lfloor d_{i-1}/c \rfloor$ of them. In general, however, we can say that the first $\lfloor d_{i-1}/c \rfloor$ cards of each column are not the selected card. We use the floor instead of the ceiling here, because this is the worst case. By symmetry, we also know that the selected card is not one of the last $\lfloor d_{i-1}/c \rfloor$ cards in each column. When the person points at a column, we learn that the selected card is somewhere in that column. However, from before we knew that the selected card is not one of the first or last $\lfloor d_{i-1}/c \rfloor$ cards in this column. There are only r cards in the column. Hence, the selected card must be one of the middle $r - 2\lfloor d_{i-1}/c \rfloor$ cards in the column. Define s_i to be this value. The new deck is formed by stacking the columns together with these cards in the middle.

9) Exit Condition: When sufficient rounds have occurred so that $s_i = 1$, then the selected card will be in the middle indexed by $f = \lceil \frac{n}{2} \rceil$.

Trick in Book: The book has $n = 21$, $c = 3$, and $r = 7$. Thus

$$\begin{array}{lll}
 s_i = r - 2 \lfloor \frac{d_{i-1}}{c} \rfloor & d_i = \frac{n - s_i}{2} & S_i = \{d_i + 1, \dots, d_i + s_i\} \\
 s_0 = n = 21 & d_0 = \frac{21 - 21}{2} = 0 & S_0 = \{1, 2, \dots, 21\} \\
 s_1 = 7 - 2 \lfloor \frac{0}{3} \rfloor = 7 & d_1 = \frac{21 - 7}{2} = 7 & S_1 = \{8, 9, \dots, 14\} \\
 s_2 = 7 - 2 \lfloor \frac{7}{3} \rfloor = 3 & d_2 = \frac{21 - 3}{2} = 9 & S_2 = \{10, 11, 12\} \\
 s_3 = 7 - 2 \lfloor \frac{9}{3} \rfloor = 1 & d_3 = \frac{21 - 1}{2} = 10 & S_3 = 11.
 \end{array}$$

Again, three and not two rounds are needed.

11) Running Time: Temporarily ignoring the floor in the equation for s_i makes the analysis easier. We have

$$s_i = r - 2 \left\lfloor \frac{d_{i-1}}{c} \right\rfloor \approx r - 2 \frac{d_{i-1}}{c} = \frac{n}{c} - 2 \frac{(n - s_{i-1})/2}{c} = \frac{s_{i-1}}{c}.$$

Again, this recurrence relation gives that $s_i = n/c^i$. If we include the floor, challenging manipulations give that $s_i = 2 \lfloor s_{i-1}/2c - \frac{1}{2} \rfloor + 1$. More calculations give that s_i is always n/c^i rounded up to the next odd integer.

Exercise 4.2.1 Give code for the original Magic Sevens trick.

Exercise 4.2.2 Suppose S and T are sorted arrays, each containing elements. Find the n th smallest of the $2n$ numbers.

4.3 VLSI Chip Testing

The following is a strange problem with strange rules. However, it is no stranger than the problems that you will need to solve in the world. We will use this as an example of

how to develop a strange loop invariant with which the algorithm and its correctness become transparent.

Specification: Our boss has n supposedly identical VLSI chips that are potentially capable of testing each other. His test jig accommodates two chips at a time. The result is either that they are the same (that is, both are good or both are bad), or that they are different (that is, at least one is bad). The professor hires us to design an algorithm to distinguish good chips from bad ones.

Impossible? Some computational problems have exponential-time algorithms, but no polynomial-time algorithms. Because we are limited in what we are able to do, this problem may not have an algorithm at all. It is often hard to know. A good thing to do with a new problem is to alternate between author and critic. The author does their best to design an algorithm for the problem. The critic does their best to prove that the author's algorithm does not work or, even better, prove that no algorithm works.

Suppose that the professor happened to have one good chip and one bad chip. The single test that they have tells them that these chips are different, but does not tell them which is which. There is no algorithm using only this single test that accomplishes the task. The professor may not be happy with our findings, but they will not be able to blame us.

Though we have shown that there is no algorithm that distinguishes these two chips, perhaps we can find an algorithm that can be of some use for the professor.

A Data Structure: It is useful to have a good data structure with which to store the information that has been collected. Here we can have a graph with a node for each chip. After testing a pair of chips, we put a solid edge between the corresponding nodes if they are reportedly the same and a dotted edge if they are different.

The Brute Force Algorithm: One way of understanding a problem better is to initially pretend that you have unbounded time and energy. With this, what tasks can you accomplish? With $\Theta(n^2)$ tests we can test every pair of chips. We assume that the test is transitive, meaning that if chip a tests to be the same as b , which tests to be the same as c , then a will test to be the same as c . Given this, we can conclude that the tests will partition the chips into sets of chips that are the same. (In graph theory, we call these sets *cliques*, as in a clique of friends in which everyone in the group is friends with everyone else in the group.) There is, however, no test available to determine which of these sets contain the good chips.

Change the Problem: When you get stuck, a useful thing to do is to go back to your boss or to the application at hand and see if you can change the problem to make it easier. There are three ways of doing this.

More Tools: One option is to allow the algorithm more powerful tools. A test that told you whether a chip was good would solve the problem. On the other hand, if the professor had such a test, you would be out of a job.

Change the Preconditions: You can change the preconditions to require additional information about the input instance or to disallow particularly difficult

instances. You need some way of distinguishing between the good chips and the various forms of bad chips. Perhaps you can get the professor to assure you that more than half of the chips are good. With this, you can solve the problem. Test all pairs of chips and partition the chips into the sets of equivalent chips. The largest of these sets will be the good chips.

Change the Postconditions: Another option is to change the postconditions by not requiring so much in the output. Instead of needing to distinguish completely between good and bad chips, an easier task would be to find a single good chip.

A Faster Algorithm: Once we have our brute force algorithm, we will want to attempt to find a faster algorithm. Let us attempt to find a single good chip from among n chips, assuming that more than $n/2$ of the chips are good, using an iterative algorithm. Hopefully, it will be faster than $\Theta(n^2)$ time.

Designing the Loop Invariant: In designing an iterative algorithm for this problem, the most creative step is designing the loop invariant.

Start with Small Steps: What basic steps might we follow to make some kind of progress? Certainly the first step is to test two chips. There are two cases.

Different: Suppose that we determine that the two chips are different. One way to make progress is to narrow down the input instance while maintaining what we know about it. What we know is that more than half of the chips are good. Because we know that at least one of the two tested chips is bad, we can throw both of them away. We know that we do not go wrong by doing this, because we maintain the loop invariant that more than half of the chips are good. From this we know that there is still at least one good chip remaining, which we can return as the answer.

Same: If the two chips test the same, we cannot throw them away, because they might both be good. However, this too seems like we are making progress, because, as in the brute force algorithm, we are building up a set of chips that are the same.

Picture from the Middle: From our single step, we saw two forms of progress. First, we saw that some chips will have been set aside. Let S denote the subset containing all the chips that we have not set aside. Second, we saw that we were building up sets of chips that we know to be the same. It may turn out that we will need to maintain a number of these sets. To begin, however, let's start with the simplest picture and build only one such set.

The Loop Invariant: We maintain two sets. The set S contains the chips that we have not set aside. We maintain that more than half of the chips in S are good. The set C is a subset of S . We maintain that all of the chips in C are the same, though we do not know whether they are all good or all bad.

Type of Loop Invariant: This is a strange loop invariant, but it has a number of familiar aspects.

More of the Input: We do consider the chips one at a time in order.

More of the Output: The set C is our first guess at what the outputted good chips will be (but this may change).

Narrow the Search Space: The narrowed set S contains at least one good chip.

Case Analysis: We check different cases as to whether the chips are the same or different.

Work Done: The sets S and C keep track of the work done so that it need not be redone.

Maintaining the Loop Invariant: $\langle \text{loop-invariant}' \rangle \ \& \ \text{not}(\text{exit-cond}) \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. Assume that all we know is that the loop invariant is true. It being the only thing that we know how to do, we must test two chips. Testing two from C is not useful, because we already know that they are the same. Testing two that are not in C is dangerous, because if we learn that they are same, then we will have to start a second set of alike chips, yet we previously decided to maintain only one. The remaining possibility is to choose any chip from C and any from $S-C$ and test them. Let us denote these chips by c and s .

Same: If the conclusion is that the chips are the same, then add chip s to C . We have not changed S , so its loop invariant still holds. From our test, we know that s has the same characteristic as c . From the loop invariant, we know that c is the same as all the other chips in C . Hence, we know that s is the same as all the other chips in C , and the loop invariant follows.

Different: If the conclusion is that at least one is bad, then delete both c and s from C and S . Now S has lost two chips, at least one of which is bad. Hence, we have maintained the fact that more than half of the chips in S are good. Also, C has only become smaller, and hence we have maintained the fact that its chips are all the same.

Either way, we maintain the loop invariant while making some (yet undefined) progress.

Handle All Cases: We can only test one chip from C and one from $S-C$ if both are nonempty. We need to consider the cases in which they are not.

S Is Empty: If S is empty, then we are in trouble, because we have no more chips to return as the answer. We must stop before this.

$S-C$ Is Empty: If $S-C$ is empty, then we know that all the chips in $S=C$ are the same. Because more than half of them must be good, we know that all of them are good. Hence, we are done.

C Is Empty: If C is empty, take any chip from S and add it to C . We have not changed S , so its loop invariant still holds. The single chip in C is the same as itself.

The Measure of Progress: The measure cannot be $|S|$, because this does not decrease when the chips are the same. Instead, let the measure be $|S - C|$. In two of our cases, we remove a chip from $S - C$ and add it to C . In another case, we remove a chip from $S - C$ and one from C . Therefore, in all cases this measure decreases by 1.

Initial Code: $\langle \text{pre-cond} \rangle \ \& \ \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$. The initial code sets S to be all the chips and C to be empty. More than half of the chips in S are good according to the precondition. Because there are no chips in C , all the chips that are in it are the same.

Exiting Loop: $\langle \text{loop-invariant} \rangle \ \& \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$. $|S - C| = 0$ is a good halting condition, but not the first. Halt when $|C| > |S|/2$, and return any chip from C . According to the loop invariant, the chips in C are either all good or all bad. The chips in C constitute more than half the chips, so if they were all bad, more than half of the chips in S would also be bad. This contradicts the loop invariant. Hence, the chips in C are all good.

Running Time: Initially, the measure of progress $|S - C|$ is n . We showed that it decreases by at least 1 each iteration. Hence, there are at most n steps before $S - C$ is empty. We are guaranteed to exit the loop by this point, because $|S - C| = 0$ assures us that the exit condition $|C| = |S| > |S|/2$ is met. S must contain at least one chip, because by the loop invariant more than half of them are good.

Additional Observations: C can flip back and forth between being all bad and being all good many times. Suppose it is all bad. If s from $S - C$ happens to be bad, then C gets bigger. If s from $S - C$ happens to be good, then C gets smaller. If C ever becomes empty during this process, then a new chip is added to C . This chip may be good or bad. The process repeats.

Extending the Algorithm: The algorithm finds one good chip. This good chip will tell you which of the other chips are good in $O(n)$ time.

Randomized Algorithm: [Chapter 29](#) provides a much easier randomized algorithm for this problem.

4.4 Exercises

Exercise 4.4.1 (See solution in Exercise Solutions.) Search a sorted matrix: The input consists of a real number x and a matrix $A[1 \dots n, 1 \dots m]$ of nm real numbers such that each row $A[i, 1 \dots m]$ is sorted and each column $A[1 \dots n, j]$ is sorted. The goal is to find the maximum array entry $[i, j]$ that is less than or equal to x , or report that all elements of A are larger than x . Design and analyze an iterative algorithm for this problem that examines as few matrix entries as possible. Be careful if you believe that a simple binary search solves the problem. [Exercise 7.0.7](#) asks for a lower bound, and [Exercise 12.1.3](#) for a recursive algorithm.

Exercise 4.4.2 Suppose a train was supposed to start at station A , pause at stations $B, C, D, \dots, Y$, and finish at station Z . However, it did not arrive at Z . Suppose in your factory the piece of equipment or process labeled A makes part B work, which in turn makes $C, D, \dots$, and Z work. However, Z is not working. You want to find out why. What algorithm do you use? Think of other applications of this technique.

Exercise 4.4.3 The following question is not about narrowing the search space. If anything, it is about doubling the size of the search space. But it still has a binary search feel. This question finds the length $D_{\min}(u, v)$ of the shortest path between any pair of nodes in a directed (or undirected) graph. The input provides the length $d(u, v) \geq 0$ of each edge $\langle u, v \rangle$ in the complete graph. It is not necessary that $(u, v) = d(v, u)$, and these lengths may be ∞ . The distance along a path is the sum of the $d(u, v)$ values along its edges. Let $d_i(u, v)$ denote the length of the shortest path from u to v with at most 2^i edges. Do steps 7, 8, 10, and 11, proving that the loop invariant is established and maintained, proving that on exiting the postcondition is established, and bounding the running time. As a hint, trace the algorithm on a graph consisting of a single path, namely, for $j \in [1, n-1]$, $d(u_j, u_{j+1}) = 1$ and all other edges have $d(u, v) = \infty$.

algorithm *Alg*(d)

⟨pre-cond⟩: $d(u, v) \in [0, \infty]$ is the length of edge $\langle u, v \rangle$ in the complete graph.

⟨post-cond⟩: Returned is the length $D_{\min}(u, v)$ of the shortest path from u to v for each pair of nodes.

begin

 for each edge $\langle u, v \rangle$, $D(u, v) = d(u, v)$

 loop $\log_2(n)$ times

⟨loop-invariant⟩: After i iterations, $D_{\min}(u, v) \leq D(u, v) \leq d_i(u, v)$

 for each edge $\langle u, v \rangle$ (iteratively or in parallel)

 % $D_i(u, v) = \text{Min}(D_{i-1}(u, v), \text{Min}_w[D_{i-1}(u, w) + D_{i-1}(w, v)])$

 loop over nodes w

 if $(D(u, v) \geq D(u, w) + D(w, v))$ then

$D(u, v) = D(u, w) + D(w, v)$

 end if

 end loop

 end loop

 end loop

 return (D)

end algorithm

One can find more about this in Exercise 25.6.2.

5 Iterative Sorting Algorithms

Sorting is a classic computational problem. During the first few decades of computers, almost all computer time was devoted to sorting. Many sorting algorithms have been developed. It is useful to know a number of them because sorting needs to be done in many different situations and because they provide a rich selection of examples for demonstrating different algorithmic techniques. We already looked at Selection, Insertion, and Bubble Sort in [Section 1.4](#), which take $O(n^2)$ time. This time can be decreased to $O(n \log n)$ using recursion. See Merge Sort and Quick Sort in [Section 12.1](#) and Heap Sort in [Section 12.1](#). [Section 5.1](#) has a fun iterative version. All of these sorting algorithms are said to be *comparison based*, because the only way of accessing the input values is by comparing pairs of them, i.e., $a_i \leq a_j$. [Chapter 7](#) proves that no comparison-based algorithm can sort faster than $\Omega(n \log n)$ time. In fact, even if more complex *bit* operations are used, $\Omega(n \log n)$ bit operations are needed. In this section, we will present sorting algorithms that run in $O(n)$ time. This requires two changes. First, the input to sorting needs to be restricted. Bucket sort requires the inputs chosen uniformly at random in [Section 5.1](#) and radix/counting sort in [Sections 5.2](#) and [5.3](#) requires having the values within a small range $[1 \dots n^d]$. Second, the computation needs to be able to do more complex operations like adding and indexing in $O(1)$ time.

5.1 Bucket Sort by Hand

Bucket Sort Specifications: The input consists of n values chosen uniformly at random within some range and the output consists of these values sorted. The expected time is $O(n)$ indexing operations. The algorithm partitions the range into n equal-sized *buckets*. Each item is thrown into the appropriate bucket. Because the items are chosen uniformly within the range, it is as if each item is thrown into a random bucket. Like Hashing, only $O(1)$ buckets have $O(\log n)$ items. These can be sorted.

Bucket Sort by Hand Specifications: As a professor, I often have to sort a large stack of students' papers by last name. The algorithm that I use is an iterative version of quick sort and bucket sort. See [Section 12.1](#).

Basic Steps

Partitioning into Five Buckets: Computers are good at using a single comparison to determine whether an element is greater than the pivot value or not.

Humans, on the other hand, tend to be good at quickly determining which of five buckets an element belongs in. I first partition the papers based on which of the following ranges the first letter of the name is within: [A–E], [F–K], [L–O], [P–T], or [U–Z]. Then I partition the [A–E] bucket into the subbuckets [A], [B], [C], [D], and [E]. Then I partition the [A] bucket based on the second letter of the name. This works for this application because the list to be sorted consists of names whose first letters are fairly predictably distributed through the alphabet.

A Stack of Buckets: One difficulty with this algorithm is keeping track of all the buckets. For example, after the second partition, we will have nine buckets: [A], [B], [C], [D], [E], [F–K], [L–O], [P–T], and [U–Z]. After the third, we will have 13. On a computer, the recursion of the algorithm is implemented with a stack of stack frames. Correspondingly, when I sort the students' papers, I have a stack of buckets.

The Loop Invariant: I use the following loop invariant to keep track of what I am doing. The papers are split between a pile of already sorted papers and a stack of piles of partially sorted papers. The papers in the sorted pile (initially empty) come before all the partially sorted papers. Within the partially sorted stack of piles, the papers within each pile are out of order. However, each paper in a pile belongs before each paper in a later pile. For example, at some point in the algorithm, the papers starting with [A–C] will be sorted, and the piles in my stack will consist of [D], [E], [F–K], [L–O], [P–T], and [U–Z].

Maintain Loop Invariant: I make progress while maintaining this loop invariant as follows. I take the top pile off the stack, here the [D]. If it only contains a half dozen or so papers, I sort them using insertion sort. These are then added to the top of the sorted pile, [A–C], giving [A–D]. On the other hand, if the pile [D] taken off the stack is larger than this, I partition it into five piles, [DA–DE], [DF–DK], [DL–DO], [DP–DT], and [DU–DZ], which I push back onto the stack. Either way, my loop invariant is maintained.

Exit Condition: When the last bucket has been removed from the stack, the papers are sorted.

Exercise 5.1.1 Try sorting a deck of cards using this algorithm.

Exercise 5.1.2 Give code for this algorithm.

5.2 Counting Sort (a Stable Sort)

The counting sort algorithm is only useful in the special case where the elements to be sorted have very few possible values.

Specifications

Preconditions: The input is a list of N values $a_0, \dots, a_{N-1}$, each within the range $0, \dots, k-1$.

Postconditions: The output is a list consisting of the same N values in nondecreasing order. The sort is *stable*, meaning that if two elements have the same value, then they must appear in the same order in the output as in the input. (This is important when extra data is carried with each element.)

Basic Steps

Where an Element Goes: Consider any element of the input. By counting, we will determine where this element belongs in the output, and then we simply put it there. Where it belongs is determined by the number of elements that must appear before it. To simplify the argument, let's index the locations with $[0, N - 1]$. This way, the element in the location indexed by 0 has no elements before it, and the element in location $\hat{c}$ has $\hat{c}$ elements before it.

Suppose that the element a_i has the value v . Every element that has a strictly smaller value must go before it. Let's denote this count with $\hat{c}_v$, that is, $\hat{c}_v = |\{j | a_j < v\}|$. The only other elements that go before a_i are elements with exactly the same value. Because the sort must be stable, the number of these that go before it is the same as the number that appear before it in the input. If this number happens to be q_{a_i} , then element a_i belongs in location $\hat{c}_v + q_{a_i}$. In particular, the first element in the input with value v goes in location $\hat{c}_v + 0$.

Example

Input: 1 0 1 0 2 0 0 1 2 0

Output: 0 0 0 0 0 1 1 1 2 2

Index: 0 1 2 3 4 5 6 7 8 9

The first element to appear in the input with value 0 goes into location 0, because there are $\hat{c}_0 = 0$ elements with smaller values. The next such element goes into location 1, the next into 2, and so on.

The first element to appear in the input with value 1 goes into location 5, because there are $\hat{c}_1 = 5$ elements with smaller values. The next such element goes into location 6, and the next into 7.

Similarly, the first element with value 2 goes into location $\hat{c}_2 = 8$.

Computing $\hat{c}_v$: We could compute $\hat{c}_v$ by making a pass through the input, counting the number of elements that have values smaller than v . Doing this separately for each value $v \in [0 \dots k - 1]$, however, would take $O(kN)$ time, which is too much.

Instead, let's first count how many times each value occurs in the input. For each $v \in [0 \dots k - 1]$, let $c_v = |\{i | a_i = v\}|$. This count can be computed with one pass through the input. For each element, if the element has value v , increment the counter c_v . This requires only $O(N)$ addition and indexing operations.

Given the c_v values, we could compute $\hat{c}_v = \sum_{v'=0}^{v-1} c_{v'}$. Computing one such $\hat{c}_v$ would require $O(k)$ additions, and computing all of them would take $O(k^2)$ additions, which is too much.

Alternatively, note that $\hat{c}_0 = 0$ and $\hat{c}_v = \hat{c}_{v-1} + c_{v-1}$. Of course, we must have computed the previous values before computing the next. Now computing

one such $\hat{c}_v$ takes $O(1)$ additions, and computing all of them takes only $O(k)$ additions.

Put in Place: The main loop in the algorithm considers the input elements one at a time in the order $a_0, \dots, a_{N-1}$ that they appear in the input and places them in the output array where they belong.

The Loop Invariant

1. The input elements that have already been considered have been put in their correct places in the output.
2. For each $v \in [0 \dots k-1]$, $\hat{c}_v$ gives the index in the output array where the next input element with value v goes.

Establishing the Loop Invariant: Compute the counts $\hat{c}_v$ as described above. This establishes the loop invariant before any input elements are considered, because this $\hat{c}_v$ value gives the location where the first element with value v goes.

Main Step: Take the next input element. If it has value v , place it in the output location indexed by $\hat{c}_v$. Then increment $\hat{c}_v$.

Maintain Loop Invariant: $\langle \text{loop-invariant}' \rangle$ & not $\langle \text{exit-cond} \rangle$ & $\text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. By the loop invariant, we know that if the next input element has value v , then it belongs in the output location indexed by $\hat{c}_v$. Hence, it is being put in the correct place. The next input element with value v will then go immediately after this current one in the output, i.e., into location $\hat{c}_v + 1$. Hence, incrementing $\hat{c}_v$ maintains the second part of the loop invariant.

Exit Condition: Once all the input elements have been considered, the first loop invariant establishes that the list has been sorted.

Code

```

 $\forall v \in [0 \dots k-1], c_v = 0$ 
loop  $i = 0$  to  $N-1$ 
     $++ c_{a[i]}$ 
 $\hat{c}_0 = 0$ 
loop  $v = 1$  to  $k-1$ 
     $\hat{c}_v = \hat{c}_{v-1} + c_{v-1}$ 
loop  $i = 0$  to  $N-1$ 
     $b[\hat{c}_{a[i]}] = a[i]$ 
     $++ \hat{c}_{a[i]}$ 

```

Running Time: The total time is $O(N+k)$ addition and indexing operations. If the input can only contain $k = O(N)$ possible values, then this algorithm works in linear time. It does not work well if the number of possible values is much higher.

5.3 Radix Sort

The radix sort is a useful algorithm that dates back to the days of card-sorting machines, now found only in computer museums.

Specifications

Preconditions: The input is a list of N values. Each value is an integer with d digits. Each digit is a value from 0 to $k - 1$, i.e., the value is viewed as an integer base k .

Postconditions: The output is a list consisting of the same N values in nondecreasing order.

Basic Steps: For some digit $i \in [1 \dots d]$, sort the input according to the i th digit, ignoring the other digits. Use a stable sort, such as counting sort.

Examples: Old computer punch cards were organized into $d = 80$ columns, and in each column a hole could be punched in one of $k = 12$ places. A card-sorting machine could mechanically examine each card in a deck and distribute the card into one of 12 bins, depending on which hole had been punched in a specified column.

A “value” might consist of a year, a month, and a day. You could then sort the elements by the year, by the month, or by the day.

Order in Which to Consider the Digits: It is most natural to sort with respect to the most significant digit first. The final sort, after all, has all the elements with a 0 as the first digit at the beginning, followed by those with a 1.

If the operator of the card-sorting machine sorted first by the most significant digit, they would get 12 piles. Each of these piles would then have to be sorted separately, according to the remaining digits. Sorting the first pile according to the second digit would produce 12 more piles. Sorting the first of those piles according to the third digit would produce 12 more piles. The whole process would be a nightmare.

Sorting with respect to the least significant digit seems silly at first. Sorting (79, 94, 25) gives (94, 25, 79), which is completely wrong. Even so, this is what the algorithm does.

The Algorithm: Loop through the digits from low to high order. For each, use a stable sort to sort the elements according to the current digit, ignoring the other digits.

Example

Sorted by first 3 digits	Considering 4th digit	Stably sorted by 4th digit
184	3184	1195
192	5192	1243
195	1195	1311
243	1243	3184
271	3271	3271
311	1311	5192

The result is sorted by the first four digits.

Loop Invariant: After sorting with respect to (wrt) the first i low-order digits, the elements are sorted wrt the value formed from these i digits.

Establishing the Loop Invariant: The loop invariant is initially trivially true, because initially no digits have been considered.

Maintain Loop Invariant: $\langle \text{loop-invariant}' \rangle \ \& \ \text{not} \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$. Suppose that the elements are sorted wrt the value formed from the lowest $i - 1$ digits. For the elements to be sorted wrt the value formed from the lowest i digits, all the elements with a 0 in the i th digit must come first, followed by those with a 1, and so on. This can be accomplished by sorting the elements wrt the i th digit while ignoring the other digits. Moreover, the block of elements with a 0 in the i th digit must be sorted wrt the lowest $i - 1$ digits. By the loop invariant, they were in this order, and because the sorting wrt the i th digit was stable, these elements will remain in the same relative order. The same is true for the block of elements with a 1 or 2 or $\dots$ in the i th digit.

Ending: $\langle \text{loop-invariant} \rangle \ \& \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$. When $i = d$, they are sorted wrt the value formed from all d digits, and hence are sorted.

Radix Counting Sort Specifications: The input consists of n integer values in the range $[0 \dots n^d]$ and the output consists of these values sorted. The time is $O(dn)$ adding and indexing operations, which is $O(n)$ when d is a constant. The algorithm views each input value as a d digit integer where each digit is in the range $[0 \dots n]$. This is done as follows. Being an integer at most n^d , the input can be represented in binary using $d \log n$ bits. Partition these bits into d blocks of $\log n$ bits each. View each such block as a digit at most n . The algorithm proceeds by doing radix sort with respect to these digits. Each digit is sorted using counting sort, where $k = n$. The running time is $O(d(n + k)) = O(dn)$.

6 More Iterative Algorithms

6.1 Euclid's GCD Algorithm

More-of-the-input iterative algorithms extend a solution for a smaller input instance into a larger one. We will see in [Chapter 12](#) that recursive algorithms do this too. The following is an amazing algorithm that does this. It finds the greatest common divisor (GCD) of two integers. For example, $GCD(18, 12) = 6$. It was first done by Euclid, an ancient Greek mathematician. Without the use of loop invariants, you would never be able to understand what the algorithm does; with their help, it is easy.

Specifications: An input instance consists of two positive integers, a and b . The output is $GCD(a, b)$.

The Loop Invariant: Like many loop invariants, designing this one required creativity. The algorithm maintains two variables x and y whose values change with each iteration of the loop under the invariant that their GCD, $GCD(x, y)$, does not change, but remains equal to the required output $GCD(a, b)$.

Type of Loop Invariant: This is a strange loop invariant. The algorithm is more like recursion. A solution to a smaller instance of the problem gives the solution to the original.

Establishing the Loop Invariant: The easiest way of establishing the loop invariant that $GCD(x, y) = GCD(a, b)$ is by setting x to a and y to b .

Measure of Progress: Progress is made by making x or y smaller.

Ending: We will exit when x or y is small enough that we can compute their GCD easily. By the loop invariant, this will be the required answer.

A Middle Iteration on a General Instance: Let us first consider a general situation in which x is bigger than y and both are positive.

Main Steps: Our goal is to make x or y smaller without changing their GCD. A useful fact is that $GCD(x, y) = GCD(x - y, y)$, e.g., $GCD(52, 10) = GCD(42, 10) = 2$, because any value that divides x and y also divides $x - y$, and similarly any value that divides $x - y$ and y also divides x . Hence, replacing x with $x - y$ would make progress while maintaining the loop invariant.

Exponential Running Time? A good idea when you are considering a loop invariant and iterations is to jump ahead in designing the algorithm and estimate

its running time. A loop executing only $x = x - y$ will iterate $\frac{a}{b}$ times. However, even if $b = 1$, this is only a iterations. This looks like it is linear time. However, you should express the running time of an algorithm as a function of input size. See [Section 35.1](#). The number of bits needed to represent the instance $\langle a, b \rangle$ is $n = \log a + \log b$. Expressed in these terms, the running time is $\text{Time}(n) = \Theta(a) = \Theta(2^n)$. This is exponential time. If $a = 1,000,000,000,000,000$ and $b = 1$, I would not want to wait for it.

Faster Main Steps: One thing to try when faced with exponential running time is to look for a way to speed up the main steps. Instead of subtracting one y from x each iteration, why not speed up the process by subtracting a multiple of y all at once? We could set $x_{\text{new}} = x - d \cdot y$ for some integer value of d . Our goal is to make x_{new} as small as possible without making it negative. Clearly, d should be $\lfloor \frac{x}{y} \rfloor$. This gives $x_{\text{new}} = x - \lfloor \frac{x}{y} \rfloor \cdot y = x \bmod y$, which is within the range $[0 \dots y - 1]$ and is the remainder when dividing y into x . For example, $52 \bmod 10 = 2$.

Maintaining the Loop Invariant: The step $x_{\text{new}} = x \bmod y$ maintains the loop invariant because $\text{GCD}(x, y) = \text{GCD}(x \bmod y, y)$, e.g., $\text{GCD}(52, 10) = \text{GCD}(2, 10) = 2$.

Making Progress: The step $x_{\text{new}} = x \bmod y$ makes progress by making x smaller only if $x \bmod y$ is smaller than x . This is only true if x is greater than or equal to y . Suppose that initially this is true because a is greater than b . After one iteration, $x_{\text{new}} = x \bmod y$ becomes smaller than y . Then the next iteration will do nothing. A solution is to then swap x and y .

New Main Steps: Combining $x_{\text{new}} = x \bmod y$ with a swap gives the main steps of $x_{\text{new}} = y$ and $y_{\text{new}} = x \bmod y$.

Maintaining the Loop Invariant: This maintains our original loop invariant because $\text{GCD}(x, y) = \text{GCD}(y, x \bmod y)$, e.g., $\text{GCD}(52, 10) = \text{GCD}(10, 2) = 2$. It also maintains the new loop invariant that $0 \leq y \leq x$.

Making Progress: Because $y_{\text{new}} = x \bmod y \in [0 \dots y - 1]$ is smaller than y , we make progress by making y smaller.

Special Cases: Setting $x = a$ and $y = b$ does not establish the loop invariant, which says that x is at least y if a is smaller than b . An obvious solution is to initially test for this and to swap x and y if necessary. However, as advised in [Section 1.2](#), it is sometimes fruitful to try tracing out what the algorithm that you have already designed would do given such an input. Suppose $a = 10$ and $b = 52$. The first iteration would set $x_{\text{new}} = 52$ and $y_{\text{new}} = 10 \bmod 52$. This last value is a number within the range $[0 \dots 51]$ that is the remainder when dividing 10 by 52. Clearly this is 10. Hence, the code automatically swaps the values by setting $x_{\text{new}} = 52$ and $y_{\text{new}} = 10$. Hence, no new code is needed. Similarly, if a and b happen to be negative, the initial iteration will make y positive, and the next will make both x and y positive.

Exit Condition: We are making progress by making y smaller. We should stop when y is small enough that we can compute the GCD easily. Let's try small values of y . Using $GCD(x, 1) = 1$, the GCD is easy to compute when $y = 1$; however, we will never get this unless $GCD(a, b) = 1$. How about $GCD(x, 0)$? This turns out to be x , because x divides evenly into both x and 0 . Let's try an exit condition of $y = 0$.

Termination: We know that the program will eventually stop as follows: $y_{new} = x \bmod y$, $y \in [0 \dots y - 1]$ ensures that each step y gets strictly smaller and does not go negative. Hence, eventually y must be zero.

Ending: Formally we prove that $\langle loop\text{-}invariant \rangle \ \& \ \langle exit\text{-}cond \rangle \ \& \ code_{post\text{-}loop} \Rightarrow \langle post\text{-}cond \rangle$. We see that $\langle loop\text{-}invariant \rangle$ gives $GCD(x, y) = GCD(a, b)$ and $\langle exit\text{-}cond \rangle$ gives $y = 0$. Hence, $GCD(a, b) = GCD(x, 0) = x$. The final code will return the value of x . This establishes the $\langle post\text{-}cond \rangle$ that $GCD(a, b)$ is returned.

Code

algorithm $GCD(a, b)$

$\langle pre\text{-}cond \rangle$: a and b are integers.

$\langle post\text{-}cond \rangle$: Returns $GCD(a, b)$.

begin

 int x, y

$x = a$

$y = b$

 loop

$\langle loop\text{-}invariant \rangle$: $GCD(x, y) = GCD(a, b)$.

 if ($y = 0$) exit

$x_{new} = y, y_{new} = x \bmod y$

$x = x_{new}$

$y = y_{new}$

 end loop

 return(x)

end algorithm

Example: The following traces the algorithm given two input instances, $\langle a, b \rangle = \langle 22, 33 \rangle$ and $\langle a, b \rangle = \langle 1,000,000,005,999,999,999 \rangle$.

Iteration	Value of x	Value of y
1 st	22	32
2 nd	32	22
3 rd	22	10
4 th	10	2
5 th	2	0

$GCD(22, 32) = 2$.

Iteration	Value of x	Value of y
1 st	1,000,000,005	999,999,999
2 nd	999,999,999	6
3 rd	6	3
4 th	3	0

$GCD(1,000,000,005,999,999,999) = 3$

Running Time: For the running time to be linear in the size of the input, the number of bits ($\log y$) to represent y must decrease by at least one in each iteration. This means

that the value of y must decrease by at least a factor of two. Consider the example of $x = 19$ and $y = 10$. Then y_{new} becomes $19 \bmod 10 = 9$, which is only a decrease of one. However, the next value of y will be $10 \bmod 9 = 1$, which is a huge drop.

We will be able to prove that every two iterations, y drops by a factor of 2, namely, that $y_{k+2} < y_k/2$. There are two cases. In the first case, $y_{k+1} \leq y_k/2$. Then we are done, because, as stated above, $y_{k+2} < y_{k+1}$. In the second case, $y_{k+1} \in [y_k/2 + 1, y_k - 1]$. Unwinding the algorithm gives that $y_{k+2} = x_{k+1} \bmod y_{k+1} = y_k \bmod y_{k+1}$. One algorithm for computing $y_k \bmod y_{k+1}$ is to continually subtract y_{k+1} from y_k until the amount is less than y_{k+1} . Because y_k is more than y_{k+1} , this y_{k+1} is subtracted at least once. It follows that $y_k \bmod y_{k+1} \leq y_k - y_{k+1}$. By the case, $y_{k+1} > y_k/2$. In conclusion, $y_{k+2} = y_k \bmod y_{k+1} \leq y_k - y_{k+1} < y_k/2$.

We prove that the number of times that the loop iterates is $O(\log(\min(a, b))) = O(n)$, as follows. After the first or second iteration, y is $\min(a, b)$. Every iteration y goes down by at least a factor of 2. Hence, after k iterations, y_k is at most $\min(a, b)/2^k$, and after $O(\log(\min(a, b)))$ iterations it is at most one.

The algorithm iterates a linear number $O(n)$ of times. Each iteration must do a *mod* operation. Poor Euclid had to compute these by hand, which must have gotten very tedious. A computer may be able to do *mods* in one operation; however, the number of bit operations needed for two n -bit inputs is $O(n \log n)$. Hence, the time complexity of this GCD algorithm is $O(n^2 \log n)$.

Lower Bound: We will prove a lower bound, not of the minimum time for any algorithm to find the GCD, but of this particular algorithm, by finding a family of input values $\langle a, b \rangle$ for which the program loops $\Theta(\log(\min(a, b)))$ times. Unwinding the code gives $y_{k+2} = x_{k+1} \bmod y_{k+1} = y_k \bmod y_{k+1}$. As stated, $y_k \bmod y_{k+1}$ is computed by subtracting y_{k+1} from y_k a number of times. We want the y s to shrink as slowly as possible. Hence, let us say that it is subtracted only once. This gives $y_{k+2} = y_k - y_{k+1}$ or $y_k = y_{k+1} + y_{k+2}$. This is the definition of Fibonacci numbers, only backward, i.e., $Fib(0) = 0$, $Fib(1) = 1$, and $Fib(n) = Fib(n-1) + Fib(n-2)$. (See Exercise 39.2.1.) On input $a = Fib(n+1)$ and $b = Fib(n)$, the program iterates n times. This is $\Theta(\log(\min(a, b)))$, because $Fib(n) = 2^{\Theta(n)}$.

Exercise 6.1.1

algorithm *Converge*($x_{original}$)

⟨pre-cond⟩: $x_{original} \in [0, \dots, 1]$.

⟨post-cond⟩: *This algorithm returns the converged value???*

or runs forever

begin

$x = x_{original}$

loop

⟨loop-invariant⟩: $x \in [0, \dots, 1]$

exit when ($x = f(x)$)

$x = f(x)$

end loop

```

    return(x)
end algorithm

```

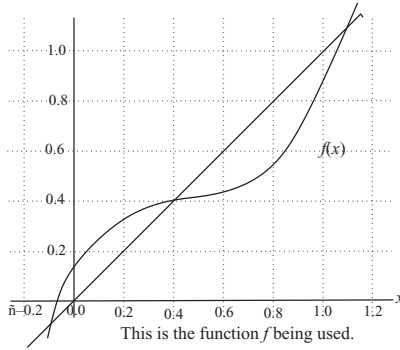


Fig. 6.1 Function used by Converge.

1. Prove that the algorithm correctly establishes the loop invariant.
2. Prove that the loop invariant is maintained.
3. Fill in the rest of the postcondition by giving as specifically as possible which value is returned by this algorithm when it does converge. Prove that if the algorithm halts, then this postcondition is met.
4. Change the algorithm so that it also has an integer input N and it halts after N iterations. What is the running time (time complexity) of this algorithm as a function of the size of the input?
5. Change the algorithm so that it halts after a billion iterations. What is the running time (time complexity) of this algorithm as a function of the size of the input?

6.2 Multiplying

Section 1.5 Code from *LI* developed a simple algorithm using mechanical steps starting with the loop invariant. In the same way, we will now develop both the elementary and the high school algorithm for multiplying integers $X \times Y$.

Elementary School Algorithm: The most obvious algorithm is $X \times Y = Y + Y + Y + \dots + Y$ adding X times. One way to implement this is to add Y to a running sum s each iteration while decrementing x from X to zero.

Loop Invariant: When x has decrease by $X-x$, the running sum will be $s = Y \times (X-x)$. Rearranging this gives our loop invariant that

$$X \times Y = x \times y + s.$$

This is similar to the GCD in that a solution to a smaller instance of the problem gives the solution to the original.

Preconditions: We establish this with the minimal work by setting $x = X$, $y = Y$, and $s = 0$.

Ending with Postcondition: Our measure of progress is that x decreases. If we exit when $x = 0$, then the loop invariant gives us the answer $X \times Y = 0 \times y + s = s$.

Head in the Right Direction & Return to Path: Because it is hard to know how to make progress and maintain the loop invariant at the same time, my advice is to do them as two separate steps.

Make Progress: Start by heading in the direction that “makes progress” in the easiest way possible. It sounds silly, but if x is the measure of progress, then to make progress the computation only needs to decrease x . This gives that $code_{loop}$ contains the code $x = x - 1$.

Reestablishing the Loop Invariant: What remains for our computation is to fix the rest of the data structure in the minimal way so that the loop invariant is true again. This is how we determine the rest of the actions $code_{loop}$.

Maintaining Loop Invariant: We need to design $code_{loop}$ in a way to make the following true.

$$\langle loop-invariant_t \rangle \ \& \ not \ \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant_{t+1} \rangle.$$

Let x_t and s_t be the values of x and s at the beginning of the iteration and let x_{t+1} and s_{t+1} be their values after going around again.

Let’s work forward from our assumptions and backward from what we want to prove.

Have $\langle loop-invariant_t \rangle$: Knowing that the loop invariant is true for the original values, we can just copy down this statement with these values substituted in, i.e., we know that $X \times Y = x_t \times y_t + s_t$.

Have $code_{loop}$: All we know so far is that $x_{t+1} = x_t - 1$. Let’s not change the value of y , giving $y_{t+1} = y_t$. What needs to change is s_{t+1} .

Want $\langle loop-invariant_{t+1} \rangle$: Wanting that the loop invariant to be true for the new values, we can just copy down this statement at the end of our proof, i.e., we need to prove that $X \times Y = x_{t+1} \times y_{t+1} + s_{t+1}$.

LHS vs. RHS: A typical way to prove that the left-hand and right-hand sides of an equation are equal is to start at both ends and try to work your way to the middle. We will use both the previous loop invariant and the loop code.

$$\begin{aligned} LHS &= X \times Y = x_t \times y_t + s_t = (x_{t+1} + 1) \times y_{t+1} + s_t \\ &= x_{t+1} \times y_{t+1} + (s_t + y_{t+1}), \\ RHS &= x_{t+1} \times y_{t+1} + s_{t+1}. \end{aligned}$$

Combining LHS and RHS gives that what we want is $s_{t+1} = s_t + y_{t+1}$. This is not surprising, because $s = s + y$ is the code we started with.

Running Time: Because x decrements from X to zero, the number of iterations is X . This looks like polynomial time, but polynomial in what? The number of bits to represent the input is $n = \log X + \log Y$. This makes the number of iterations be $X = O(2^n)$, which is officially exponential time.

High School Algorithm: Because the elementary school algorithm is so slow, we will now develop the high school algorithm for multiplying.

Make Progress: To make the algorithm faster, let's cut the value of x in half, i.e., $x_{t+1} = \frac{1}{2}x_t$. Because x decreases by a factor of two from X to 1, the number of iterations is $\log X$, which is the number of bits to write down the value X .

Maintaining Loop Invariant: We need to design $code_{loop}$ in a way to make the following true.

$$\langle loop-invariant_t \rangle \ \& \ not \ \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant_{t+1} \rangle.$$

We have $\langle loop-invariant_t \rangle$ that $X \times Y = x_t \times y_t + s_t$. We have $code_{loop}$ that $x_{t+1} = \frac{1}{2}x_t$. We want $\langle loop-invariant_{t+1} \rangle$ that $X \times Y = x_{t+1} \times y_{t+1} + s_{t+1}$.

LHS vs. RHS: Again, let's work from both ends.

$$\begin{aligned} LHS &= X \times Y = x_t \times y_t + s_t = (2x_{t+1}) \times y_t + s_t \\ &= x_{t+1} \times (2y_t) + s_t, \\ RHS &= x_{t+1} \times y_{t+1} + s_{t+1}. \end{aligned}$$

Combining LHS and RHS gives that what we want is $y_{t+1} = 2y_t$ and $s_{t+1} = s_t$, i.e., we double y to compensate for halving x and we keep s the same.

Bug: If x is odd, then we can't divide it by 2. We could develop a new algorithm in this case. But let's just do what we did above.

Code:

```
begin routine Multiply(X,Y)
   $\langle pre-cond \rangle$ : Integers
   $x = X; y = Y; s = 0$ 
  loop
     $\langle loop-invariant \rangle$ :  $X \times Y = x \times y + s$ 
    exit when  $(x = 0)$ 
    if  $(x \text{ is even})$ 
       $x = \frac{1}{2}x; y = 2y$            % Lots of progress because  $x$  loses a bit.
    else
       $x = x - 1; s = s + y$        % Lots of progress because  $x$  becomes even.
  return( $s$ )
   $\langle post-cond \rangle$ : Returns  $X \times Y$ .
end routine
```

Running Time: The algorithm iterates $\log X$ times in which X is halved and at most the same number in which X is made even. Each iteration requires adding in y , which takes $O(\log Y)$ bit operations because this is the number of bits to represent the value y . This gives a running time of $O(\log X \cdot \log Y) = O(n^2)$, where $n = \log X + \log Y$ is the number of bits to represent the input.

Ethiopians: The ancient Egyptians and the Ethiopians had advanced mathematics. Merely by halving and doubling, they could multiply any two numbers correctly. Say they want to buy 15 sheep at 13 Ethiopian dollars each. Here is how they figure out the product. Put 13 in a left column, 15 on the right. Halving the left value; you get $6\frac{1}{2}$. Ignore the $\frac{1}{2}$. Double the right value. Repeat this (keeping all intermediate values) until the left value is 1. What you have is

13	15
6	30
3	60
1	120

Even numbers in the left column are evil and, according to the story, must be destroyed, along with their guilty partners. So scratch out the 6 and its partner 30. Now add the right column giving $15 + 60 + 120 = 195$, which is the correct answer. Note that this algorithm is the same as the one we just developed. The sum s adds all the values of y for which x was odd.

The High School Algorithm: Though this algorithm seems strange, it is in fact exactly the same as the high school algorithm for multiplying in binary. Let's first understand multiplication $X \times Y$ in terms of bits. Let $a_4a_3a_2a_1a_0$ denote the bits of X written in binary. For example, $X = 5 = 101_2$ and $Y = 58 = 111010_2$. The bit a_i is shifted into place by multiplying it by 2^i . This gives $X = \sum_i 2^i a_i$. Plugging this in gives that $X \times Y = [\sum_i 2^i a_i] \times Y = \sum_i [2^i a_i Y]$.

$$\begin{array}{r}
 Y \quad 111010 \\
 X \quad \underline{101} \\
 111010 \quad 2^0 a_0 Y \\
 000000 \quad 2^1 a_1 Y \\
 111010 \quad 2^2 a_2 Y \\
 \hline
 100100010
 \end{array}$$

Now let's see how the high school algorithm for multiplying in binary implements this. First the bits of Y are written on the top line and the bits of X , namely, $a_4a_3a_2a_1a_0$, are written below it. On the i^{th} line below this, we multiply a_i by Y and shift this over i spots. This gives $2a_i y$ on this i^{th} line. The algorithm then adds these lines up giving the answer $\sum_i 2a_i y$ as required.

Now let's consider our algorithm. Let's first see how it determines the bits of X . The variable x starts at X and comes down by a factor of two rounded down each iteration and hence after i iterations is $x_i = \lfloor \frac{X}{2^i} \rfloor$. Similarly $y_i = 2^i Y$. An iteration adds in y_i to s iff x_i is odd. Then $x_i = \lfloor \frac{X}{2^i} \rfloor$ is odd iff the i^{th} bit of X is $a_i = 1$. In our example, $a_0 = 1$ because $x_0 = X = 5$ is odd, $a_1 = 0$ because $x_1 = \lfloor \frac{X}{2} \rfloor = 2$ is even, and $a_2 = 1$ because $x_2 = \lfloor \frac{X}{2^2} \rfloor = 1$ is odd. Hence, the bit a_i indicates whether or not y_i is added to s . Hence, our answer, which is the final value of s , is $\sum_i a_i y = \sum_i 2a_i y$ as required.

7 The Loop Invariant for Lower Bounds

Time Complexity: The time complexity of a computational problem P is the minimum time needed by an algorithm to solve it:

$$\begin{aligned} \exists A, \forall I, [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq T_{upper}(|I|)] \\ \forall A, \exists I, [A(I) \neq P(I) \text{ or } \text{Time}(A, I) \geq T_{lower}(|I|)]. \end{aligned}$$

Asymptotic Notation: When we want to bound the running time of an algorithm while ignoring multiplicative constants, we use the following notation.

Name	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = O(g(n))$	$f(n) \leq O(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$

See [Chapter 37](#).

An Upper Bound Is an Algorithm: An upper bound for P is obtained by constructing an algorithm A that outputs the correct answer, namely, $A(I) = P(I)$, within the bounded time, i.e., $\text{Time}(A, I) \leq T_{upper}(|I|)$, on every input instance I .

A Lower Bound Is an Algorithm: Amusingly enough, lower bound, proving that there is no faster algorithm for the problem P , is also obtained by constructing an algorithm, but it is an algorithm for a different problem. The input to this problem is an algorithm A claiming to solve P in the required time. The output, as proof this is false, is an input instance I on which the given algorithm A either does not give the correct answer, namely, $A(I) \neq P(I)$, or uses too much time, namely, $\text{Time}(A, I) \geq T_{lower}(|I|)$.

Read Part IV: Additional Topics: To understand this better you may have to read two discussions in [Part IV: Chapter 34](#) on how to think of statements with existential and universal quantifiers as a game between two players, and [Section 35.2](#) on time complexity.

Circular Argument: Proving lower bounds can lead to the following circular argument. Given an arbitrary algorithm A , we must find an input instance I for which A gives the wrong answer. The problem is that you do not know for which input instance the algorithm will give the wrong answer until you know what the algorithm does. But you do not know what the algorithm does until you give it an input instance and

run it. This paradox is avoided by stepping through the computation on A one time step at a time, at each step narrowing the search space for I . This makes your algorithm for solving the lower bound problem an iterative algorithm. As such, it needs a loop invariant.

The Loop Invariant Argument

Knowledge: At each time step, the actions taken by algorithm A depend on the *knowledge* that it has collected already. For example, if the input is $\langle x_1, \dots, x_n \rangle$ and during the first time step A tests if $x_5 < x_6$, then A can base what it does during the second time step on whether or not $x_5 < x_6$, but it does not yet *know* anything else about the input instance. We define A 's knowledge, or *state*, to be determined by the values of its variables (except for the variables storing the input instance) and which line of code it is on.

The Loop Invariant: The loop invariant will be a classic narrowing-the-search-space type. It states that we have a set S of input instances on which algorithm A 's knowledge and actions for its first t time steps are identical.

Establishing the Loop Invariant: Initially the set S is some large set of instances that we want to focus on. The loop invariant is trivially established for $t = 0$, because initially the algorithm knows nothing and has done nothing.

Maintaining the Loop Invariant: The loop invariant is maintained as follows. Assume that it is true at time $t - 1$. Though we do not know which input instance I from S will ultimately be given to algorithm A , we do know that what A learns during its first $t - 1$ time steps is independent of this choice. Algorithm A , knowing what it has learned during these first $t - 1$ steps, but unaware that it has not been given a specific input instance, will then state what action it will do during time step t . What A learns at time t from this action will depend on which instance $I \in S$ is given to A . We then partition S based on what A learns and narrow S down to one such part. This maintains the loop invariant, i.e., that we have a set S of input instances on which the algorithm A 's knowledge and actions for its first t time steps are identical.

Measure of Progress: The measure of progress for our lower bound algorithm is that S does not get too much smaller.

Exit Condition: The exit condition is then $t = T_{\text{lower}}(|I|)$.

Ending: From the loop invariant and the exit condition, we obtain the postcondition by finding two input instances I and I' in S for which the computational problem P requires different outputs: $P(I) \neq P(I')$. If, on instance I , algorithm A either does not give the correct answer, so that $A(I) \neq P(I)$, or uses too much time, so that $\text{Time}(A, I) \geq T_{\text{lower}}(|I|)$, then the postcondition is met. Otherwise, we turn our attention to instance I' . By the loop invariant and the exit condition, the computation of A is identical on the two instances I and I' for the first $T_{\text{lower}}(|I|)$ time steps, because both I and I' are in S . Hence, their outputs must be identical:

$A(I) = A(I')$. By our choice of instances, $P(I) \neq P(I')$. Because $A(I) = P(I)$, it follows that $A(I') \neq P(I')$. Again we have found an instance on which A does not give the correct answer, and the postcondition is met.

Example 7.0.1 (Sorting)

We have seen a number of algorithms that can sort N numbers using $O(N \log N)$ comparisons between the elements, such as merge, quick, and heap sort. We will prove that no algorithm can sort faster.

Information Theory: The lower bounds technique just described does not consider the amount of work that must get done to solve the problem, but the amount of *information* that must be transferred from the input to the output. The problem with these lower bounds is that they are not bigger than linear with respect to the bit size of the input and the output.

$n = \Theta(N \log N)$: At first, it may appear that this is a superlinear lower bound. However, N is the number of elements in the list. Assuming that the N numbers to be sorted are distinct, each needs $\Theta(\log N)$ bits to be represented, for a total of $n = \Theta(N \log N)$ bits. Hence, the lower bound does not in fact say that more than $\Theta(n)$ bit operations are required when n is the total number of bits in the input instance.

Definition of Binary Operation: Before we can prove that no algorithm exists that quickly sorts, we need to first be very clear about what an algorithm is and what its running time is. This is referred to as a *model of computation*. For this sorting lower bound, we will be very generous. We will allow the algorithm to perform any binary operation. This operation can use any information about the input or about what has already been computed by the algorithm, but the result of the operation is restricted to a yes–no answer. For example, as is done in merge sort, it could ask whether the i th element is less than the j th element. For a stranger example, it could ask with one operation whether the number of odd elements is odd.

Definition of the Sorting Problem P : The standard sorting problem, given N elements, is to output the same N elements in sorted order. To make our life easier, we will define the problem P to sort pointers to the elements instead of sorting the elements themselves. Note this is often done when elements are too large to move easily. For example, if the input is $I = \langle 19, 5, 81 \rangle$ pointers to these elements are $\langle 1, 2, 3 \rangle$, the output will be $\langle 2, 1, 3 \rangle$, because the first element in the sorted order $\langle 5, 19, 81 \rangle$ was second in I , the second was the first, and the third element was the third. Similarly, the output for $I' = \langle 19, 81, 5 \rangle$ will be $\langle 3, 1, 2 \rangle$. What makes our lives easier in this version of the sorting problem is that the instances $I = \langle 19, 5, 81 \rangle$ and $I' = \langle 19, 81, 5 \rangle$ have different outputs, while in the standard problem definition, they would both have the output $\langle 5, 19, 81 \rangle$. This change is reasonable because any sorting algorithm needs to learn the order that the elements should be in.

The Initial Set of Instances: Because of the way we modified the sorting problem, the nature of the elements being sorted does not matter, only their initial order. Hence,

we may as well assume that we are sorting the numbers 1 to N . Let the initial set S of input instances being considered consist of every permutation of these numbers.

$P(I) \neq P(I')$: Note that each pair of instances $I, I' \in S$ has different outputs for the sorting problem P , i.e., $P(I) \neq P(I')$. This is good because our search is supposed to end by finding two input instances I and I' in S for which this is the case.

The Measure of Progress: Our measure of progress, as we search for an instance I on which algorithm A does not work, will be the number $|S|$ of instances still being considered. Initially, because S consists of all permutations of N elements, $|S| = N!$. We will prove that each iteration, S does not decrease by more than a factor of 2. Hence, after t iterations, $|S| \geq N! / 2^t$. By setting $T_{\text{lower}}(|I|)$ to be $\log_2(N!) - 1$, we know that in the end we have at least two input instances remaining to be our I and I' .

Math: In $N! = 1 \times 2 \times 3 \times \dots \times N$, $\frac{N}{2}$ of the factors are at least $\frac{N}{2}$, and all N of the factors are at most N . Hence, $N!$ is in the range $[N/2^{N/2}, N^N]$. Hence, $\log N!$ is in the range $[\frac{N}{2} \log \frac{N}{2}, N \log N]$.

Maintaining the Loop Invariant: Assume that the loop invariant is true at time $t - 1$ and that S is the set of input instances on which algorithm A 's knowledge and actions for its first $t - 1$ time steps are identical. Given this, the action A will perform during time step t is fixed. The model of computation dictates that the result of A 's actions is restricted to a yes–no answer. We then partition S into two sets based on whether this answer on this instance I is yes or no. We simply narrow S down to the part of S that is larger of the two. Restricting the algorithm to learning only this one answer maintains the loop invariant. Clearly, the larger of the two parts has size at least a half.

The Lower Bound: This completes the lower bound that any algorithm requires at least $\Omega(N \log N)$ binary operations to correctly solve the sorting problem.

Example 7.0.2 (Binary Search Returning Index) Consider the problem of searching a sorted list of N elements where the output states the index of the key in the list. Binary search solves the problem with $\log_2(N)$ comparisons. We will now prove a matching lower bound.

The Initial Set of Instances: To follow the same technique that we did for sorting, we need a set of legal input instances, each of which has a unique output. Now, however, there are only N possible outputs. Let the initial set of instances be $S = \{I_j | j \in [1, N]\}$, where I_j is the input instance searching for the key 5 within the list that has the first $j - 1$ elements zero, the j th element 5, and the last $n - j$ elements 10.

The Measure of Progress: Initially, $|S| = N$. As before, S does not decrease by more than a factor of 2 at each iteration. Hence, after t iterations, $|S| \geq N / 2^t$. By setting $T_{\text{lower}}(|I|)$ to be $\log_2(N) - 1$, we know that in the end we have at least two input instances remaining to be our I and I' .

The Lower Bound: The rest of the lower bound is the same, proving that any algorithm requires at least $\Omega(\log N)$ binary operations to correctly solve the problem of searching a sorted list.

You Have to Look at the Data Lower Bounds: The following lower bounds do not really belong in the iterative algorithms part of this book, because in these cases we do not find the instances I and I' iteratively. However, the basic idea is the same. These lower bounds say that at least $N' \leq N$ operations are required on an input of size N , because you must look at at least N' of the input values.

Example 7.0.3 (Parity)

The easiest example is for the problem of computing *parity*. The input consists of n bits, and the output simply states whether the number of ones is even or odd.

The Information-Theoretic Approach Does Not Work: The information-theoretic approach given above allows the model of computation to charge only one time step for any yes–no operation about the input instance, because it counts only the bits of information learned. However, this does not work for the parity problem. If any yes–no operation about the input instance is allowed, then the algorithm can simply ask for the parity. This solves the problem in one time step.

Reading the Input: Suppose, on the other hand, the model of computation charges one time step for reading a single bit of the input. (We could even give any additional operations for free.) Clearly, an algorithm cannot know the parity of the input until it has read all of the bits. This proves the lower bound that any algorithm solving the problem requires at least n time. We will see, however, that there is a bug in this argument.

Example 7.0.4 (Multiplexer)

The *multiplexer* computational problem has two inputs: an n -bit string x , and a $\log_2(n)$ -bit index i , which has the range 1 to n . The output is simply the i th bit of x . As we did for parity, we might give lower bound of n for this problem as follows: If the algorithm does not read a particular bit of x , then it will give the wrong answer when this bit is the required output. This argument is clearly wrong, because the following is a correct algorithm that has running time $\log_2(n) + 1$: It reads i and then learns the answer by reading the i th bit of x .

Dynamic Algorithms: Proving a lower bound based on how many bits need to be read is a little harder, because an algorithm is allowed to change which bits it reads based on what it has read before. Given any single instance, the algorithm might read only $n - 1$ of the bits, but which bit is not read depends on the input instance.

Fixing One Instance and Flipping a Bit: Before we can know what the algorithm A does, we must give it a specific input instance I . We must choose one. Then we determine the set $J \subseteq [1, n]$ of bits of this instance that are critical, meaning for each $j \in J$, if you flip just the j th bit of I but leave the rest of the instance alone, then the answer to the computation problem on this instance changes. We then obtain a lower bound of $n' = |J|$ on the time required to solve the problem as follows. We run the algorithm on I and see which bits of this instance it reads. If it reads $n' = |J|$ bits, then

we are done. Otherwise, there is some bit $j \in J$ that the algorithm does not read on this instance. Because the algorithm does not read it, we can flip this bit of the instance without affecting the answer the algorithm gives. We are not allowed to change any of the bits that the algorithm does read, because not only may this change the answer that it gives, it may also change which bits it reads. We have made sure, however, that the flipping of this single bit changes the answer to the computation question. Hence, on one of the two input instances, the algorithm must give the wrong answer.

Example 7.0.5 (Parity)

We now obtain a formal lower bound of n for the parity problem, as follows. Let I be the all-zero instance. Let $J = [1, n]$ be the set of all bits of the input. For each $j \in J$, changing the j th bit of I changes the answer from even parity to odd parity. Hence, if the algorithm does not read the j th bit when given instance I , it gives the wrong answer either on instance I or on the instance with this bit flipped.

Example 7.0.6 (Multiplexer)

We have obtained a $\log_2(n) + 1$ -time algorithm for solving the multiplexer problem. Now we obtain a matching lower bound. Let I be the instance with $x = 100,000$, i.e., one in its first bit and zero in the rest, and with $j = 1$. Let J consist of the $\log_2(n)$ bits of j and the first bit of x . The output of the multiplexer on I is one, because this is the j th bit of x . But if you flip any bit of j , then a different bit of x is indexed and the answer changes to zero. If you flip the first bit of x from being a one, then the answer also changes. Hence, if the algorithm does not read one of these bits when given instance I , it either gives the wrong answer on instance I or does so on the instance with this bit flipped.

Example 7.0.7 (Binary Search Returning Yes or No)

In [Example 7.0.2](#) we proved a $\log_2(N)$ lower bound for searching a sorted list of N elements. We are to do the same again. The difference now is that if the key is in the list, the problem returns only the output yes, and if not then no.

The Approach

The Information-Theoretic Approach Does Not Work: Again, the information-theoretic approach does not work, because if any yes–no operation on the input instance is allowed, then the algorithm can simply ask whether the key is in the list, solving the problem in one time step.

The Set- J -of-Bits-to-Flip Approach Does Not Work: The initial instance I needs to consist of the key being searched for and some sorted list. Given this, there are not many elements J that can be changed in order to change the output of the searching problem.

Some Combination: Instead, we will use a combination of the two lower bound approaches.

The Initial Set of Instances: As we did when we proved the lower bound for this problem in [Example 7.0.2](#), we consider the input instance I_j , which is to search for the

key 5 within the list with the first $j - 1$ elements zero, the j th element 5, and the last $n - j$ elements 10. Unlike before, however, these instances all have the same outputs: yes. As in the set- J -of-bits-to-flip approach, let I'_j be the same instance, except the j th element is changed from 5 to 6 so that I_j and I'_j have opposite answers. Considering these, let the initial set of instances be $S = \{I_j | j \in [1, n]\} \cup \{I'_j | j \in [1, n]\}$.

The Standard Loop Invariant: As before, the loop invariant states that we have a set S of input instances on which algorithm A 's knowledge and actions for its first t time steps are identical.

Another Loop Invariant: We have additional loop invariants stating that the current structure of S is $S = \{I_j | j \in [j_1, j_2]\} \cup \{I'_j | j \in [j_1, j_2]\}$, where $[j_1, j_2]$ is a subinterval of the sorted list of size. Moreover, $|[j_1, j_2]| \geq N/2^t$.

Maintaining the Loop Invariant: We maintain the loop invariant as follows. Assume that the loop invariant is true for time $t - 1$. Let m be the index of the element read at time t by the algorithm on all inputs instances in S .

$m \notin [j_1, j_2]$: If the algorithm reads an element m before our subrange $[j_1, j_2]$, then for all instances in S , this m th element is zero. Similarly, if m is after, then this element is definitely 10. In either case, the algorithm learns nothing that has not already been fixed. The loop invariant is maintained trivially without changing anything.

$m \in [j_1, j_{mid}]$: Let $[j_1, j_{mid}]$ and $[j_{mid} + 1, j_2]$ split our subrange in half. If m is in the first half $[j_1, j_{mid}]$, then we set our new subinterval to be the second half $[j_{mid} + 1, j_2]$. This narrows our set of instances down to $S = \{I_j | j \in [j_{mid} + 1, j_2]\} \cup \{I'_j | j \in [j_{mid} + 1, j_2]\}$. For all instances in this new S , the m th element is zero. The algorithm reads and learns the value zero and proceeds. The loop invariant is maintained.

$m \in [j_{mid} + 1, j_2]$: If m is in the second half $[j_{mid} + 1, j_2]$, then we set our new subinterval to be the first half, and for all instances in the new S , the m th element is 10.

Ending: The exit condition is then $t = T_{lower}(|I|) = \log_2(N)$. From the loop invariant, when we exit our subinterval $[j_1, j_2]$, its size is at least $N/2^t = 1$. Let $j = j_1 = j_2$. Our set S still contains the two instances I_j and I'_j . By the definition of these instances, the first requires the answer $P(I_j) = \text{yes}$ and the second $P(I'_j) = \text{no}$. From the loop invariant, the computation of A is identical on these instances for the first $T_{lower}(|I|)$ time steps, and hence their outputs must be identical: $A(I_j) = A(I'_j)$. Hence, the computation must give a wrong answer on at least one of them.

The Lower Bound: The rest of the lower bound is the same, proving that any algorithm requires at least $\Omega(\log N)$ binary operations to correctly solve the problem of searching a sorted list.

Current State of the Art in Proving Lower Bounds: Lower bounds are hard to prove, because you must consider *every* algorithm, no matter how strange or complex. After

all, there are examples of algorithms that start out doing very strange things and then in the end magically produce the required output.

Information Theory: The technique used here to prove lower bounds does not consider the amount of work that must get done to solve the problem, but the amount of *information* that must be transferred from the input to the output. The problem with these lower bounds is that they are not bigger than linear with respect to the bit size of the input and the output.

Restricted Model: A common method of proving lower bounds is to consider only algorithms that have a restricted structure. My PhD thesis proved lower bounds on the trade-offs between the time and space needed to check $(s - t)$ -connectivity of a graph in a model that only allows pebbles to slide along edges and jump between each other.

General Model: The theory community is just now managing to prove the first non-linear lower bounds on a general model of computation. This is quite exciting for those of us in the field.

Exercise 7.0.1 How would the lower bound change if a single operation, instead of being only a yes–no question, could be a question with at most r different answers? Here r is some fixed parameter.

Exercise 7.0.2 (See solution in Exercise Solutions.) Recall the Magic Sevens card trick introduced in [Section 4.2](#). Someone selects one of n cards, and the magician must determine what it is by asking questions. Each round, the magician rearranges the cards into rows and asks which of the r rows the card is in. Give an information-theoretic argument to prove a lower bound on the number of rounds, t , that are needed.

Exercise 7.0.3 (See solution in Exercise Solutions.) Suppose that you have n objects that are completely identical except that one is slightly heavier. The problem P is to find the heavier object. You have a scale. A single operation consists of placing any two disjoint sets of the objects the two sides of the scale. If one side is heavier, then the scale tips over. Give matching upper and lower bounds for this problem.

Exercise 7.0.4 Communication complexity: Consider the following problem: Alice has some object from the M objects $\{I_1, \dots, I_M\}$, and she must communicate which object she has to Bob by sending a string of bits. The string sent will be an *identifier* for the object. The goal is to assign each object a unique identifier so that the longest one has as few bits as possible.

Exercise 7.0.5 State and prove a lower bound when instead of bits Alice can send Bob letters from some fixed alphabet Σ .

Exercise 7.0.6 (See solution in Exercise Solutions.) The *AND* computational problem given n bits determines whether at least one of the bits is a one. This is the same as the *game show* problem mentioned in [Chapter 29](#), which requires finding which of the n doors conceals a prize. The way this differs from the parity problem is that the

algorithm can stop as soon as it finds a prize. Give a tight lower bound for this problem. In the lower bound for the parity problem, in which initial instances would input I work? Which ones work for the AND problem?

Exercise 7.0.7 Search a sorted matrix: The input consists of a real number x and a matrix $A[1 \dots n, 1 \dots m]$ of nm real numbers such that each row $A[i, 1 \dots n]$ is sorted and each column $A[1 \dots n, j]$ is sorted. The goal is to find the maximum array entry $A[i, j]$ that is less than or equal to x , or report that all elements of A are larger than x .

1. [Exercise 4.4.1](#) gives an iterative algorithm that accesses $T(n, n) = m + n - 1 = 2n - 1$ entries when $n = m$. Prove a matching lower bound of $T(n, n) = 2n - 1$ for this case. (Start with a lower bound of n if you like.)
2. [Exercise 12.1.3](#) gives a recursive algorithm that accesses $T(n, m) = n \log_2(\frac{m}{n})$ elements when $m \gg n$. Prove a matching lower bound.

Exercise 7.0.8 Consider the problem of determining the smallest element in a max heap. The smallest elements of a max heap must be one of the $\lceil n/2 \rceil$ leaves. (Otherwise, there must be a nonleaf that is smaller than one of its descendants, which means the tree is not a max heap.) Thus, it is sufficient to search all leaves. Prove a lower bound that searching all the leaves is necessary.

8 Key Concepts Summary: Loop Invariants and Iterative Algorithms

8.1 Loop Invariants and Iterative Algorithms

If you are designing, describing, and proving correct an algorithm using loop invariants, the following is the list of the things you must consider. This step order is a nice one for presenting them because it is the order in which they are often logically developed. You might prefer to present them in the order that the computation proceeds. It is a judgment call. Each step should be contained in its own paragraph with the heading given.

Specification: Define the Problem (pre- and postconditions): Likely defined by the problem.

* **Define the Loop Invariant:** It needs to be a picture of what is true every time the algorithm is at the top of the loop. Also specify which of the following it is:

More of the Input: A prefix of the input has been read so far. Pretending that this prefix is the entire input, I have a complete solution. I also have any additional information stored so that I can solve the problem for the entire input without needing to go back and reread the parts of the input that I have already read.

More of the Output: The first of so many of the objects in the required output have been correctly produced.

Narrowing the Search Space: A narrowed search space has been specified, and if the thing being searched for is anywhere, then then it is in this narrowed space.

Shrinking Instance: An instance has been produced that is smaller than the actual instance, and these two instances require the same output. Hence, it is sufficient for me to forget about the actual instance and simply give the output for the new, smaller one.

Fairy Godmother: I have arrived from Mars. Some amount of progress has been made already, and my Fairy Godmother has set the state of the computation to be just what I want so that I feel as comfortable as I could be given the amount of progress that has been made.

*** Step:** Define the step $code_{loop}$ that is executed each iteration and is guaranteed to *maintain the loop invariant* while *making progress*. Though this step, when actually coding, may require a nested loop or other fancy code, here it suffices to give a quick description of what actions take place. (Though you can't say that miracles occur.)

- One common way to decide what actions should be taken in an iteration is to first change the data structure in some way that makes progress. Perhaps you increase i by one, consider the next item, or cut the value of x in half.
- You know that the loop invariant was true when you were at the top of the loop. However, the change you made to make progress messed up the loop invariant being true.
- Now you have to make a change to the data structure that brings it back to the loop invariant, without undoing the progress that you just made. Breathe and move back onto the path.
- On the other hand, sometimes something else messes up the loop invariant. Maybe the monster runs around the river after you. Your initial and main focus then might be to maintain the loop invariant. Afterward, you can give some thought to making progress.

*** Maintaining the Loop Invariant:**

$\langle loop-invariant' \rangle \ \& \ not \ \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant'' \rangle$:

- Imagine that you have just flown in from Mars and are at the top of the loop. All you know about the state of the data structure is that the loop invariant is true and the exit condition is not. (OK, you have not actually defined the exit condition, but you know that you are not done.)
- Let x' be the current values of the variables and x'' after going around the loop again.
- State what you know about x' from the fact that $\langle loop-invariant' \rangle$ holds for these values.
- The step $code_{loop}$ constructs the values x'' from the values x' . State what this tells you about x'' .
- State how it then follows that $\langle loop-invariant'' \rangle$ holds for the values x'' .

*** Exit:** Define the Exit Condition.

Not Postcondition: It would be natural to exit when you are done and hence it is natural to make the exit condition be the same as the postcondition. But this just passes the problem of whether the postcondition will ever be met onto whether your loop will ever exit. Do not do this.

Sufficient Progress: Instead, define a measure of progress and have your exit condition be that sufficient progress has been made. If it is clear that progress is made each iteration, then it will be obvious that the loop will eventually exit. In this case, you can skip the Termination step.

Panic: Another good exit condition is the reason that that you can no longer maintain the loop invariant. Note that defining such an exit condition makes it easier to do the previous step of proving that if the exit condition is not true, then the step taken maintains the loop invariant. You may have to revisit this previous step.

*** Establishing the Loop Invariant:**

$\langle pre-cond \rangle \ \& \ code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$:

Don't work hard. Do the minimum work to make the loop invariant true.

- State what you know about the input instance because of $\langle pre-cond \rangle$.
- State the effect of the code before the loop.
- State how it then follows that $\langle loop-invariant \rangle$ has been established.

*** Obtaining the Postcondition:** $\langle loop-invariant \rangle \ \& \ \langle exit-cond \rangle \ \& \ code_{post-loop} \Rightarrow \langle post-cond \rangle$:

- State what you know because of $\langle loop-invariant \rangle$.
- State what you know because of $\langle exit-cond \rangle$.
- State what you know because of $code_{post-loop}$.
- State how it then follows that $\langle post-cond \rangle$.

Termination: As stated above, if it is obvious that the exit condition will be met, you do not need to do these steps.

- Define measure of progress.
- Initially finite progress is required.
- Prove that at each iteration you make progress.
- Prove that with sufficient progress, the exit condition will be met.

*** Running Time:** Give an upper bound on the number of times that this algorithm iterates. Also estimate the total running time.

8.2 System Invariants

Like *loop invariants*, another extremely useful tool is *system invariants*. They are similar, but have the following differences.

Goal: Loop invariants are used to prove the correctness of an iterative algorithm computing a function from the given input to the required output. In contrast, system invariants are used to maintain the integrity of a system throughout time.

A Function: Given an input meeting a precondition, an iterative algorithm computing a function repeated takes the *step* within the loop in order to compute an output meeting the postcondition.

A System: A system/class/ADT/object has an internal data structure that stores its current state and has a set of operations that can be executed by the user/client (or users) of the system.

Relationship to Loop Invariants:

“Loop” through Time: The system does not have an internal loop like an iterative algorithm that repeatedly executes its step until the exit condition is met. Instead, the system’s “loop” is time. Throughout time, the external user of the system specifies a sequence of the system’s operations. The goal of the system is to perform each of these operations when it arises in time.

More of the Input Loop Invariant: A system invariant can be viewed as more of the input loop invariant, if the input $I = \langle R_1, R_2, \dots, R_n \rangle$ is the sequence of requests that will be made by the client of the system. Each R_i specifies which operation needs to be executed and on what inputs. Then the system invariant says that a prefix of these requests has been handled and the state of the system meets the system invariants.

Invariants: System invariants might be *public* or *private*.

Public Invariants: Public system invariants are specified in the description of the abstract data type or system so that the user can better understand how it works.

Bank Account: The system invariant of a bank account is that the amount is not negative.

Stack: The system invariant of a stack is that it contains an ordered list of objects that cannot be changed except at the front end via the explicit push and pop operations.

Drug Allocation: A system that keeps track of which drugs have been allocated by various doctors for various conditions to various patients would have the invariant that a patient is not simultaneously prescribed a set of drugs that interact poorly with each other.

Private Invariants: A particular implementation of an ADT or system likely will have its own invariants that are not revealed to its users. They are used to ensure that the algorithm for each operation works correctly and efficiently.

Balanced Binary Search Tree: A balanced binary search tree is an implementation of the ADT *Set* that allows the client to store a set of objects. These objects can be searched for, can be added, and can be deleted. The private invariants will ensure that the objects are stored at the nodes of a binary tree such that for each node all the values in its left subtree are at most that at this node and all in the right are at least that. To ensure $O(\log n)$ search times, an additional invariant might impose some tree-balancing requirement.

Drug Allocation: A private invariant of an implementation of a drug system might be that each patient’s record contains a list of the patient’s doctors, the patient’s ailments, and the drugs they are currently taking, as well as those previously taken.

Maintaining the System Invariants: No matter what sequence of operations occurs on an ADT/system, their invariants must be maintained.

$\langle \text{system-invariant}' \rangle \ \& \ \text{code}_{op} \Rightarrow \langle \text{system-invariant}'' \rangle$:

- Imagine that you have just flown in from Mars. All you know about the state of the system is that its invariants are true. Then some operation gets executed.
- Let x' be the current values of the variables and x'' after executing the operation.
- State what you know about x' from the fact that $\langle \text{system-invariant}' \rangle$ holds for these values.
- The operation code_{op} constructs the values x'' from the values x' . State what this tells you about x'' .
- State how it then follows that $\langle \text{system-invariant}'' \rangle$ holds for the values x'' .

Operation Specification: Each operation allowed on the ADT/system will have an input, an output, and a change to the state of the system. It will be specified using pre- and postconditions.

As a Function: The search & delete operation on a binary search tree can be thought of as executing the following function whose input is $\langle \text{key}, \text{tree} \rangle$ and whose output is $\langle \text{object}, \text{tree}' \rangle$. Given key meets the precondition of a valid key and tree is the current state of the ADT meeting all public and private system invariants, then object as specified by the postcondition will be the object in the data structure tree with key key (assuming it exists) and tree' is the same as tree except this object has been deleted. As always, the operation must guarantee that tree' still meets all the system requirements.

As a System Operation: The only difference when this operation is thought of as a system operation instead of as a function is that tree and tree' are no longer inputs and outputs, but are global variables specifying the internal state of the system. These variables are hidden from the client so they cannot be accessed or changed except through these operations.

Proof of Correctness:

$\langle \text{system-invariant}' \rangle \ \& \ \langle \text{pre-cond}_{op} \rangle \ \& \ \text{code}_{op} \Rightarrow \langle \text{post-cond}_{op} \rangle \ \& \ \langle \text{system-invariant}'' \rangle$

Obtaining the Postcondition: Unlike an iterative algorithm, a system may continue forever or might get terminated by one of its operations. But as seen above, after each operation is executed, the prover must prove that the system invariants ensure that the postconditions of the operation are fulfilled correctly.

Initializing the System: We have seen how an iterative algorithm must establish the loop invariants from the preconditions on its inputs before the first iteration of the loop. Similarly, each system/ADT will have operations that create a new instance of the system/ADT. Such operations will have inputs that must meet given preconditions. They must establish the initial state of the system so that all the system requirements are true.

$\langle \text{pre-cond}_{\text{InitializingOp}} \rangle \ \& \ \text{code}_{\text{InitializingOp}} \Rightarrow \langle \text{system-invariant} \rangle$

9 Additional Exercises: Part I

I.1. Loop Invariants

- (a) What is a loop invariant?
- (b) What is the formal statement that an algorithm is **correct** for a problem?
- (c) What are the formal mathematical things involving loop invariants that must be proved to prove if your program exits, then it obtains the postcondition?
- (d) What is the loop invariant for the GCD alg?
- (e) What is the loop invariant for the binary search?
- (f) Explain what the *More of the input* loop invariant is and explain how that is used.

I.2. **Steps:** If you are designing, describing, and proving correct an algorithm using loop invariants, list EVERYTHING that you must include? (Be sure to include all of these things in the next question.)

I.3. **Selection Sort:** Describe the algorithm Selection Sort using the loop invariant method. Give the key steps.

I.4. **RadixSort:** RadixSort sorts d -digit numbers:

```
loop i = 1 to d
    stable sort cards with respect to the  $i^{\text{th}}$  least significant digit.
end loop
```

At every iteration, the following assertion is true: “The input is stable-sorted with respect to the i^{th} least-significant digit.” Is this a good loop invariant? Why or why not?

I.5. **Tiling Chess Board:** You are given a $2n \times 2n$ chess board. You have many tiles, each of which can cover two adjacent squares. Your goal is to place non-overlapping tiles on the board to cover each of the tiles except for the top-left corner and the bottom-right corner. Prove that this is impossible. To do this give a loop invariant that is general enough to work for any algorithm that places tiles. Hint: Chess boards color the squares black and white.

I.6. **Longest Contiguous Increasing Subsequence (LCIS):** The input consists of a sequence $A[1 \dots n]$ of integers and we want to find a longest contiguous subsequence $A[k_1 \dots k_2]$ such that the elements are *strictly* increasing. For example, the optimal solution for $[5, 3, 1, 3, 7, 9, 8]$ is $[1, 3, 7]$.

- (a) Provide a tight lower bound on the running time for this problem. Prove that your answer is indeed a tight lower bound.
- (b) Specify an appropriate loop invariant, measure of progress, and exit condition for an iterative solution to this problem. No explanations are necessary.

- (c) Provide concise pseudo-code for an algorithm $\text{LCIS}(A, n)$ that returns the indices k_1, k_2 of the LCIS of $A[1, \dots, n]$, and uses the loop invariant, measure of progress, and exit condition you specified above. Assume $n \geq 1$.
- (d) Provide an informal argument for how your code is able to maintain the loop invariant. A formal proof is not required.
- (e) What are the other key steps in proving your algorithm correct? List and provide a concise proof for each.

I.7. Iterative 3 in a Row in an Infinite Line: The game board consists of a single line of squares that is infinitely long in both directions. Each iteration your opponent places a white piece and then you place a black piece. If a player gets three in a row, they win. Your primary goal is to stop them from doing this. If the game goes on forever, then this is considered a tie.

Your game strategy will be to maintain the following loop invariants:

After we each have played the same number of times the following are true.

- (a) Every contiguous block of whites has a black on its left end.
- (b) Every contiguous block of whites of length two or more also has a black on its right end.
- (c) There are no contiguous blocks of white of length three or more.

(i.e., A block consisting of one white has a black to its left, a block with two whites has a black on both ends, and there are no blocks of length three or more.)

The exit condition will be that someone has won.

The postcondition will be that you win.

Your task is to follow all the loop invariant steps outlined in the steps and then to prove that either the game goes on forever as a tie or you win.

Suppose your opponent has just placed a white. There are nine cases depending on whether there is a black, white, or blank on the left of this newly placed white and whether there is a black, white, or blank on the right. To help you, I group these nine into the five cases indicated in the following five lines. Then I group these further into three cases indicated by the black dots. In each of these five cases, state where you will place your black in response and in each of these three cases, prove that this maintains each of the three loop invariants.

- There is a blank to the left of where your opponent placed their white.
There is a black to the left.
- There is a white to the left and a blank to the right.
There is a white to the left and a black to the right.
- There is a white to the left and a white to the right.

Is it possible to have an algorithm that guarantees you win even if you go first?

I.8. Iterative n in a Row in an Infinite Line

The game board is an infinitely long line of squares. In a normal round, you will place a white and then your opponent will place a black. The goal is to build long contiguous blocks of your color. There is no algorithm that allows you to build a

block longer than two. We will now develop an algorithm that builds arbitrarily long contiguous white blocks. To help, you will be given the advantage that every tenth round you get to place a white but your opponent must skip their turn. (The regular expression is $((WB)^9W)^*$.)

It seemed at first to me that even if you got 100 times as many turns, that you could not build a contiguous block longer than 200, i.e., you put a 100, they cap one end by placing a black there, you place a second 100 on the other end, and your opponent that end with a black. If you did manage to build a really long block, it would take you a while. Then in just two moves they can destroy it by capping both ends. Surprisingly, you can build arbitrarily long lines. (Funny, though, you can't just build a block that gets longer and longer arbitrarily. You must have the length you want to build in mind when you start.)

To make the algorithm easier, let's assume that the opponent is not trying to build a block of blacks. Their only goal is to prevent you from building a long contiguous block of white by capping the ends of any block you are working on with a black. All blocks that we start will be far enough away from each other that they will never grow long enough to interact with each other. We will be cautious. If the opponent caps one of the ends of one of our blocks or even puts a black anywhere near that block, then we will not wait for them to cap the second end but will abandon that block completely. Hence, all we are concerned about is how many blocks we have that have no blacks near them and how long are these blocks are. The adversary in each of their turns, being as greedy as they can be, will simply cap the first end of our currently longest such block. Given this, we will keep our blocks as close to being the same size as possible.

Input: The input to this problem is an integer n (say, a billion).

Precondition: The board is empty.

Postcondition: You have produced a continuous block of whites of length n .

Iterations: This will be an iterative algorithm. You will have "iterations," for $i = 0, 1, \dots, n$. Each such iteration will contain as many rounds of turns as you need to make progress and maintain the loop invariant.

Loop Invariant: After i "iterations," you have constructed many *special* blocks.

Length: Each such special block consists of i whites in a row.

Isolation: Each is far away from any black and from each other.

Number: The number of such special blocks in 10^{n-i} .

In addition to these special blocks the board will contain many white blocks that are abandoned because they have a black on one end.

- Complete the steps in completing the description/proof of this algorithm.
- Also compute the total number N of whites that you place.

I.9. Multiplying Using Adding: My son and I wrote a JAVA compiler for his grade 10 project. We needed the following. Suppose you want to multiply $X \times Y$ two n bit positive integers X and Y . The challenge is that the only operations you have are adding, subtracting, and tests like $\leq$. The standard high school algorithm seems to require looking at the bits of X and Y and hence can't obviously be implemented here. The Ethiopian algorithm requires dividing by two, so it can't be implemented either. A few years after developing this algorithm, I noticed that it is in fact identical to this high school multiplication algorithm, but I don't want to tell you this because I don't want you thinking about the algorithm as a whole. This will only frighten you. All you need to do is to take it one step at a time. I only want you to establish and maintain the loop invariant and use it to get the postcondition. To help, I will give you the loop invariants, the measure of progress, and the exit condition. I also want you to explain what the time complexity of this algorithm is, i.e., the number of iterations and the total number of bit operations as a function of the size of the input.

We have values $i, x, a, u[]$, and $v[]$ such that

Useful Arrays: Before the main loop, I will set up two arrays for you with the following values. Then they will not be changed.

LI0': $u[j] = 2^j$, (for all j until $u[j] > X$).

LI0'': $v[j] = u[j] \times Y$. Note $v[j] - u[j] \times Y = 0$.

My Code: For completion, I include my code, but it is not necessary for you to understand it.

```

u[0] = 1
v[0] = Y
j = 0
while( u[j] ≤ X )
    u[j + 1] = u[j] + u[j]
    v[j + 1] = v[j] + v[j]
    j = j + 1
end while

```

For Main Loop: These are the loop invariants for you to deal with.

LI0: Neither X nor Y change.

LI1: $X \times Y = x \times Y + a$ (i.e., Shrinking Instance).

LI2: $x \geq 0$.

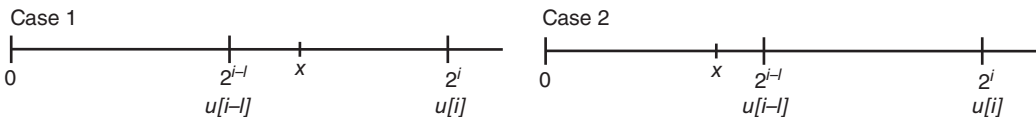


Fig. 9.1

LI3: $x < 2^i = u[i]$ (below are two cases).

Measure of progress: i . It decreases by 1 each iteration.

Exit Condition: $i = 0$

Add the steps to complete the description of the algorithm.

- I.10. **The Carnival Coin Game:** You are running a game booth at your local village carnival. In your game you lay out an array of coins on a table. You and your customer alternate turns. During a turn, the player must either pick up one coin or pick up two. The player forced to pick up the last coin loses. Your task is to design a strategy (loop invariant algorithm) that ensures that you always win.

[Actually, we first present a simplified problem for $r = 2$. You can skip the $r = 2$ question and go straight to solving the problem for general r .]

- You get to choose the number n of coins to start with. Part of your strategy is to specify a precondition on n . (Don't answer $n = 1$, as you need to allow for n to be chosen from an infinite set of possible values.)
- You graciously allow (require) your customer to go first. Each round (step) starts with the customer picking up either 1 or 2 coins. Your strategy must specify how you in turn respond to what they do. Let i denote the number of coins currently left. As a hint, here are a few possibilities for you to consider.
 - (a) You pick up the same number as your customer, i.e., if they pick up one, then you pick up one too. Same for two.
 - (b) You pick up the opposite number as they do, i.e., if they pick up one, then you pick up two and visa versa.
 - (c) Or maybe you have some fancier strategy depending on the number i of coins left.
- The postcondition is "I win because the number of coins left is $i = 1$ and it is the customer's turn." Many students are tempted to have part of the exit condition be this same " $i = 1$ ". I, however, never like having the post- and the exit conditions being the same, because then instead of needing to prove the postcondition is eventually true, you need to prove the exit condition is eventually true. Instead, I like an exit condition that is **CLEARLY** reached if enough progress is made. If the number left is, say, $i = 10$, then you don't exit. What do you do if i is 2 or 3? Maybe we should exit the loop and move to the code after the loop. So let's make the exit condition be something like $i \leq 5$ or $i \leq 3$.
- Another issue is that we also need to **PROVE** that the number of coins i does not become negative. But let's be bolder and add to your loop invariant that $i \geq 1$. Take care when maintaining the loop invariant that this condition is also maintained.
- That was just a warmup. Let's redo the problem. In the previous game, each player is allowed only to take one or two coins. In the r version of the game, each turn the player must take one, two, three, four, . . . , or r of the coins. The

problem above has $r = 2$. I had a version of this game in a plastic board with marbles in which $r = 3$. It had a mechanical AI that played the winning strategy. If you answer it just for $r = 2$, you will be given part marks. For full marks, you need to write the loop invariant algorithm for some arbitrary value of r .

- I.11. **Square Root:** Compute the square root using binary search with the extra twist that the size of the search space is a power of two so that your running time is faster. With this alone, you should be able to develop the algorithm by quickly completing the steps.

Warning: Don't let what I write below panic you. Some of the points are answers that I give you for free. Some are quick hints. Skim them quickly.

Pre- and Postconditions: The precondition is that the inputs are a positive real value p and a value ϵ like $0.00001 > 0$. The postcondition is that a real value q is returned that is a close enough to the square root of p so that $|q - \sqrt{p}| \leq \epsilon$.

Loop Invariant: We have a narrowed search space $[p_a, p_b]$ that sandwiches the input p and one $[q_a, q_b]$ that sandwiches its square root q .

i.e., $p_a = q_a^2 \leq p \leq q_b^2 = p_b$.

The size $r = q_b - q_a$ of this second range is an ever decreasing power of 2.

i.e., $r = 2^{n'-t}$ where t is the number of iterations.

Establishing the Loop Invariant: We establish the loop invariant as quickly as we can. I love zero, so set $q_a = 0$, $p_a = q_a^2 = 0$. Then, $r = q_b - 0$ needs to be a power of two, so set them to be $2^{n'}$ for some integer n' . This gives that $p_b = q_b^2 = 2^{2n'}$. We need that $p_b \geq p$. If n' was a real value then we could set $2^{2n'} = p$ and solve giving $n' = \frac{1}{2} \log_2 p$. In order to make it a bigger integer, we round up to $n' = \lceil \frac{1}{2} \log_2 p \rceil$. (Increasing n' by at most an additive one increases $p_b = 2^{2\lceil \frac{1}{2} \log_2 p \rceil}$ above p by at most a multiplicative of four.)

Obtaining the Postcondition: The measure of progress is the size of the remaining search space $r = q_b - q_a$. This is decreased by a factor of 2 each iteration. The exit condition is when it is at most ϵ . Setting our answer q to be either q_a or q_b then gives us $|q - \sqrt{p}| \leq q_b - q_a \leq \epsilon$.

Finding the Middle: Classically, the middle of the search space is computed as $q_{mid} = \frac{1}{2}(q_a + q_b)$.

Because $r = q_b - q_a$, $q_{mid} = \frac{1}{2}(q_a + q_b) = q_a + \frac{1}{2}(q_b - q_a) = q_a + \frac{1}{2}r$.

I recommend that you compute q_{mid} as $q_a + \frac{1}{2}r$.

Faster with Power of Two: A twist is that we want to use the following trick to do the algorithm faster. The straightforward algorithm for multiplying two n bit/digit numbers together takes $O(n^2)$ time because each digit in the one number is multiplied by each in the other. If, however, one of the numbers is a power of 2, i.e., 2^i , then in binary it has a single 1 bit followed by some i zeros. (Similarly consider the case where i is negative.) Multiplying in binary then involves shifting the decimal point of the other number by i places. This only takes $O(n)$ time.

- (a) Prove that the loop invariant can be quickly maintained by squaring q_{mid} .
- (b) What then is the total running time as a function of the size of the input of the following three algorithms?
 - i. Find the square root of an integer p by trying q to be $1, 2, 3, \dots, p$. How many values of q need to be tried as a function of the size of the input?
 - ii. A better thing would be to do a binary search for the correct value of q . How many values of q need to be tried as a function of the size of the input? Suppose that for each such value of q , you square it. The straightforward algorithm for multiplying takes $O(n^2)$ time. What then is the total running time?

Hint: For fewer marks, you could assume that p and ϵ are integers.

Hint: If the inputs p and ϵ are real numbers, then the size n of the input formally is the number of bits/digits needed to write it down. For example, the size of 45.67 and of 0.0001 could both be considered size $n = 4$. Let's set n to be $\log p + \log \frac{1}{\epsilon}$ because we only care about real numbers with at most $\log p$ bits above the decimal and at most $\log \frac{1}{\epsilon}$ below it.

- iii. How does binary search time improve when each iteration we multiply by a power of two r ?

I.12 **Cube Root:** Instead of computing the square root, suppose your goal is to find the cube root using binary search. I do NOT want the entire algorithm here. The following is a key part.

```

loop
  <loop-invariant>: You have a value  $q_t$  and know  $q_{(3,t)} = (q_t)^3$ 
  exit when ??
   $r = 2^x$  for some strange integer  $x$ 
   $q_{t+1} = q_t + r$ 
   $q_{(3,t+1)} = (q_{t+1})^3$ 
end loop

```

Your goal is to explain how to accomplish the same task while using only $O(n)$ time per iteration.

Hint: My answer simply redoes the code and gives a two-sentence explanation.

Hint: $(q + r)^3 = q^3 + 3rq^2 + 3r^2q + r^3$.

Question: In one sentence explain whether the same trick can be used to compute the fifth root.

I.13. **Simple Sum**

algorithm *Sum*(a, b)

<pre-cond>: b is a positive integer. a can be any real.

<post-cond>: $a + b$ is returned

begin

$x = a$

$y = b$

```

 $r = 2^{\lfloor \log_2 y \rfloor}$ 
(the biggest power of 2 not bigger than  $y$ .)
loop
   $\langle \text{loop-invariant} \rangle : ???$ 
  exit when  $y = 0$ 
  if(  $y \geq r$  )
     $\langle \text{case 1} \rangle$ 
     $x = x + r$ 
     $y = y - r$ 
  else
     $\langle \text{case 2} \rangle$ 
     $r = r/2$ 
  end if
end loop
return( $x$ )

```

- (a) Use loop invariants to prove that if the loop exits, then the following code returns the sum $a + b$.

Hint: Give minimal thought to the value of r other than that it is always at least one.

- (b) Consider the following additional loop invariants.

LI2: y is an integer and $y \geq 0$.

LI3: r is an integer power of 2, namely, one of $1, 2, 4, \dots$ (Note that r is never $\frac{1}{2}$.)

LI4: $r > \frac{1}{2}y$.

Hint: I will establish these LI for you.

LI2: The precondition gives that $y = b$ is a positive integer.

LI3: For $y \geq 1$, $\lfloor \log_2 y \rfloor$ is a non-negative integer and hence $r = 2^{\lfloor \log_2 y \rfloor}$ is a power of 2,

i.e., one of $1, 2, 4, \dots$

For example, if $y = 10 = 1010_2$, then $r = 8 = 1000_2$.

LI4: $r = 2^{\lfloor \log_2 y \rfloor} > 2^{(\log_2 y) - 1} = \frac{1}{2}y$.

Hint: We have already obtained the postcondition.

Your job is to prove that these three LIs are maintained.

- (c) The loop does not exit until y is exactly zero. Think of generic code with a value of y and two cases within a loop. List four potential ways that the code could run forever without ever having $y = 0$. For each way, prove that this way does not occur. This should be sufficient to prove that eventually the program does in fact halt.
- (d) Prove that the case 1 in the loop cannot happen twice in a row.
- (e) Argue what the bigOh of the number of iterations is as a function of the size of the input.

- (f) Change the line of code $r = 2^{\lceil \log_2 y \rceil}$ to $r = 1$ and delete the line $r = r/2$. Argue what the bigOh of the number of iterations now is as a function of the size of the input.

I.14. **2024:** This problem is a simplified version of a new game called 2048. Maybe you have seen it.

Screen: The screen contains a set of boxes, each with a number in it.

Iteration Code: We partition time so that each “iteration” one of the following two things occurs.

Arrivals: A new box appear whose value is two. We call these 2-boxes.

Clicking/Merging: The player clicks and two boxes with the *same* value collapse into one box whose value is the sum of the previous two boxes.

Adversary: An adversary decides when a 2-box will arrive and which pair of boxes will merge. However, to avoid the process getting stuck, they must have new 2-boxes arrive if there are no pairs that can merge.

Unimportant Details: Note that time is not really broken into “iterations.” At any point, the adversary can send in a 2-box and at any point the player can click. But the proof is easier if we break time up so only one of these things happens per iteration. For example, if the screen contains two 3-boxes and two 7-boxes, then the adversary has three choices: Slip in a 2-box, or let the player click and merge the two 3-boxes into a 6-box, or let the player click and merge the two 7-boxes into a 14-box. If the screen only contains one pair that has the same value, then when the player clicks these two will merge. (For people who know the actual game, we are not worrying about rows and columns.) If the screen only contains no pairs that have the same value, then the player is not allowed to click. This is why we force the adversary to send in a 2-box. If there is still no equal pair, then again the adversary will have to send in a 2-box. At this point there is an equal pair and the player can click.

The Game: The game is defined by:

Precondition: Assume the game begins with an empty board.

Exit Condition: The game stops when the player has clicked M times. (You must determine the smallest value for this that you can.)

Postcondition: The player wins because there is (or was) a box with the value 2048.

Variables:

Consider some point in time (the algorithm is at the top of the loop).

- Let n be the number of 2-boxes that have arrived so far.

- Let m be the number of times that the player has clicked so far, i.e., the number of that a pair of boxes with the same value have merged into one. Note $m \leq M$.
- Let b be the number of boxes currently on the screen.
- Let s be the sum of the values in the boxes currently on the screen.

Loop Invariants:

- Give a loop invariant that states some very simple mathematical property that each number currently on the screen is assured to have. (Add to the loop invariant that if a value appears, then every smaller value with this property must have appeared at some point. This is used to ensure that we don't skip our target 2048.)
- Give a loop invariant that gives the value of b as a simple function of n and m .
- Give a loop invariant that gives the value of s as a simple function of n and m .
- Hint: To formulate these, first determine how the different events affect the value of boxes, s and b .

Give a way that the adversary can proceed so that it goes on forever without a 2048-valued box ever being created.

I.15. Let us add an extra constraint.

Max # of Boxes: The adversary will ensure that there are never more than 20 boxes on the screen, i.e., at all times $0 \leq b \leq 20$.

Your task is to do the **key steps** of the loop invariant iterative algorithm for this game.

Hint: Use the fact that *Max Box Value* $\geq$ *Avg Box Value*.

Does your bound M on the number of clicks needed make you wonder whether anyone has ever won this game? Have you heard of it? Have you won it?

HINT: DO NOT PANIC! It really is not hard. Just follow the steps.

I.16. **Counting in Editor:** This problem arises in my life periodically. You have a blank document *main* open in your favorite editor, e.g., Word or Emacs. The goal is to produce in this document all of the d digit integers in order. For example, if $d = 9$, then the required output is the following:

```
000000000
000000001
000000002
...
999999999
```

The obvious algorithm is the brute force one that simply types the $d \cdot 10^d$ required characters, but you are much lazier than that. The Java code is easy, “loop $i=0 \dots 10^d-1$ print(i),” but installing the Java compiler would also be too much work so you plan on sticking to using only the editor. However, within your editor all you can do is (write a macros which will) type characters, move the cursor, cut, copy, paste, change from one document to another, and global search and replace. You do **not** have access to variables, if, print, add, $\dots$. Because a macro cannot print the key ‘6’ one iteration and ‘7’ next, we will store the digits 0,1,2, $\dots$, 9 in a file to be used as needed. Your algorithm will have a “loop,” but by this I only mean steps that you personally repeat over and over or, even better, a keyboard macro that gets executed a given number of times.

Describe your algorithm using the loop invariant method. Actually, the algorithm will have two nested loops. The outer loop will be executed d times (think of Radix Sort) and the inner loop will be executed 10 times (think of “more of the output loop invariant”). We have not covered about iterative algorithms with two loops, so let me give you the basic format.

begin routine

$\langle pre-cond \rangle$: Main is empty, d is number of digits needed per number
 $i = 0\%$. This is just conceptual, because in fact we have no variables.

Set of three files.

loop

$\langle inner\ loop's\ pre-cond \rangle = \langle outer\ loop's\ loop-invariant_i \rangle$

exit when $i = d$

$code_{pre\ inner\ loop}$

loop

$\langle inner\ loop-invariant \rangle$: See below.

exit when $\langle exit-cond \rangle$

$code_{inner\ loop's\ step}$

end loop

$code_{post\ inner\ loop}$

$\langle inner\ loop's\ post-cond \rangle = \langle outer\ loop's\ loop-invariant_{i+1} \rangle$

end loop

$code_{post\ outer\ loop}$

$\langle post-cond \rangle$: Main contains the 10^d integers with d digits

end routine.

To help you, I will give you the loop invariant for the inner loop.

Loop Invariant: You are day-dreaming, so the only thing you know is the current state of the editor. Let i denote the number of times the outer loop has executed so far and let j denote the number of times the inner loop has executed within the $i+1^{st}$ iteration of the outer loop. You have three documents (other than the main) open. When $i = 4$ and $j = 6$, their contents are as follows.

Contents of Document D1	Contents of Document D2	Contents of Document D3
0	#0000	#00000
1	#0001	#00001
2	#0002	#00002
3	#0003	#00003
4	...	...
5	#9998	#59998
6 ■	#9999	#59999
7		
8		
9		

D1: Document D1 contains on separate lines the characters 0, 1, 2, 3, ..., 9. The cursor is located on the digit j .

D2: Document D2 contains the correct output as if $d = i$, i.e., it has the i digit integers in order. The only difference is each of these integers is preceded by the character “#.”

D3: Document D3 contains the correct output as if $d = i + 1$, i.e., it has the $i + 1$ digit integers preceded by “#.” However, it does not have all of them. It only has those starting with the first j characters, namely, 0, 1, ..., $j - 1$.

- (a) For us, the inner loop begins when the computation is at the top of the i^{th} iteration of the outer loop. Hence, we have that the precondition for the inner loop is the loop invariant of the outer loop for this iteration. Similarly, the inner loop ends when the computation is at the bottom of this same iteration of the outer loop, which is of course the top of its next iteration. Hence, the postcondition for the inner loop is the loop invariant of the $i + 1^{\text{st}}$ iteration of outer loop. (See the above code.) Quickly state what this condition is, namely,

$$\begin{aligned} \langle \text{inner loop's pre-cond} \rangle &= \langle \text{outer loop's loop-invariant}_i \rangle \text{ and} \\ \langle \text{inner loop's post-cond} \rangle &= \langle \text{outer loop's loop-invariant}_{i+1} \rangle. \end{aligned}$$

- (b) You are to complete the inner loop and prove that it works correctly using the key steps of the loop invariant method. Hint: Remember the “more of the output” technique.
- (c) Describe the key steps of the loop invariant method for the outer loop of the algorithm. Hint: Remember Radix Sort.
- (d) Running Time: The input consists of the single integer d and the output consists of the 10^d integers with d characters each.
- If by time we mean the number of operations that the computer behind the editor must do, then what is the running time of your algorithm?
 - If the work of the computer is free and by time we mean the number of keys that you must type, then what is the running time of your algorithm? Compare this to the previous.
 - Macros: I love the editor Emacs because it lets me build very powerful macros. Macros just gives a name to a block of the editor commands:

type, move the cursor, cut, copy, paste, change documents, and global search and replace. Then you can give Emacs the name of a macro and an integer and it will execute this macro this number of times.

Many times I have needed to produce the numbers between 0 and $10^d - 1$. I have done this by quickly writing a macro M1 that executes the steps within the inner loop. Then I write a macro M2 that executes the steps within the outer loop, which in this case simply executes M1 10 times. Finally, I establish the loop invariant of the outer loop and execute M2 d times. If there are no bugs, this produces the desired output.

If the work of the computer is free and by time we mean the number of keys that you must type, then what is the time required to write and run these macros? Compare this to the previous.

- (e) Suppose instead of wanting numbers base 10, I want to produce all the d digit integers base b . What changes in the running of the algorithm and in the running times?

I.17. **Gaussian Elimination:** In high school, you would have learned the algorithm *Gaussian Elimination* for solving systems of linear equations and for inverting a matrix. Here we are to think of this algorithm as an iterative algorithm with a loop invariant. (Don't panic. No linear algebra or memory of the algorithm is needed.) The input is an $n \times n$ -matrix M that is invertible and the output is its inverse N , i.e., the $n \times n$ -matrix such that $N \times M = I$. Here I is the identity matrix, i.e., that for which $I \times M = M$. It would be weird, but the statement $M = I \times M$ could be added to the precondition. As a huge hint, the loop invariant is that the algorithm maintains matrices A and B for which $A = B \times M$. (I recall drawing these two matrices next to each other with a vertical line between them.) What I am looking for is really easy. Hence, if you are doing something hard, you are doing it wrong.

- (a) Start by finding simple steps that establish the loop invariant. Give and prove the required formal statement.
- (b) What is needed to turn the loop invariant into the postcondition? Give and prove the required formal statement.
- (c) The action in the next iteration of the algorithm will do one of the following two operations:
 - i. Choose two rows $i, i' \in [n]$ of A and two constants c_1 and c_2 . Replace the i'^{th} of A with c_1 times its i^{th} row plus c_2 times its i'^{th} . Perform this same row operation on B .
 - ii. Or swap two rows in A and swap the same two in B .

Formally state (but do not prove) what needs to be proved in order to prove that this operation maintains the loop invariant.

- (d) You are not required to prove the last statement. But if you really want to, the following are hints.

The matrix equation $A = B \times M$ really consists of n^2 separate linear equations, i.e., for each pair $\langle i, j \rangle \in [n]^2$ of row and column indexes, we have

the equation that $A_{(i,j)} = \sum_{k \in [n]} B_{(i,k)} \times M_{(k,j)}$. Think of $M_{(k,j)}$ as being the variable $x_{(k,j)}$ and $A_{(i,j)}$ and $B_{(i,k)}$ being the coefficients.

Recall that if you have two true equations, say, $5 = 3x_1 + 8x_2$ and $7 = 4x_1 + 3x_2$ then one can get another true equation by taking a linear equation of the first two, namely, $4 \times [5] - 3 \times [7] = 4 \times [3x_1 + 8x_2] - 3 \times [4x_1 + 3x_2]$ becoming $-1 = 0x_1 + 23x_2$.

- (e) Assuming this computation ever exits, it will give the correct answer. We prove that this does happen by proving that progress is made. What is your measure of progress? In a half dozen sentences, sketch how you ensure that progress is made.

If you are feeling more bold, state a loop invariant giving a picture of the order in which this progress is made and show how you can maintain this property.

- (f) According to your definition of progress, how much progress needs to be made? How much progress is made each iteration? How many such iterations are needed? What is the computation time for each? What is the total time of Gaussian Elimination?

I.18. Insertion Sort on a Turing Machine: We will design a TM that sorts an array of some n values, each in the range $\{1, \dots, d\}$. Each TM cell can contain one value from $\{-\infty, 1, \dots, d, \text{blank}\}$.

algorithm InsertionSort(Tape of values)

⟨pre-cond⟩:

The TM tape contains the n input values in the cells “indexed” by $\{1, \dots, n\}$.

Cell 0 contains $-\infty$ (so that it is already sorted).

Cell $n+1$ contains a blank (to indicate the end).

The head is on cell 1.

⟨post-cond⟩:

The tape still contains the same n input values in the same n cells except that the values been sorted.

The head is on the blank at cell $n+1$.

begin

01: allocate memory for a new variable p

02: let $p = -\infty$

03: loop

⟨Measure-of-Progress 1⟩:

The tape still contains the same n input values except that some have been sorted. Denote by r the largest index so that values in the cells indexed by $\{1, \dots, r\}$ are sorted.

<loop-invariant 1>:

Denote by i the cell index of the head. The values are sorted up to but maybe not including cell i , i.e., $i \in \{1, \dots, r+1\}$. Also the TM has a variable p in which it remembers the value in the previous cell, i.e., the value in cell $i-1$.

<Goal 1>:

To make progress by increasing r while maintaining this loop invariant.

All the discussions within this loop are relative to the tape values as they are here now.

04: loop

<loop-invariant 2>:

The tape contents has not changed.

The loop invariant is LI1.

The change is that the head may have moved right.

<Goal 2>:

To make progress by increasing i while maintaining this loop invariant.

05: Let c denote the value at head

06: if(c is a blank) then

07: exit and return from entire sort program

08: else if($p \leq c$) then

09: $p = c$

10: move the head right

11: else

12: goto line 15

13: end if

14: end loop

15: forget the value in p and deallocate the memory

16: allocate memory for a new variable q

17: $q =$ value at head

18: loop

<loop-invariant 3>:

The index r has been found, i.e., the first cell such that $Tape[r] > Tape[r+1]$.

⟨loop-invariant 3⟩: The too small value that had been in cell $r+1$ has been saved in variable q .

The head is on cell i , for some $i \in \{1, \dots, r+1\}$.

For each value $j \in \{i, \dots, r\}$, the value that had been in cell j has been moved to cell $j+1$.

For example, the last sorted value, i.e., that in cell r , was shifted to cell $r+1$ where the smaller value had been and the value in cell i , the current cell, has already been shifted over by one.

Note that we don't need the value in the current cell and hence this space can be used for something else.

Also, all of these shifted values are greater than the too small value that had been in cell $r+1$ and that is now saved in q .

⟨Goal 3⟩:

To make progress by decreasing i while maintaining this loop invariant.

```

19:      write  $q$  into cell at head
20:      forget the value in  $q$  and deallocate the memory
21:      move head left
22:      reallocate memory for variable  $p$ 
23:      let  $p$  = value at head
24:      move head right
25:      let  $q$  denote the value at head
           (ie we know that this value that had been in  $q$ , but we are
           not storing it)
26:      if(  $p \leq q$  ) then goto line 3
27:      allocate memory for  $q$ 
28:      let  $q$  = value at head
29:      write  $p$  into cell
30:      deallocate the memory for  $p$ 
31:      move head left
32:  end loop
33: end loop

```

Your tasks:

- (a) Prove all the loop invariant steps for both the inner and outer loops.
- (b) In 1936, Alan Turing designed a *Turing Machine* as the first general model of computation. It has a finite set of states that it transitions between and an infinite memory single head tape. Read [Chapter 26](#) Assertions – TM and DFA – Forgetting for further explanation. Translate this code into TM

states and transitions. Marks will be lost if you fail to express it in a neat and compact way.

Hint: Generally, the state changes one per head movement, though the odd “stay” command is fine.

- (c) How many states does your TM have?
 (d) What is the bigOh running time of your TM (in terms of TM steps)?

Does it depend on d ? Why or why not?

You may know that the advertised running time of Insertion Sort is $O(n^2)$.

This TM seems to do a lot in one time step, but also must move its head back and forth a lot. So, is its running time more or less than $O(n^2)$?

Analyze and explain.

- I.19. **Pegs Winning Strategy:** Alice and Bob play a game with some k pegs in a line of some n holes. At the beginning of the game, the pegs occupy the k leftmost holes. A legal move consists of moving a single peg to any vacant hole that is further to the right. The players alternate moves, with Alice playing first. The game ends when the pegs are in the k rightmost holes, so whoever is next to play cannot move and therefore loses. The goal is to determine (when possible) a winning strategy for Alice. More specifically, I will partition the set of all board positions into two sets: Losing and Winning. Your task is to prove that the current player has a winning strategy if and only if the current board position is in the Winning subset.

Hint: Most of your answers will be *proof by picture*.

Losing Board Positions: Whenever I say “losing board position,” I mean the following definition (with your additions added).

- Partition the n holes into consecutive pairs.
- The pegs can be paired so that each such pair occupies one such pair of holes.
- X = peg in hole, 0 = hole with no peg.
- e.g., $00\ XX\ 00\ 00\ XX\ XX\ 00\ XX\ 00\ 00\ 00\ 00$ where X = peg in hole, 0 = hole with no peg.

Hint: In this case, both n and k are even. We will consider other cases later.

Loop Invariant: Currently it is Bob’s turn and the board position is *losing* (as defined above).

Hint: If our definition of a losing position is correct, then Alice would have winning strategy from here.

Maintaining the Loop Invariant: Start by stating the logic statement needed for maintaining the loop invariant.

Assume for now that both n and k are even.

To be concrete, suppose it is currently Bob’s turn and the current board position is:

– e.g., 00 XX 00 00 XX XX 00 XX 00 00 00 00.

For each possible move Bob might make, specify which move Alice responds with to move the board position back to a losing one (according to the above definition).

Hint: Give as many examples as needed to give the idea.

Hint: Give a few sentences.

Other Cases: When both n and k are even, we gave the example:

– e.g., 00 XX 00 00 XX XX 00 XX 00 00 00 00.

Now consider all four cases, i.e., n even and odd and k even and odd.

For each, draw a picture like that above to demonstrate the losing board positions.

Hint: Here are four ways that you may choose how to pair consecutive holes: 00 00 00 00 or 00 00 00 0 or 0 00 00 00 or 0 00 00 00 0.

Hint: If the number of pegs k is odd, I recommend putting the unpaired peg where it effectively does not exist for the game.

By the definition of the game, when the pegs occupy the k rightmost holes, the current player loses. Hence, this is a losing board position. For each of the four cases, show how your definition includes this example.

When n is odd, there is an unpaired hole, and when k is odd, there is an unpaired peg. For each case, show that this does not affect how the loop invariant is maintained.

Postcondition: Bob has just lost, i.e., it is his turn and the pegs occupy the k rightmost holes.

Exit Conditions: The obvious exit condition is that the pegs occupy the k rightmost holes.

We need to make the exit condition so that it is obvious that the loop will eventually exit.

Clearly, if each move the player moves a peg to the right, then eventually this exit condition will be met.

But this makes me uncomfortable because the loop invariant states that it is Bob's turn. Hence, this loop invariant and this exit condition together obviously give the postcondition that Bob loses. This is too easy.

The problem is that the loop invariant assumes that at the top of the loop it is Bob's turn. We do not know whether the above exit conditions will ever be met when at the top of the loop.

Instead, make the exit condition the following:

It is Bob's turn and either:

- the pegs occupy the k right most holes and hence Bob loses or
- the pegs are such that Bob can make one move so that the pegs occupy the k rightmost holes and Alice loses.

Obtaining the Postcondition: In the last question, you were asked to demonstrate that having the pegs occupy the k rightmost holes is a losing position according to your definition. Let's wave our hands and state that it follows that if

the current peg position is one move away from occupying the k rightmost holes, then it is not such a losing position.

Complete the proof that the postcondition is obtained.

Precondition: Its beginning of the game, i.e., it is Alice's turn and the pegs occupy the k leftmost holes.

Establishing the Loop Invariant: Consider each of the four cases, i.e., n even and odd, k even and odd.

For each, draw pictures like those above to either:

- demonstrate that having the pegs occupy the k leftmost holes is itself a losing position and hence Bob has a winning strategy.
- demonstrate which single peg Alice moves to go from the pegs occupying the k leftmost holes to Bob being in a losing position, ie show how Alice has a winning strategy.

Determine the running time.

- I.20. **Producing k :** Suppose you start with the number 1 and it costs you one to add two numbers together that you already have. (Numbers can be reused.) Once you have produced a number, you can use it many times. Given an integer k as input, design an algorithm to produce k as cheaply as possible. What does it cost? The space requirement is how many numbers you need to keep at any given time. Try to get an algorithm with minimum space. Hint 1: Think of the input k as an $n = \log_2(k)$ bit binary string. (Consider reading it forward and backward). Hint 2: Try “More of the Input,” “More of the Output,” and “Narrowing the Search Space.” One of these three works fairly easily. (10 bonus points if you can do it in time $T(k) = n + \Theta(\frac{n}{\log n})$.)
- I.21. **Engineers' Binary Search Wire:** Imagine a wire that over time carries a 0/1 signal. A front-end processor has sampled it at a rate of 1 GHz (i.e., every nanosecond) and has given you an array A of length n . $A[i] = 0$ if there is a 0 on the wire at time i nsecs and $A[i] = 1$ if there is a 1 on the wire at that time. A transition (or “edge”) is defined to be a place in time where the signal transitions from one value to another. This may be of “positive polarity,” i.e., from 0 to 1 or of “negative polarity,” meaning from 1 to 0. The time of the transition is the average of these two indexes. For example, if $A[7] = 0$ and $A[8] = 1$, then we say that we “time-stamp” this transition with the time $t = 7.5$ nsecs. Our goal, given such an array A is to find the time-stamp of one such transition (if it exists). If there are many such transitions in A , then we can return any one of them.
- (a) Give a lower bound that any algorithm that solves this problem, on the worst case input, requires n time to find the answer. Imagine this as a game in which some adversary gives you an algorithm and you must find an input array A for which this algorithm queries A n times before finding a transition. Your solution will describe how you will use the actions of the algorithm to produce the input. Be clear how you handle the following circular argument. You don't know what input instance to give the algorithm until you know

what the algorithm does. But you don't know what the algorithm does until you give it an input instance and run it.

- (b) Now let us add the additional constraint that at the beginning and end of the time interval, the wire has a different signal, i.e., $A[1] \neq A[n]$. Note that this assumption ensures that there is at least one transition, though it could still have many transitions. Design a sublinear time recursive algorithm that given the 0/1 array A returns the time t of a transition. Be very clear about your pre- and post-conditions and that you are giving your friend a smaller instance that meets the precondition. Though you must be clear what the algorithm is, actual code is not needed. Give and solve the recurrence relation for its running time.

I.22. Engineers' Shifted Streams: You receive two streams of data, $A[1], A[2], A[3], \dots, A[n]$ and $B[1], B[2], B[3], \dots, B[n]$. Once a number arrives, you cannot request it again. You must output a stream of data containing the values $A[1]+B[1], A[2]+B[2], A[3]+B[3], \dots, A[n]+B[n]$. Once you output it, you do not need to store it. There are three things that make this more challenging.

- 1) The B data streams are shifted in time by some amount s .
- 2) You have only a limited amount of memory, $M[0], M[1], M[2], M[3], \dots, M[m]$, of which you want to use as little as possible.
- 3) You have only a limited amount of time between data numbers arriving.

Give **all** the required steps to describe this LI algorithm. (Even if you do not know how to do a step for this algorithm, minimally state the step.)

As a hint, your loop invariant should include:

- (a) The numbers $A[1], A[2], A[3], \dots, A[t]$ and the numbers $B[1], B[2], B[3], \dots, B[t-s]$ have arrived.
- (b) Which values have been outputted?
- (c) Which values do you want to be remembering in memory? (Use as little memory as possible.)
- (d) In which memory cell do you save each such value? (You want to move things as little as possible.)

I.23. Amortized Analysis Binary Counter: Suppose we have a binary counter such that the cost to increment the counter is equal to the number of bits that need to be flipped. For example, incrementing from $100,111,111_2$ to $101,000,000_2$ costs 7. Consider a sequence of n increments increasing the counter from zero to n . Some of these increments, like that from $101,111,010_2$ to $101,111,011_2$, costs only one. Others, like that from $111, \dots, 111_2 = 2^{\lceil \log_2 n \rceil} - 1 \approx n$ to $1000, \dots, 000_2$, costs $O(\log n)$. This one is the worst case. Given that we generally focus on worst cases, one would say that operations cost $O(\log n)$. It might be more fair, however, to consider the average case. If a sequence of n operations requires a total of at most $T(n)$ time, then we say that each of the operations has *amortized cost* $\frac{T(n)}{n}$.

- (a) Suppose the counter begins at zero and we increment it n times. Show that the amortized cost per increment is just $O(1)$.

Hint: Let $T_{x,j} = 1$ if incrementing value x requires flipping the j^{th} bit. Note $\sum_{x=1}^n \sum_j T_{x,j} = \sum_j \sum_{x=1}^n T_{x,j}$.

- (b) Suppose that we want to be able to both increment and decrement the counter. Starting with a counter of zero, give a sequence of n of operations where each is either an increment or a decrement operation that gives the highest amortized cost per operation (within Θ). (Don't have the counter go negative.) Compute this amortized cost.

- (c) To fix the problem from part (b), consider the following redundant ternary number system. A number is represented by a sequence of trits $X = x_{n-1}, \dots, x_1 x_0$, where each x_i is 0, +1, or -1. The value of the number is $\sum_{i=0}^{n-1} x_i \cdot 2^i$. For example, $X = 101_2$ is 5 because $4 + 1 = 5$. However, $X = (1)(1)(-1)_2$ is also 5 because $4 + 2 - 1 = 5$. Also $X = (1)(-1)(0)(-1)(1)(1)(1) = 64 - 32 - 8 + 4 + 2 + 1 = 31$.

The process of incrementing a ternary number is analogous to that operation on binary numbers. One is added to the low-order trit. If the result is 2, then it is changed to 0, and a carry is propagated to the next trit. This process is repeated until no carry results. For example, $X = (1)(-1)(0)(-1)(1)(1)(1) = 31$ increments to $X + 1 = (1)(-1)(0)(0)(0)(0)(0) = 64 - 32 = 32$. Note this is a change of $8 - 4 - 2 - 1 = 1$. Decrementing a number is similar. One is subtracted from the low-order trit. If it becomes -2 then it is replaced by 0, and a borrow is propagated. For example, $X + 1 = (1)(-1)(0)(0)(0)(0)(0)$ decrements to $X = (1)(-1)(0)(0)(0)(0)(-1) = 64 - 32 - 1 = 31$. Note that this increment followed by a decrement resulted in a different representation of the number 31 than the one we started with.

The cost of an increment or a decrement is the number of trits that change in the process. Our goal is to prove that for any sequence n of operations where each is either an increment or decrement operation, starting with a counter of zero, the amortized cost per operation is at most $O(1)$. Let $T(t)$ be the total number of trit changes that have occurred during the first t operations. Let $Z(t)$ be the number of non-zero trits in our counter after the t operations. For example, if the resulting counter is $X = (1)(-1)(0)(-1)(0)(1)(1)$, then $Z(t) = 5$. We will define $P(t) = T(t) + Z(t)$ to be our *measure of progress* or *potential function* at time t .

Bound the maximum amount that $P(t)$ can change in any one increment or decrement operation.

- (d) Use induction (or loop invariants) to bound $P(t)$. In doing so, prove that for any sequence n of operations where each is either an increment or decrement operations; starting with a counter of zero, the amortized cost per operation is at most $O(1)$.

- I.24. **Lower Bounds Leaves in Tree:** Prove using strong induction on M that every binary tree with M leaves must have a leaf with depth of at least $\log_2 M$.
- I.25. **BST: Lower Bound:** Within the decision tree model, derive a lower bound on the BST construction problem, i.e., given a set of n keys in no particular order, construct a BST that holds those n keys.

10 Partial Solutions to Additional Exercises: Part I

I.1. Loop Invariants

- (a) A picture of what must be true at the top of the loop
- (b) If for every legal input instance, the required output is produced. If the input instance does not meet the preconditions, then all bets are off. Formally, we express this as

$$\langle pre-cond \rangle \ \& \ code_{alg} \Rightarrow \langle post-cond \rangle .$$

This correctness is only with respect to this specifications.

- (c)
 - i. $\langle pre-cond \rangle \ \& \ code_{pre-loop} \Rightarrow \langle loop-invariant \rangle .$
 - ii. $\langle loop-invariant' \rangle \ \& \ not \ \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant'' \rangle .$
 - iii. $\langle loop-invariant \rangle \ \& \ \langle exit-cond \rangle \ \& \ code_{post-loop} \Rightarrow \langle post-cond \rangle .$
- (d) On input $\langle a, b \rangle$, the algorithm maintains two variables x and y such that $GCD(x, y) = GCD(a, b)$.
- (e) If the thing being searched for is anywhere, then it is in this narrowed sublist.
- (f) The *More of the Input* loop invariant is that we have a solution for each of the first i objects. It is used so that we only have to make a small change when the next object is considered.

I.2. Steps

- The LI
- A measure of progress.
- The main step within the loop.
- Exit condition.
- Proof that if not exited:
 - * LI is maintained.
 - * progress is made.
- How the LI is established in the beginning.
- Proof that with sufficient progress, the exit condition will be met.
- Proof that with the LI and the exit condition together, the problem can be solved.

I.3. Selection Sort

If the input for selections sort is presented as an array of values, then sorting can happen in place. The first k entries of the array store the sorted sublist, while the remaining entries store the set of values that are on the side. Finding the smallest value from $A[k + 1], \dots, A[n]$ simply involves scanning the list for it. Once it is

found, moving it to the end of the sorted list involves only swapping it with the value at $A[k + 1]$. The fact that the value $A[k + 1]$ is moved to an arbitrary place in the right-hand side of the array is not a problem, because these values are considered to be an unsorted set anyway. The running time is computed as follows. We must select n times. Selecting from a sublist of size i takes $\Theta(i)$ time. Hence, the total time is $\Theta(n + (n-1) + \dots + 2 + 1) = \Theta(n^2)$.

I.4. RadixSort: Sorts d -digit Numbers:

No, this is not a good loop invariant. It is true but not enough to know that the input is stable-sorted with respect to JUST the i^{th} least-significant digit. It is a more of the input type of loop invariant. If you consider each number to be just its least-significant i digits, then the list is sorted stably.

I.5. Tiling Chess Board

Precond: The board is empty.

Postcond: All but the corner two white tiles are covered.

Define Loop Invariant: The number of blacks and white tiles covered is equal.

Establishing the Loop Invariant: $\langle pre-cond \rangle \ \& \ code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$: By $\langle pre-cond \rangle$, there are zero white and zero blacks covered. Hence, these numbers are the same.

Step: Place a tile. This covers one white and one black square.

Maintaining the Loop Invariant: $\langle loop-invariant' \rangle \ \& \ not \ \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant'' \rangle$: By $\langle loop-invariant' \rangle$, at the beginning of the iteration the number of white and black tiles covered are equal. The step covers one more white and one more black. Hence, after the iteration the numbers are still equal.

Exit: Exit whenever you like.

Ending: Obtaining that the postcondition is NEVER met.

i.e., $\langle loop-invariant \rangle \Rightarrow \neg \langle post-cond \rangle$.

Prove the contrapositive,

i.e., $\langle post-cond \rangle \Rightarrow \neg \langle loop-invariant \rangle$:

Assume that the postcondition is true. It states that all but the corner two white tiles are covered. However, the board has an equal number of black and white tiles. Hence, the number of white and black tiles covered are not equal. Hence, the LI is not true.

I.6. Longest Contiguous Increasing Subsequence (LCIS)

- (a) The problem takes at least $\Omega(n)$ time because the input needs to be read. Let A be an arbitrary algorithm. Give it the input consisting of all zero. If it does not read some integer, change this integer to a 1 (if it is the first integer, change it to -1). It answers the same longest length in both cases; however, in the first

the longest has length one and in the second two. Therefore, it is wrong in at least one of these cases.

- (b) **LI:** We maintain an overall optimal LCIS found so far and the LCIS that we are currently working on.

$A[k_1 \dots k_2]$ is an LCIS of $A[1 \dots i]$ and $A[p \dots i]$ is the LCIS of $A[1 \dots i]$ that ends at i .

Note the first LI is “More of the Input,” namely, if we consider the prefix $A[1 \dots i]$ to be the entire input, then we have the solution.

Most people miss the second one, but it is needed if the LI is to be maintained.

Measure of Progress: the number i of elements read. Many give that i increases. This is progress, but not a measure.

Exit Condition: When all the elements have been read because $i = n$.

algorithm $[k_1, k_2] = \text{LCIS}(A, n)$

<pre-cond>: $A[1 \dots n]$ is an array of integers

<post-cond>: $A[k_1 \dots k_2]$ is an LCIS of $A[1 \dots n]$

begin

$k_1 = k_2 = p = i = 1$

loop

<loop-invariant>: $A[k_1 \dots k_2]$ is an LCIS of $A[1 \dots i]$ and

$A[p \dots i]$ is the LCIS of $A[1 \dots i]$ that ends at i

$i = i + 1$

if($A[i-1] \geq A[i]$) then

$p = i$

end if

if $(q - p) > (k_2 - k_1)$

$k_1 = p$

$k_2 = q$

end if

end if

end algorithm

- (c) $\langle \text{loop-invariant}' \rangle \ \& \ \text{not} \ \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$

Suppose at the beginning of an iteration, the LI is true for the current values, i.e., $A[k'_1 \dots k'_2]$ is an LCIS of $A[1 \dots i']$ and $A[p' \dots i']$ is the LCIS of $A[1 \dots i']$ that ends at i' .

The first block of code either extends the LCIS that we are currently working on from $A[p' \dots i']$ to $A[p' \dots i'+1]$ or shrinks it to contain only the one element $A[i'+1]$ depending on whether or not $A[i'+1]$ is bigger than $A[i']$. This is achieved by moving its end point from i' to $i'' = i'+1$ and either leaving its begging at $p'' = p'$ or moving it to $p'' = i''$. Either way, the LI is maintained that $A[p'' \dots i'']$ is the LCIS of $A[1 \dots i'']$ that ends at i'' .

The second block of code updates the overall LCIS found so far simply by moving it to the current one, if the current one has grown to be larger. Either way, the LI is maintained that $A[k''_1 \dots k''_2]$ is an LCIS of $A[1 \dots i'']$

- (d) **Establishing the LI:** $\langle pre-cond \rangle \ \& \ code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$.

On entry, $k_1 = k_2 = p = i = 1$. Thus, the first part of the loop invariant requires that $A[1]$ be an LCIS of $A[1]$, which is trivially true. The second part of the loop invariant requires that $A[1]$ is the LCIS of $A[1]$ that ends at location 1, which is also trivially true.

Making Progress: i increments by 1 on each iteration. Program exits when $i = n$.

Establishing Postcondition: $\langle loop-invariant \rangle \ \& \ \langle exit-cond \rangle \ \& \ code_{post-loop} \Rightarrow \langle post-cond \rangle$. When $i = n$, the first LI gives that $A[k_1 \dots k_2]$ is an LCIS of $A[1 \dots n]$, which is the postcondition.

I.7. Iterative 3 in a Row in an Infinite Line

Establishing the Loop Invariant: $\langle pre-cond \rangle \ \& \ code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$

The precondition gives that the board is empty. A very important but silly fact is if there are no elephants in the room, then the statement “Every elephant in the room is pink” is a true statement. Similarly, if there are no contiguous blocks of whites, then L1 “Every contiguous block of whites has a black on its left end” is trivially true. Similarly L2, L3, which states that there are no contiguous blocks of white of length three or more, is clearly true when the board is empty.

Responding Move and Maintaining the Loop Invariant: $\langle loop-invariant' \rangle \ \& \ not \ \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant'' \rangle$.

When they place a white, I move as follows.

- If there is a blank to the left, I go there.
 - If there is a black to the left, I go somewhere else trying to win.
 - * We made sure that this block of whites has a black on its left end. Hence, we maintained LI1.
 - * Because there is not a white to the left of where they went, they did not extend a white block further to the right. Neither did they extend a white block further to the left because LI1 ensures that at the beginning of their turn every white block has a black on its left end. Hence, their white does not make a white block longer and hence maintains LI2 and LI3.
- If there is a white to the left and a blank to the right, I go in the blank.
 - If there is a white to the left and a black to the right, I go somewhere else trying to win.
 - * By LI1, this block of whites already has a black on the left end so LI1 is maintained.
 - * We made sure that this block of whites has a black on its right end. Hence, we maintained LI2.
 - * The opponent could not have made a block of two (or more) whites longer because by LI2 such a block would have had a black on both ends. Hence, LI3 is maintained.

- It is impossible that there is a white to the left and a white to the right.
 - * LI1 ensures that at the beginning of the opponents turn, there is never a blank to the left of a white. Hence, there could not have been such a blank space for him to go into.

Obtaining the Postcondition: $\langle \text{loop-invariant} \rangle \quad \& \quad \langle \text{exit-cond} \rangle \quad \& \quad \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$:

By the loop invariant L3, we know that the opponent has not won.

By the exit condition, someone has won.

Hence, it must be us.

Either the Game Goes on Forever as a Tie or You Win: We don't have a measure of progress, because the game is likely to go on forever as a tie. But we did prove that if the game does exit, you win.

Going First: This same thing proves that there is no algorithm that guarantees you win even if you go first.

I.8. Iterative n in a Row in an Infinite Line

Establishing the Loop Invariant: $\langle \text{pre-cond} \rangle \quad \& \quad \text{code}_{\text{pre-loop}} \Rightarrow \langle \text{loop-invariant} \rangle$

The precondition gives that the board is empty and perhaps the value n . To establish the loop invariant for $i = 0$, we must produce 10^n blocks of whites of length zero far away from each other and far away from any black. This requires placing $N_0 = 0$ whites. It only requires deciding where the blocks will be.

Maintaining the Loop Invariant: $\langle \text{loop-invariant}' \rangle \quad \& \quad \text{not } \langle \text{exit-cond} \rangle \quad \& \quad \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}'' \rangle$.

Suppose the LI is true for $i-1$ "iterations," i.e., the board contains 10^{n-i+1} blocks of whites each of length $i-1$ far away from each other and far away from any black. Our goal is to make these blocks one longer. For each of our next 10^{n-i+1} turns, we simply place a white next to one of the blocks of these length $i-1$, making it a block of length i . Our opponent gets nine moves for each of our ten. Hence, during these 10^{n-i+1} turns of ours, they get $0.9 \times 10^{n-i+1}$ turns. During each of these moves, they can, if they want, "destroy" one of our blocks of length i by placing a black near it. But whatever they do, they cannot destroy more than $0.9 \times 10^{n-i+1}$ of them. As required by the loop invariant, this leaves $10^{n-i+1} - 0.9 \times 10^{n-i+1} = 10^{n-i}$ blocks of length i . During this i^{th} iteration, we placed $N_i = 10^{n-i+1}$ white stones.

Obtaining the Postcondition: $\langle \text{loop-invariant} \rangle \quad \& \quad \langle \text{exit-cond} \rangle \quad \& \quad \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$:

The exit condition is that $i = n$. By the loop invariant, there is $10^{n-n} = 1$ contiguous white block of length n .

Measure of Progress: Clearly the measure of progress is the value i , it increases each iteration and after n such iteration the exit condition is obtained.

Running Time N : The total number of whites that you place is $N = \sum_{i=1}^n N_i = \sum_{i=1}^n 10^{n-i+1} \leq 2 \cdot 10^n$.

I.9. Multiplying Using Adding

Establishing the Loop Invariant: $\langle pre-cond \rangle \ \& \ code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$:

$code_{pre-loop}$:

$i = j$
 $x = X$
 $a = 0$

LI1: $x \times Y + a = X \times Y + 0 = X \times Y$.

LI2: $x = X \geq 0$ by precondition.

LI3: $x = X < u[j] = u[i]$ by exit condition of while loop that constructs $u[j]$.

Steps

```

loop
   $\langle loop-invariant \rangle$ 
  exit when  $i = 0$ 
   $i = i - 1$ ;
  if( $u[i] \leq x$ )
     $x = x - u[i]$ 
     $a = a + v[i]$ 
  end if
end loop

```

Maintaining the Loop Invariant: $\langle loop-invariant' \rangle \ \& \ not \ \langle exit-cond \rangle \ \& \ code_{loop} \Rightarrow \langle loop-invariant'' \rangle$:

Let x' be the value of x when we are at the top of the loop and let x'' be that after going around the loop one more time. (Same for i and a .)

There are two cases:

if($u[i''] \leq x'$): The code in the step gives $i'' = i' - 1$, $x'' = x' - u[i'']$, and $a'' = a' + v[i'']$.

Maintain LI1: $x'' \times Y + a'' = (x' - u[i'']) \times Y + (a' + v[i'']) = x' \times Y - u[i''] \times Y + a' + v[i''] = (x' \times Y + a') + (v[i''] - u[i''] \times Y) = (X' \times Y \text{ by LI1}) + (0 \text{ by LI0}) = X \times Y$.

Maintain LI2: By the if statement $u[i''] \leq x'$ and hence $x'' = x' - u[i''] \geq 0$.

Maintain LI3: $x'' = x' - u[i''] = x' - 2^{i'-1} < 2^{i'} - 2^{i'-1}$ (by LI3) = $2^{i'-1} = 2^{i''} = u[i'']$.

else The code in the step gives $i'' = i' - 1$, $x'' = x'$, and $a'' = a'$

Maintain LI1&2: Trivial.

Maintain LI3: By the else of the if statement, $u[i''] > x'$.

Obtaining the Postcondition: $\langle \text{loop-invariant} \rangle \quad \& \quad \langle \text{exit-cond} \rangle \quad \& \quad \text{code}_{\text{post-loop}} \Rightarrow \langle \text{post-cond} \rangle$:

By exit condition, $i = 0$.

By LI2&3, $0 \leq x < u[i] = 1$, and hence $x = 0$.

By LI1, $X \times Y = x \times Y + a = 0 \times Y + a = a$.

It follows that the postcondition is met simply by outputting a .

Time Complexity: The “size” of the input is $n = \log_2 X + \log_2 Y$, which is the number of bits to write down X and Y . The number of iterations is the initial value of i , which is set so that $X = x < u[i] = 2^i$, giving $i = \lceil \log_2 X \rceil \leq O(n)$. Each iteration requires add/subtract operations that take $O(n)$ bit operations. Hence, this algorithm will require $O(n^2)$ bit operations.

The High School Algorithm: See [Section 1.7](#) for a comparison of this algorithm with the high school one.

Part II

Recursion

11 Abstractions, Techniques, and Theory

Iterative algorithms start at the beginning and take one step at a time toward the final destination. Another technique used in many algorithms is to slice the given task into a number of disjointed pieces, solve each of these separately, and then combine these answers into an answer for the original task. This is the *divide-and-conquer* method. When the subtasks are different, it leads to different subroutines. When they are instances of the original problem, it leads to recursive algorithms.

People often find recursive algorithms very difficult. To understand them, it is important to have a good, solid understanding of the theory and techniques presented in this chapter.

11.1 Thinking about Recursion

There are a number of ways to view a recursive algorithm. Though the resulting algorithm is the same, having the different paradigms at your disposal can be helpful.

Code: Code is useful for implementing an algorithm on a computer. It is precise and succinct. However, code is prone to bugs, is language-dependent, and often lacks higher levels of intuition.

Stack of Stack Frames: Recursive algorithms are executed using a stack of stack frames. See [Section 11.6](#). Though this should be understood, tracing out such an execution is painful.

Tree of Stack Frames: This is a useful way of viewing the entire computation at once. It is particularly useful when computing the running time of the algorithm. However, the structure of the computation tree may be very complex and difficult to understand all at once.

Friends, on Strong Induction: The easiest method is to focus on one step at a time. Suppose that someone gives you an instance of the computational problem. You solve it as follows. If it is sufficiently small, solve it yourself. Otherwise, you have a number of friends to help you. You construct for each friend an instance of the same computational problem that is *smaller* than your own. We refer to these as *subinstances*. Your friends magically provide you with the solutions to these. You then combine these *subsolutions* into a solution for your original instance.

I refer to this as the *friends* level of abstraction. If you prefer, you can call it the *strong induction* level of abstraction and use the word “recursion” instead of “friend.” Either way, the key is that you concern yourself only with your task. Do not worry about how your friends solve the subinstances that you assigned them. Similarly, do not worry about whoever gave you your instance and what they do with your answer. Leave these things up to them. Trust your friends.

Use It: I strongly recommend using this method when designing, understanding, and describing a recursive algorithm.

Faith in the Method: As with the loop invariant method, you do not want to be rethinking the issue of whether or not you should steal every time you walk into a store. It is better to have some general principles with which to work. You do not want to be rethinking the issue of whether or not you believe in recursion every time you consider a hard algorithm. Understanding the algorithm itself will be hard enough. While reading this chapter, you should once and for all come to understand and believe how the following steps are sufficient to describing a recursive algorithm. Doing this can be difficult. It requires a whole new way of looking at algorithms. However, at least for now, adopt this as something that you believe in.

11.2 Looking Forward vs. Backward

Circular Argument? Recursion involves designing an algorithm by using it as if it already exists. At first, this looks paradoxical. Suppose, for example, the key to the house that you want to get into is in that same house. If you could get in, you could get the key. Then you could open the door, so that you could get in. This is a circular argument. It is not a legal recursive program because the subinstance is not smaller.

One Problem and a Row of Instances: Consider a row of houses. Each house is bigger than the next. Your task is to get into the biggest one. You are locked out of all the houses. The key to each house is locked in the house of the next smaller size. The recursive problem consists in getting into any specified house. Each house in the row is a separate instance of this problem.

The Algorithm: The smallest house is small enough that one can use brute force to get in. For example, one could simply lift off the roof. Once in this house, we can get the key to the next house, which is then easily opened. Within this house, we can get the key to the house after that, and so on. Eventually, we are in the largest house as required.

Focus on One Step: Though this algorithm is quite simple to understand, more complex algorithms are harder to understand all at once. Instead, we focus on one step at a time. Here, one step consists in opening house i . We ask a friend to open house $i - 1$, out of which we take the key with which we open house i . We do not worry about how to open house $i - 1$.

Working Forward vs. Backward: An iterative algorithm works forward. It knows about house $i - 1$. It uses a loop invariant to show that this house has been opened. It searches this house and learns that the key within it is that for house i . Because of this, it decides that house i would be a good one to go to next.

A recursion algorithm works backward. It knows about house i . It wants to get it open. It determines that the key for house i is contained in house $i - 1$. Hence, opening house $i - 1$ is a subtask that needs to be accomplished.

There are two advantages of recursive algorithms over iterative ones. The first is that sometimes it is easier to work backward than forward. The second is that a recursive algorithm is allowed to have more than one subtask to be solved. This forms a tree of houses to open instead of a row of houses.

Do Not Trace: When designing a recursive algorithm, it is tempting to trace out the entire computation. “I must open house n , so I must open house $n - 1, \dots$. The smallest house I rip the roof off. I get the key for house 1 and open it. I get the key for house 2 and open it. $\dots$ I get the key for house n and open it.” Such an explanation is complicated and unnecessary.

Solving Only Your Instance: An important quality of any leader is knowing how to delegate. Your job is to open house i . Delegate to a friend the task of opening house $i - 1$. Trust them, and leave the responsibility to them.

11.3 With a Little Help from Your Friends

The following are the steps to follow when developing a recursive algorithm within the friends level of abstraction.

Specifications: Carefully write the specifications for the problem.

Preconditions: The preconditions state any assumptions that must be true about the input instance for the algorithm to operate correctly.

Postconditions: The postconditions are statements about the output that must be true when the algorithm returns.

This step is even more important for recursive algorithms than for other algorithms, because there must be tight agreement between what is expected from you in terms of pre- and postconditions and what is expected from your friends.

Size: Devise a measure of the *size* of each instance. This measure can be anything you like and corresponds to the measure of progress within the loop invariant level of abstraction.

General Input: Consider a large and general instance of the problem.

Magic: Assume that by magic a friend is able to provide the solution to any instance of your problem as long as the instance is strictly smaller than the

current instance (according to your measure of size). More specifically, if the instance that you give the friend meets the stated preconditions, then their solution will meet the stated postconditions. Do *not*, however, expect your friend to accomplish more than this. (In reality, the friend is simply a mirror image of yourself.)

Subinstances: From the original instance, construct one or more *subinstances*, which are smaller instances of the same problem. Be sure that the preconditions are met for these smaller instances. Do not refer to these as “subproblems.” The problem does not change, just the input instance to the problem.

Subsolutions: Ask your friend to (recursively) provide solutions for each of these subinstances. We refer to these as *subsolutions* even though it is not the solution, but the instance, that is smaller.

Solution: Combine these subsolutions into a solution for the original instance.

Generalizing the Problem: Sometimes a subinstance you would like your friend to solve is not a legal instance according to the preconditions. In such a case, start over, redefining the preconditions in order to allow such instances. Note, however, that now you too must be able to handle these extra instances. Similarly, the solution provided by your friend may not provide enough information about the subinstance for you to be able to solve the original problem. In such a case, start over, redefining the postcondition by increasing the amount of information that your friend provides. Again, you must now also provide this extra information. See [Section 13.3](#).

Natural Pre- and Postconditions: On the other hand, have the more generalized problem still be a natural problem. Do not attempt to pass it lots of extra information about your instance. For the very first call (or stack frame) of the computation to pass a value through the chain of recursive calls is a type of global-variable “cheat.” It also makes it look like you are micromanaging your friends. Similarly, a stack frame (friend) should not know what level of recursion it is on.

Both the Pre- and the Postconditions Act as Loop Invariants: The *loop invariant* in an iterative algorithm states what is maintained as the control gets passed from iteration to iteration. It provides a picture of what you want to be true in the middle of this computation. With recursion, however, there are two directions. The *precondition* states what you want to be true halfway down the recursion tree. The *postcondition* states what you want to be true halfway back up the recursion tree.

Minimizing the Number of Cases: You must ensure that the algorithm that you develop works for *every* valid input instance. To achieve this, the algorithm will often require many separate pieces of code to handle inputs of different types. Ideally, the algorithm developed has as few such cases as possible. One way to help you minimize the number of cases needed is as follows. Initially, consider an instance that is as large and as general as possible. If there are a number of different types of instances, choose

one whose type is as general as possible. Design an algorithm that works for this instance. Afterward, if there is another type of instance that you have not yet considered, consider a general instance of this type. Before designing a separate algorithm for this new instance, try executing your existing algorithm on it. You may be surprised to find that it works. If, on the other hand, it fails to work for this instance, then repeat the above steps to develop a separate algorithm for this case. You may need to repeat this process a number of times.

For example, suppose that the input consists of a binary tree. You may well find that the algorithm designed for a tree with a full left child and a full right child also works for a tree with a missing child and even for a child consisting of only a single node. The only remaining case may be the empty tree.

Base Cases: When all the remaining unsolved instances are sufficiently small, solve them in a brute force way.

Running Time: Use a recurrence relation or a tree of stack frames to estimate the running time.

A Link to the Techniques for Iterative Algorithms: The techniques that often arise in iterative algorithms also arise in recursive algorithms, though sometimes in a slightly different form.

More of the Input: When the input includes n objects, this technique for iterative algorithms extends (for $i = 1, \dots, n - 1$) a solution for the first $i - 1$ objects into a solution for the first i . This same technique also can be used for recursive algorithms. Your friend provides you a solution for the first $n - 1$ objects in your instance, and then you extend this to a solution to your entire instance. This iterative algorithm and this recursive algorithm would be two implementations of the same algorithm. The recursion is more interesting when one friend can provide you a solution for the first $\lfloor \frac{n}{2} \rfloor$ objects in your instance, another friend can provide a solution for the next $\lceil \frac{n}{2} \rceil$ objects, and you combine them into a solution for the whole.

More of the Output: This technique for iterative algorithms builds the output one piece at a time. Again, a recursive algorithm could have a friend build all but the last piece and have you add the last piece. However, it is better to have one friend build the first half of the output, another the second half, and you combine them somehow.

Narrowing the Search Space: Some iterative algorithms repeatedly narrow the search space in which to look for something. Instead, a recursive algorithm may split the search space in half and have a friend search each half.

Case Analysis: Instead of trying each of the cases oneself, one could give one case to each friend.

Work Done: Work does not accumulate in recursive algorithms as it does in iterative algorithms. We get each friend to do some work and then we do some work, ourselves to combine these solution.

11.4 The Towers of Hanoi

The towers of Hanoi is a classic puzzle for which the only possible way of solving it is to think recursively.

Specification: The puzzle consists of three poles and a stack of N disks of different sizes.

Precondition: All the disks are on the first of the three poles.

Postcondition: The goal is to move the stack over to the last pole. See the first and the last parts of [Figure 11.1](#).



Fig. 11.1 The towers of Hanoi problem.

You are only allowed to take one disk from the top of the stack on one pole and place it on the top of the stack on another pole. Another rule is that no disk can be placed on top of a smaller disk.

Lost with First Step: The first step must be to move the smallest disk. But it is by no means clear whether to move it to the middle or to the last pole.

Divide: Jump into the middle of the computation. One thing that is clear is that at some point, you must move the biggest disk from the first pole to the last. In order to do this, there can be no other disks on either the first or the last pole. Hence, all the other disks need to be stacked on the middle pole. See the second and the third parts of [Figure 11.1](#). This point in the computation splits the problem into two subproblems that must be solved. The first is how to move all the disks except the largest from the first pole to the middle. See the first and second parts of [Figure 11.1](#). The second is how to move these same disks from the middle pole to the last. See the third and fourth parts of [Figure 11.1](#).

Conquer: Together these steps solve the entire problem. Starting with all disks on the first pole, somehow move all but the largest to the second pole. Then, in one step, move the largest from the first to the third pole. Finally, somehow move all but the largest from the second to the third pole.

Magic: In order to make a clear separation between the task of solving the entire problem and that of solving each of the subproblems, I like to say that we delegate to one friend the task of solving one of the subproblems and delegate to another friend the other.

More General Specification: The subproblem of moving all but the largest disk from the first to the middle pole is very similar to original towers of Hanoi problem. However, it is an instance of a slightly more general problem, because not all of the disks are moved. To include this as an instance of our problem, we generalize the problem as follows.

Precondition: The input specifies the number n of disks to be moved and the roles of the three poles. These three roles for poles are peg_{source} , $peg_{destination}$, and peg_{spare} . The precondition requires that the smallest n disks be currently on peg_{source} . It does not care where the larger disks are.

Postcondition: The goal is to move these smallest n disks to $pole_{destination}$. Pole peg_{spare} is available to be used temporarily. The larger disks are not moved.

Subinstance: Our task is to move all the disks from the first to the last pole. This is specified by giving $n = N$, $peg_{source} = first$, $peg_{destination} = last$, and $peg_{spare} = middle$. We will get one friend to move all but the largest disk from the first to the middle pole. This is specified by giving $n = N - 1$, $peg_{source} = first$, $peg_{destination} = middle$, and $peg_{spare} = last$. On our own, we move the largest disk from the first to the last disk. Finally, we will get another friend to move all but the largest disk from the middle to the last pole. This is specified by giving $n = N - 1$, $peg_{source} = middle$, $peg_{destination} = last$, and $peg_{spare} = first$.

Code:

algorithm *TowersOfHanoi*($n, source, destination, spare$)

$\langle pre-cond \rangle$: The n smallest disks are on peg_{source} .

$\langle post-cond \rangle$: They are moved to $peg_{destination}$.

begin

 if($n \leq 0$)

 Nothing to do

 else

TowersOfHanoi($n - 1, source, spare, destination$)

 Move the n th disk from peg_{source} to $peg_{destination}$.

TowersOfHanoi($n - 1, spare, destination, source$)

 end if

end algorithm

Running Time: Let $T(n)$ be the time to move n disks. Clearly, $T(1) = 1$ and $T(n) = 2 \cdot T(n - 1) + 1$. Solving this gives $T(n) = 2^n - 1$.

11.5 Checklist for Recursive Algorithms

Writing a recursive algorithm is surprisingly hard when you are first starting out and surprisingly easy when you get it. This section contains a list of things to think about to make sure that you do not make any of the common mistakes.

0) The Code Structure: The code does not need to be much more complex than the following.

algorithm $Alg(a, b, c)$

<pre-cond>: Here a is a tuple, b an integer, and c a binary tree.

<post-cond>: Outputs x, y , and z , which are useful objects.

begin

if ($\langle a, b, c \rangle$ is a sufficiently small instance) return($\langle 0, 0, 0 \rangle$)

$\langle a_{sub1}, b_{sub1}, c_{sub1} \rangle =$ a part of $\langle a, b, c \rangle$

$\langle x_{sub1}, y_{sub1}, z_{sub1} \rangle = Alg(\langle a_{sub1}, b_{sub1}, c_{sub1} \rangle)$

$\langle a_{sub2}, b_{sub2}, c_{sub2} \rangle =$ a different part of $\langle a, b, c \rangle$

$\langle x_{sub2}, y_{sub2}, z_{sub2} \rangle = Alg(\langle a_{sub2}, b_{sub2}, c_{sub2} \rangle)$

$\langle x, y, z \rangle =$ combine $\langle x_{sub1}, y_{sub1}, z_{sub1} \rangle$ and $\langle x_{sub2}, y_{sub2}, z_{sub2} \rangle$

return($\langle x, y, z \rangle$)

end algorithm

1) Specifications: You must clearly define what the algorithm is supposed to do.

2) Variables: A great deal is understood about an algorithm by understanding its variables. As in any algorithm, you want variables to be well documented and to have meaningful names. It is also important to carefully check that you give variables values of the correct type, e.g., k is an integer, G is a graph, and so on. Moreover, with recursive programs there are variables that play specific roles and should be used in specific ways. This can be a source of many confusions and mistakes. Hence, I outline these carefully here.

2.1) Your Input: Your mission, if you are to accept it, is received through your inputs. The first line of your code, algorithm $Alg(a, b, c)$, specifies both the name Alg of the routine and the names of its inputs. Here $\langle a, b, c \rangle$ is the input instance that you need to find a solution for. I sometimes use $Alg(\langle a, b, c \rangle)$ because it emphasizes the viewpoint that we are receiving one instance, even if that instance might be composed of a tuple of things. Your preconditions must clearly specify what each of these components a, b , and c are and any restrictions on their values. You must be able to handle any instance that meets these conditions.

2.2) Your Output: You must return a solution $\langle x, y, z \rangle$ to your instance $\langle a, b, c \rangle$ through a return statement $return(\langle x, y, z \rangle)$. Your postconditions must clearly specify what each of the components x, y , and z of your solution are and their required relation to the input instance $\langle a, b, c \rangle$.

2.2.1) Every Path: Given any instance meeting the precondition, you must return a correct solution. Hence, if your code has *if* or *loop* statements, then every path through the code must end with a return statement.

2.2.2) Type of Output: Each return statement must return a solution $\langle x, y, z \rangle$ of the right type. The one partial exception to this is: if the postcondition leaves open the possibility that a solution does not exist, then some paths through the code may end with the statement *return* (“no solution exists”).

2.3) Your Friend's Input: To get help from friends, you must create a subinstance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ for each friend. You pass this to a friend by recursing with $Alg(\langle a_{sub}, b_{sub}, c_{sub} \rangle)$. To be able to give a subinstance to a friend, it needs to meet the preconditions of your problem. Do not recurse with $Alg(a_{sub}, b_{sub})$.

2.4) Your Friend's Output: You can trust that each friend will give you a correct solution $\langle x_{sub}, y_{sub}, z_{sub} \rangle$ to the subinstance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ that you give to them. Be sure to save their result in variables of the correct type, using the code $\langle x_{sub}, y_{sub}, z_{sub} \rangle = Alg(\langle a_{sub}, b_{sub}, c_{sub} \rangle)$. In contrast, the code $Alg(\langle a_{sub}, b_{sub}, c_{sub} \rangle)$ as a line by itself is insulting to your friend, because you got them to do all of this work and then you dropped their result in the garbage.

2.5) Rarely Need New Inputs or Outputs: I did speak of the need to generalize the problem by adding new inputs and/or outputs. This, however, is needed far less often than people think. Try hard to solve the problem using the friend analogy without extra variables. Only add them if absolutely necessary. If you do add extra inputs or outputs, clearly specify in the pre- and postconditions what they are for. Do not have inputs or outputs that are not explained.

2.6) No Global Variables or Global Effects: When you recurse with the line $\langle x_{sub}, y_{sub}, z_{sub} \rangle = Alg(\langle a_{sub}, b_{sub}, c_{sub} \rangle)$, the only thing that should happen is that your friend passes back a correct solution $\langle x_{sub}, y_{sub}, z_{sub} \rangle$ to the subinstance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ that you gave them. If the code has a local variable n , then your variable n is completely different from your friend's. (They are stored in different stack frames. See [Section 11.6](#).) If you set your variable n to 5 and then recurse, your friend's variable n will not have this value. If you want them to have a 5, you must pass it in as part of their subinstance $\langle a_{sub}, 5, c_{sub} \rangle$. Similarly, if your friend sets their variable n to 6 and then returns, your variable will not have this value, but will still have the value 5. If you want your friend to give you a 6, they must return it as part of their solution $\langle x_{sub}, 6, z_{sub} \rangle$.

I often suspect that people intend for a parameter in their algorithm's arguments to both pass a value in and pass a value out. Though I know there are programming languages that allow this, I strongly recommend not doing this. The code $\langle x_{sub}, y_{sub}, z_{sub} \rangle = Alg(\langle a_{sub}, b_{sub}, c_{sub} \rangle)$ does not change the values of a_{sub} , b_{sub} , or c_{sub} .

I have seen lots of code that loops n times recursing $Alg(\langle a_{sub}, b_{sub}, c_{sub} \rangle)$ on the exact same subinstance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$. One definition of insanity is repeating the same thing over and over and expecting to get a different result. Your friend on the same subinstance will give you the same solution. Do not waste their time.

It is tempting to use a global variable that everyone has access to. However, this is very bad form, mainly because it is very prone to errors and side effects that you did not expect.

Similarly, you can have no global returns. For example, suppose your friend's friend's friend finds something that you are looking for. It needs to be

passed back friend to friend, because things returned by your friends do not get returned to your boss unless you do the returning.

2.7) Few Local Variables: An iterative algorithm consists of a big loop with a set of local variables holding the current state. Each iteration of these variables gets updated. Because thinking iteratively comes more naturally to people, they want to do this with recursive algorithms. Don't. Generally, there is no need for a loop in a recursive algorithm unless you require the immediate help of many friends and you loop through them, creating subinstances for each and considering their subsolutions. In fact, despite the name "variable," rarely is there a need to change the value of a variable once initially set. For example, the variables $\langle a, b, c \rangle$ storing your instance are sacred. This is the instance you must solve. Why ever change it? You must construct a solution $\langle x, y, z \rangle$. Create it and return it. Why ever change it? Similarly for what you give $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ and receive $\langle x_{sub}, y_{sub}, z_{sub} \rangle$ from each friend. Other local variables are rarely needed. If you do need them, be sure to document what they are for.

3) Tasks to Complete: Your mission, given an arbitrary instance $\langle a, b, c \rangle$ meeting the preconditions, is to construct and return a solution $\langle x, y, z \rangle$ that meets the postcondition. The following are the only steps that you should be following toward this goal.

3.1) Accept Your Mission: Imagine that you have an $\langle a, b, c \rangle$ meeting the pre conditions. Know the range of things that your instance might be. For example, if the input instance is a binary tree, make sure that your program works for a general tree with big left and right subtrees, a tree with big left and empty right, a tree with empty left and big right, and the empty tree. Also know what is required of your output.

3.2) Construct Subinstances: For each friend, construct from your instance $\langle a, b, c \rangle$ a subinstance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ to give this friend. Sometimes this requires a block of code. Sometimes it happens right in place. For example, if your instance is $\langle \langle a_1, a_2, \dots, a_n \rangle, b, c \rangle$, you might construct the subinstance $\langle \langle a_1, a_2, \dots, a_{n-1} \rangle, b - 5, \text{leftSub}(c) \rangle$ for your friend by stripping the last object off the tuple a , subtracting 5 from the integer b , and taking the left subtree of the tree c . This subinstance can be constructed and passed to your friend in the one line

$$\langle x_{sub}, y_{sub}, z_{sub} \rangle = \text{Alg}(\langle \langle a_1, a_2, \dots, a_{n-1} \rangle, b - 5, \text{leftSub}(c) \rangle).$$

3.2.1) Valid Subinstance: Be sure that the subinstance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ that you give your friend meets the preconditions.

3.2.2) Smaller Subinstance: Be sure that the subinstance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ that you give your friend is smaller in some way than your own subinstance $\langle a, b, c \rangle$.

3.3) Trust Your Friend: Focus on only your mission. Trust your friend to give you a correct solution $\langle x_{sub}, y_{sub}, z_{sub} \rangle$ to the instance $\langle a_{sub}, b_{sub}, c_{sub} \rangle$ that you

give them. Do not worry about how they get their answer. Do not trace through the entire computation. Do not talk of your friends' friends' friends. I cannot emphasize this enough. Time and time again, I see students not trusting. It causes them no end of trouble until they finally see the light and let go.

3.4) Construct Your Solution: Using the solutions $\langle x_{sub}, y_{sub}, z_{sub} \rangle$ provided by your friends for your subinstances $\langle a_{sub}, b_{sub}, c_{sub} \rangle$, your next task is to construct a solution $\langle x, y, z \rangle$ for your subinstance $\langle a, b, c \rangle$. This generally requires a block of code, but sometimes it can be contained in a single line. For example, if the only output is a single integer x , then the one line of code `return(Alg( $a_{sub1}, b_{sub1}, c_{sub1}$ ) + Alg( $a_{sub2}, b_{sub2}, c_{sub2}$ )))` combines the friends' solutions x_{sub1} and x_{sub2} to give your solution $x = x_{sub1} + x_{sub2}$ and returns it.

3.5) Base Cases: Consider which instances get solved by your program. For those that don't, either add more cases to solve them recursively or add base cases to solve them in a brute force way. If your input instance is sufficiently small according to *your* definition of size, then you must solve it yourself as a base case.

This is all that you need to do. Do not do more.

Exercise 11.5.1 You are now the professor. Which of the above steps to develop a recursive algorithm did the students fail to do correctly in the following code? How? How would you fix it? See [Exercise 13.3.1](#) for correct code for this problem.

algorithm *Smallest*(*tree*, *k*, *num*, *v*)

<pre-cond>: *tree* is a binary search tree and $k > 0$ is an integer.

<post-cond>: Outputs the *k*th smallest element *s*.

begin

```

    if( $k = 0$ ) return(0)
    if( $v = k$ ) return(element)
     $n = 0$ 
    Smallest(leftSub(tree))
    ++  $n$ 
    if( $n = k$ )
        return(root(tree))
    end if
    Smallest(rightSub(tree))
    end if

```

end algorithm

algorithm *Smallest*(*tree*, *k*, *num*, *v*)

<pre-cond>: *tree* is a binary search tree and $k > 0$ is an integer.

<post-cond>: Outputs the *k*th smallest element *s*.

begin

```

     $n = 0$ 
    while( $n < k$ )

```

```

    Smallest(leftSub(tree))
    ++ n
    Smallest(rightSub(tree))
  end while
  return (element)
end algorithm

```

Exercise 11.5.2 (See solution in Exercise Solutions.) In the friends level of abstracting recursion, you can give your friend any legal instance that is smaller than yours according to some measure as long as you solve in your own any instance that is sufficiently small. For which of these algorithms has this been done? If so, what is your measure of the size of the instance? On input instance $\langle n, m \rangle$, either bound the depth to which the algorithm recurses as a function of n and m , or prove that there is at least one path down the recursion tree that is infinite.

algorithm $R_a(n, m)$

<pre-cond>: n & m ints.

<post-cond>: Say Hi.

begin

 if($n \leq 0$)

 Print("Hi")

 else

$R_a(n - 1, 2m)$

 end if

end algorithm

algorithm $R_b(n, m)$

<pre-cond>: n & m ints.

<post-cond>: Say Hi.

begin

 if($n \leq 0$)

 Print("Hi")

 else

$R_b(n - 1, m)$

$R_b(n, m - 1)$

 end if

end algorithm

algorithm $R_c(n, m)$

<pre-cond>: n & m ints.

<post-cond>: Say Hi.

begin

 if($n \leq 0$ or $m \leq 0$)

 Print("Hi")

 else

$R_c(n - 1, m)$

$R_c(n, m - 1)$

 end if

end algorithm

(d) Replace recursive lines with

$R_d(n - 1, m + 2)$

$R_e(n + 6, m - 3)$

(e) Replace recursive lines with

$R_e(n - 4, m + 2)$

$R_e(n + 6, m - 3)$

11.6 The Stack Frame

Tree of Stack Frames: Tracing out the entire computation of a recursive algorithm, one line of code at a time, can get incredibly complex. This is why the friends level of abstraction, which considers one stack frame at a time, is the best way to understand,

explain, and design a recursive algorithm. However, it is also useful to have some picture of the entire computation. For this, the tree-of-stack-frames level of abstraction is best.

The key thing to understand is the difference between a particular routine and a particular execution of a routine on a particular input instance. A single routine can at one moment in time have many executions going on. Each such execution is referred to as a *stack frame*. You can think of each as the task given to a separate friend. Even though each friend may be executing exactly the same routine, each execution may currently be on a different line of code and have different values for the local variables.

If each routine makes a number of subroutine calls (recursive or not), then the stack frames that get executed form a tree. In the example in [Figure 11.2](#), instance *A* is called first. It executes for a while and at some point recursively calls *B*. When *B* returns, *A* then executes for a while longer before calling *H*. When *H* returns, *A* executes for a while before completing. We have skipped over the details of the execution of *B*. Let's go back to when instance *A* calls *B*. Then *B* calls *C*, which calls *D*. Instance *D* completes; then *C* calls *E*. After *E*, *C* completes. Then *B* calls *F*, which calls *G*. Then *G* completes, *F* completes, *B* completes, and *A* goes on to call *H*. It does get complicated.

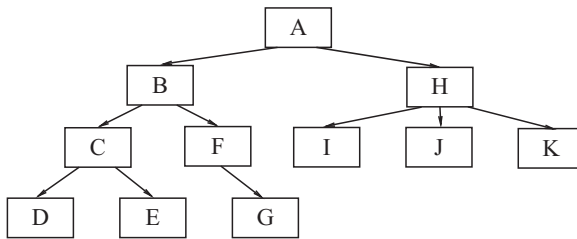


Fig. 11.2 Tree of stack frames.

Stack of Stack Frames: The algorithm is actually implemented on a computer by a stack of stack frames. What is stored in the computer memory at any given point in time is only a single path down the tree. The tree represents what occurs throughout time. In [Figure 11.2](#), when instance *G* is active, *A*, *B*, *F*, and *G* are in the stack. *C*, *D*, and *E* have been removed from memory as these have completed. *H*, *I*, *J*, and *K* have not been started yet. Although we speak of many separate stack frames executing on the computer, the computer is not a parallel machine. Only the top stack frame *G* is actively being executed. The other instances are on hold, waiting for the return of a subroutine call that it made.

Memory: Here is how memory is managed for the simultaneous execution of many instances of the same routine. The routine itself is described only once, by a block of code that appears in static memory. This code declares a set of variables. On the other hand, each instance of this routine that is currently being executed may be storing different values in these variables and hence needs to have its own separate copy of these variables. The memory requirements of each of these instances are stored in a separate *stack frame*. These frames are stacked on top of each other within stack memory.

Using a Stack Frame: Recall that a stack is a data structure in which either a new element is *pushed* onto the top or the last element to have been added is *popped* off (Section 13.1). Let us denote the top stack frame by A . When the execution of A makes a subroutine call to a routine with some input values, a stack frame is created for this new instance. This frame denoted B is pushed onto the stack after that for A . In addition to a separate copy of the local variables for the routine, it contains a pointer to the next line of code that A must execute when B returns. When B returns, its stack frame is popped, and A continues to execute at the line of code that had been indicated within B . When A completes, it too is popped off the stack.

Silly Example: This example demonstrates how difficult it is to trace out the full stack-frame tree, yet how easy it is to determine the output using the friends (strong-induction) method:

algorithm $Fun(n)$

<pre-cond>: n is an integer.

<post-cond>: Outputs a silly string.

begin

 if($n > 0$) then

 if ($n = 1$) then

 put “X”

 else if($n = 2$) then

 put “Y”

 else

 put “A”

$Fun(n - 1)$

 Put “B”

$Fun(n - 2)$

 Put “C”

 end if

 end if

end algorithm

Exercise 11.6.1 Attempt to trace out the tree of stack frames for the silly example $Fun(5)$.

Exercise 11.6.2 (See solution in Exercise Solutions.) Now try the following simpler approach. What is the output of $Fun(1)$? What is the output of $Fun(2)$? Trust the answers to all previous questions; do not recalculate them. (Assume a trusted friend gave you the answer.) Now, what is the output of $Fun(3)$? Repeat this approach for $n = 4, 5$, and 6.

11.7 Proving Correctness with Strong Induction

Whether you give your subinstances to friends or you recurse on them, this level of abstraction considers only the algorithm for the top stack frame. We must now prove

that this suffices to produce an algorithm that successfully solves the problem for every input instance. When proving this, it is tempting to talk about stack frames. This stack frame calls this one, which calls that one, until you hit the base case. Then the solutions bubble back up to the surface. These proofs tend to make little sense. Instead, we use strong induction to prove formally that the friends level of abstraction works.

Strong Induction: Strong induction is similar to induction, except that instead of assuming only $S(n-1)$ to prove $S(n)$, you must assume all of $S(0), S(1), S(2), \dots, S(n-1)$.

A Statement for Each n : For each value of $n \geq 0$, let $S(n)$ represent a Boolean statement. For some values of n , this statement may be true, and for others it may be false.

Goal: Our goal is to prove that it is true for every value of n , namely that $\forall n \geq 0, S(n)$.

Proof Outline: Proof by strong induction on n .

Induction Hypothesis: For each $n \geq 0$, let $S(n)$ be the statement that $\dots$ (It is important to state this clearly.)

Base Case: Prove that the statement $S(0)$ is true.

Induction Step: For each $n \geq 0$, prove $S(0), S(1), S(2), \dots, S(n-1) \Rightarrow S(n)$.

Conclusion: By way of induction, we can conclude that $\forall n \geq 0, S(n)$.

See [Exercises 11.7.1](#) and [11.7.2](#).

Proving the Recursive Algorithm Works

Induction Hypothesis: For each $n \geq 0$, let $S(n)$ be the statement “The recursive algorithm works for *every* instance of size n .”

Goal: Our goal is to prove that $\forall n \geq 0, S(n)$, i.e., that the recursive algorithm works for *every* instance.

Proof Outline: The proof is by strong induction on n .

Base Case: Proving $S(0)$ involves showing that the algorithm works for the base cases of size $n = 0$.

Induction Step: The statement $S(0), S(1), S(2), \dots, S(n-1) \Rightarrow S(n)$ is proved as follows. First assume that the algorithm works for every instance of size strictly smaller than n , and then prove that it works for every instance of size n . This mirrors exactly what we do on the friends level of abstraction. To prove that the algorithm works for every instance of size n , consider an arbitrary instance of size n . The algorithm constructs subinstances that are strictly smaller. By our induction hypothesis we know that our algorithm works for these. Hence, the recursive calls return the correct solutions. On the friends level of abstraction, we proved that the

algorithm constructs the correct solutions to our instance from the correct solutions to the subinstances. Hence, the algorithm works for this arbitrary instance of size n . The $S(n)$ follows.

Conclusion: By way of strong induction, we can conclude that $\forall n \geq 0$, $S(n)$, i.e., the recursive algorithm works for every instance.

Exercise 11.7.1 Give the process of strong induction as we did for regular induction.

Exercise 11.7.2 (See solution in Exercise Solutions.) As a formal statement, the base case can be eliminated in strong induction because it is included in the formal induction step. How is this? (In practice, the base cases are still proved separately.)

12 Some Simple Examples of Recursive Algorithms

I will now give some simple examples of recursive algorithms. Even if you have seen them before, study them again, keeping the techniques and theory from [Chapter 8](#) in mind. For each example, look for the key steps of the friend paradigm. What are the subinstances given to the friend? What is the size of an instance? Does it get smaller? How are the friend's solutions combined to give your solution? What does the tree of stack frames look like? What is the time complexity of the algorithm?

12.1 Sorting and Selecting Algorithms

The classic divide-and-conquer algorithms are merge sort and quick sort. They both have the following basic structure.

General Recursive Sorting Algorithm

- Take the given list of objects to be sorted (numbers, strings, student records, etc.).
- Split the list into two sublists.
- Recursively have friends sort each of the two sublists.
- Combine the two sorted sublists into one entirely sorted list.

This process leads to four different algorithms, depending on the following factors (see [Exercise 12.1.1](#)):

Sizes: Do you split the list into two sublists each of size $\frac{n}{2}$, or one of size $n - 1$ and one of size one?

Work: Do you put minimal effort into splitting the list but put lots of effort into recombining the sublists, or put lots of effort into splitting the list but put minimal effort into recombining the sublists?

Example 12.1.1 (Merge Sort (Minimal Work to Split in Half))

This is the classic recursive algorithm.

Friend's Level of Abstraction: Recursively give one friend the first half of the input to sort and another friend the second half to sort. Then combine these two sorted sublists into one completely sorted list. This combining process is referred to as *merging*. A simple linear-time algorithm for it can be found in [Section 3.3](#).

Size: The size of an instance is the number of elements in the list. If this is at least two, then the sublists are smaller than the whole list. Hence, it is valid to recurse on them with the reassurance that your friends will do their parts correctly. On the other hand, if the list contains only one element, then by default it is already sorted and nothing needs to be done.

Generalizing the Problem: If the input is assumed to be received in an array indexed from 1 to n , then the second half of the list is not a valid instance, because it is not indexed from 1. Hence, we redefine the preconditions of the sorting problem to require as input both an array A and a subrange $[i,j]$. The postcondition is that the specified sublist is to be sorted in place.

Running Time: Let $T(n)$ be the total time required to sort a list of n elements. This total time consists of the time for two subinstances of half the size to be sorted, plus $\Theta(n)$ time for merging the two sublists together. This gives the recurrence relation $T(n) = 2T(n/2) + \Theta(n)$. See [Chapter 39](#) to learn how to solve recurrence relations like these. In this example, $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$ and $f(n) = \Theta(n^1)$, so $c = 1$. Because $\frac{\log a}{\log b} = c$, the technique concludes that the time is dominated by all levels and $T(n) = \Theta(f(n) \log n) = \Theta(n \log n)$.

Tree of Stack Frames: [Figure 12.1](#) is a tree of stack frames for a concrete example.

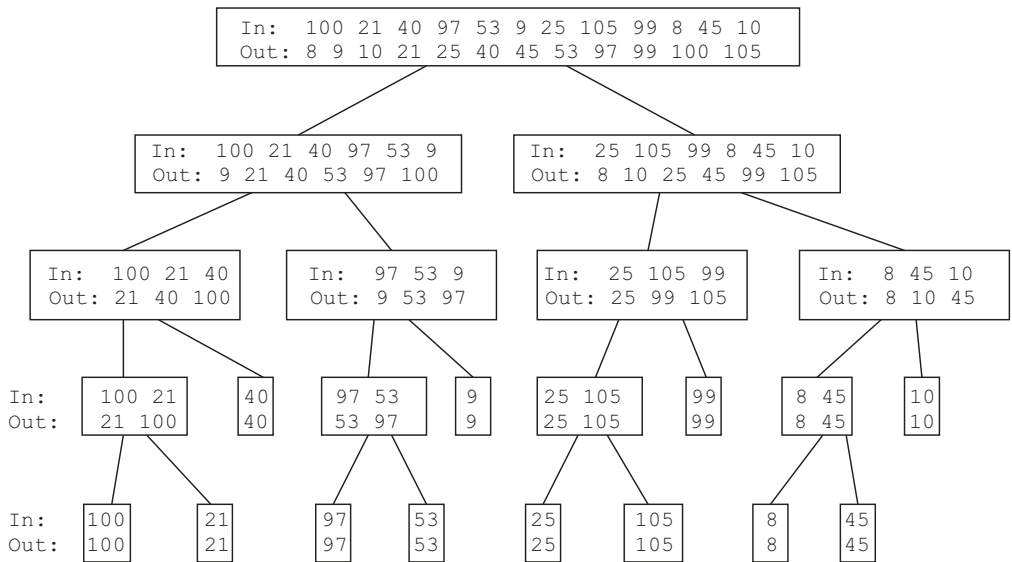


Fig. 12.1 Tree of Stack Frames in an execution of Merge Sort from [Example 12.1.1](#).

Example 12.1.2 (Quick Sort (Minimal Work to Recombine the Halves))

The following is one of the fastest sorting algorithms. Hence the name.

Friend’s Level of Abstraction: The algorithm partitions the list into two sublists where all the elements that are less than or equal to a chosen *pivot element* are to the

left of the pivot element and all the elements that are greater than it are to the right of it. (There are no requirements on the order of the elements in the sublists.) Next, recursively have a friend sort those elements before the pivot and those after it. Finally, (without effort) put the sublists together, forming one completely sorted list.

The first step in the algorithm is to choose one of the elements to be the pivot element. How this is to be done is discussed below.

Tree of Stack Frames: Figure 12.2 is a tree of stack frames for a specific input.

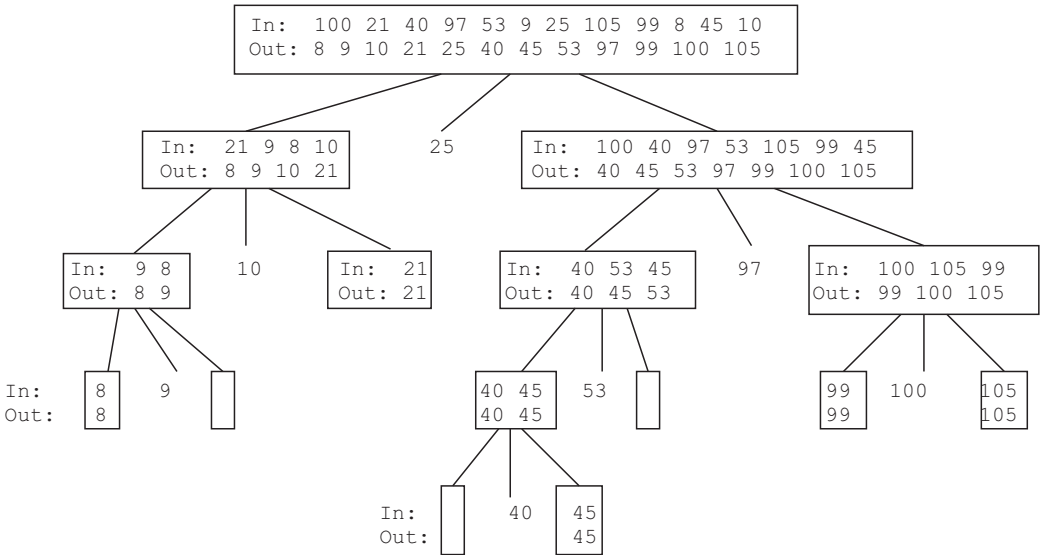


Fig. 12.2 Tree of Stack Frames in an execution of Quick Sort from Example 12.1.2.

Running Time: The computation time depends on the choice of the pivot element.

Median: If we are lucky and the pivot element is close to having the median value, then the list will be split into two sublists of size approximately $n/2$. We will see that partitioning the array according to the pivot element can be done in time $\Theta(n)$. In this case, the timing is $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.

Reasonable Split: The above timing is quite robust with respect to the choice of the pivot. For example, suppose that the pivot always partitions the list into one sublist of one-fifth the original size and one of four-fifths the original size. The total time is then the time to partition plus the time to sort the sublists of these sizes. This gives $T(n) = T(\frac{1}{5}n) + T(\frac{4}{5}n) + \Theta(n)$. Because $\frac{1}{5} + \frac{4}{5} = 1$, this evaluates to $T(n) = \Theta(n \log n)$. (See Chapter 39.)

Worst Case: On the other hand, suppose that the pivot always splits the list into one of size $n - 1$ and one of size 1. In this case, $T(n) = T(n - 1) + T(1) + \Theta(n)$, which evaluates to $T(n) = \Theta(n^2)$. This is the worst case scenario.

We will return to quick sort after considering the following related problem.

Example 12.1.3 (Finding the k th Smallest Element)

Given an unsorted list and an integer k , this example finds the k th smallest element from the list. It is not clear at first that there is an algorithm for doing this that is any faster than sorting the entire list. However, it can be done in linear time using the subroutine *Pivot*.

Friend's Level of Abstraction: The algorithm is like that for binary search. Ignoring input k , it proceeds just like quick sort. A pivot element is chosen randomly, and the list is split into two sublists, the first containing all elements that are all less than or equal to the pivot element and the second those that are greater than it. Let ℓ be the number of elements in the first sublist. If $\ell \geq k$, then we know that the k th smallest element from the entire list is also the k th smallest element from the first sublist. Hence, we can give this first sublist and this k to a friend and ask them to find it. On the other hand, if $\ell < k$, then we know that the k th smallest element from the entire list is the $(k - \ell)$ th smallest element from the second sublist. Hence, on giving the second sublist and $k - \ell$ to a friend, they can find it.

Tree of Stack Frames: [Figure 12.3](#) is a tree of stack frames for our input.

Running Time: Again, the computation time depends on the choice of the pivot element.

Median: If we are lucky and the pivot element is close to the median value, then the list will be split into two sublists of size approximately $n/2$. Because the routine recurses on only one of the halves, the timing is

$$T(n) = T(n/2) + \Theta(n) = \Theta(n).$$

Reasonable Split: If the pivot always partitions the list so that the larger half is at most $\frac{4}{5}n$, then the total time is at most $T(n) = T(\frac{4}{5}n) + \Theta(n)$, which is still linear time, $T(n) = \Theta(n)$.

Worst Case: In the worst case, the pivot splits the list into one of size $n - 1$ and one of size 1. In this case, $T(n) = T(n - 1) + \Theta(n)$, which is $T(n) = \Theta(n^2)$.

Choosing the Pivot: In [Examples 12.1.2](#) and [12.1.3](#), the timing depends on choosing a good pivot element quickly.

Fixed Value: If you know that you are sorting elements that are numbers within the range $[1 \dots 100]$, then it is reasonable to partition these elements based on whether they are smaller or larger than 50. This is often referred to as *bucket sort*. See [Section 5.1](#). However, there are two problems with this technique. The first is that in general we do not know what range the input elements will lie in. The second is that at every level of recursion, another pivot value is needed with which to partition the elements. The solution is to use the input itself to choose the pivot value.

Use $A[1]$ as the Pivot: The first thing one might try is to let the pivot be the element that happens to be first in the input array. The problem with this is that if

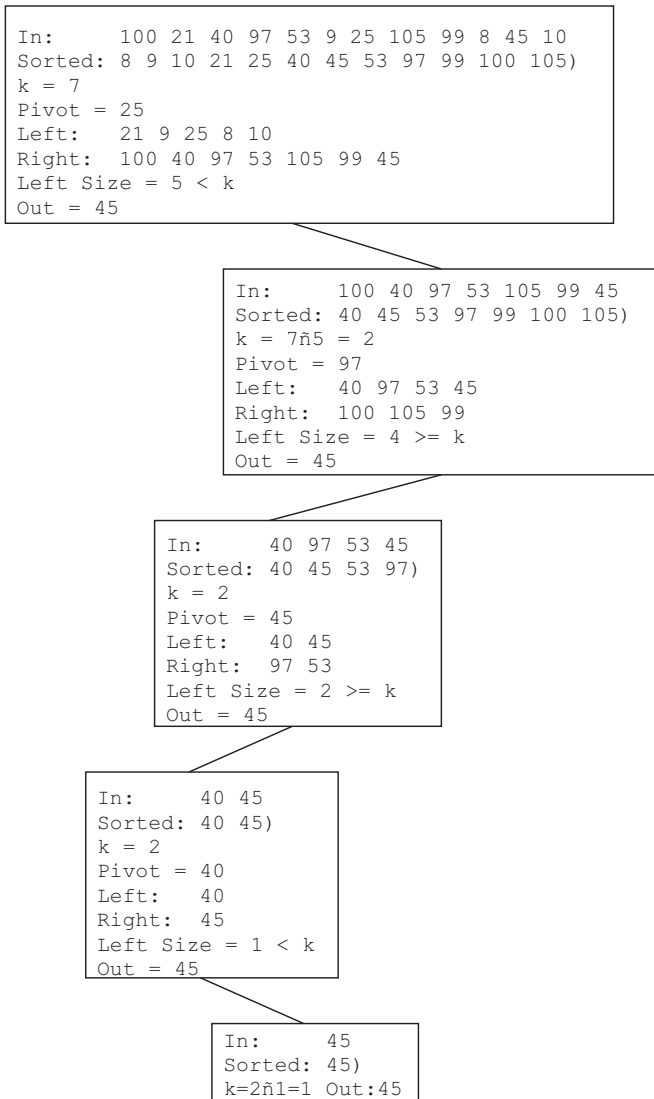


Fig. 12.3 Tree of Stack Frames in an execution of Finding the k th Smallest Element from Example 12.1.3.

the input happens to be sorted (or almost sorted) already, then this first element will split the list into one of size zero and one of size $n - 1$. This gives a worst case time of $\Theta(n^2)$. Given random data, the algorithm will execute quickly. On the other hand, if you forget that you sorted the data and you run it a second time, then the second run will take a long time to complete.

Use $A[\frac{n}{2}]$ as the Pivot: Motivated by the last attempt, one might use the element that happens to be located in the middle of the input array. For all practical purposes, this would likely work well. It would work exceptionally well when the

list is already sorted. However, there are some strange inputs cooked up for the sole purpose of being nasty to this particular implementation of the algorithm, on which the algorithm runs in $\Theta(n^2)$ time. The adversary will provide such an input, giving a worst case time complexity of $\Theta(n^2)$.

A Randomly Chosen Element: In practice, what is often done is to choose the pivot element randomly from the input elements. See [Section 29.1](#). The advantage of this is that the adversary who is choosing the worst case input instance knows the algorithm, but does not know the random coin tosses. Hence, all input instances are equally good and equally bad.

We will prove that the expected computation time is $\Theta(n \log n)$. What this means is that if you ran the algorithm 1,000,000 times on the same input, then the average running time would be $\Theta(n \log n)$.

Intuition: One often gains good intuition by assuming that what we expect to happen, happens reasonably often. If the pivot always partitions the list into one sublist of one-fifth the original size and one of four-fifths the original size, then the total time is $T(n) = T(\frac{1}{5}n) + T(\frac{4}{5}n) + \Theta(n) = \Theta(n \log n)$. When a pivot is chosen randomly, the probability that it partitions the list at least this well is $\frac{3}{5}$. When a partition is worse than this, it is not a big problem. We just say that no significant progress is made, and we try again. After all, we expect to make progress in approximately three of every five partitions.

More Formal: Formally, we set up and solve a difficult recurrence relation. Suppose that the randomly chosen pivot element happens to be the i th smallest element. This splits the list into one of size i and one of size $n - i$, in which case the running time is $T(i) + T(n - i) + \Theta(n)$. Averaging this over all possible values of i gives the recursive relation $T(n) = \text{Avg}_{i \in [0..n]} [T(i) + T(n - i) + \Theta(n)]$. With a fair bit of work, this evaluates to $\Theta(n \log n)$.

Randomly Choose Three Elements: Another option is to randomly select three elements from the input list and use the middle one as the pivot. Doing this greatly increases the probability that the pivot is close to the middle and hence decreases the probability of the worst case occurring. However, it so also takes time. All in all, the expected running time is worse.

A Deterministic Algorithm: Though in practice such a probabilistic algorithm is easy to code and works well, theoretical computer scientists like to find a deterministic algorithm that is guaranteed to run quickly.

The following is a deterministic method of choosing the pivot that leads to a worst case running time of $\Theta(n)$ for finding the k th smallest element. First group the n elements into $\frac{n}{5}$ groups of five elements each. Within each group of five elements, do $\Theta(1)$ work to find the median of the group. Let S_{median} be the set of $\frac{n}{5}$ elements containing the median from each group. Recursively ask a friend to

find the median element from the set S_{median} . This element will be used as our pivot.

I claim that this pivot element has at least $\frac{3}{10}n$ elements that are less than or equal to it and another $\frac{3}{10}n$ elements that are greater than or equal to it. The proof of the claim is as follows. Because the pivot is the median within S_{median} , there are $\frac{1}{10}n = \frac{1}{2}|S_{median}|$ elements within S_{median} that are less than or equal to the pivot. Consider any such element $x_i \in S_{median}$. Because x_i is the median within its group of five elements, there are three elements within this group (including x_i itself) that are less than or equal to x_i and hence in turn less than or equal to the pivot. Counting all these gives $3 \times \frac{1}{10}n$ elements. A similar argument counts this many that are greater than or equal to the pivot.

The algorithm to find the k th largest element proceeds as stated originally. A friend is asked to find either the k th smallest element within all elements that are less than or equal to the pivot or the $(k - \ell)$ th smallest element from all those that are greater than it. The claim ensures that the size of the sublist given to the friend is at most $\frac{7}{10}n$.

Unlike the first algorithm for finding k th smallest element, this algorithm recurses twice. Hence, one would naively assume that the running time is $\Theta(n \log n)$. However, careful analysis shows that it is only $\Theta(n)$. Let $T(n)$ denote the running time. Finding the median of each of the $\frac{1}{5}n$ groups takes $\Theta(n)$ time. Recursively finding the median of S_{median} takes $T(\frac{1}{5}n)$ time. Recursing on the remaining at most $\frac{7}{10}n$ elements takes at most $T(\frac{7}{10}n)$ time. This gives a total of $T(n) = T(\frac{1}{5}n) + T(\frac{7}{10}n) + \Theta(n)$ time. Because $\frac{1}{5} + \frac{7}{10} < 1$, this evaluates to $T(n) = \Theta(n)$. (See [Chapter 39](#).)

A deterministic quick sort algorithm can use this deterministic $\Theta(n)$ -time algorithm for finding the k th smallest element, to find the median of the list to be the pivot. Because partitioning the elements according to the pivot already takes $\Theta(n)$ time, the timing is still $T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$.

Partitioning According to the Pivot Element: This is an iterative step. The input consists of a list of elements $A[I], \dots, A[J]$ and a pivot element. The output consists of the rearranged elements and an index i , such that the elements $A[I], \dots, A[i - 1]$ are all less than or equal to the pivot element, $A[i]$ is the pivot element, and the elements $A[i + 1], \dots, A[J]$ are all greater than it.

The loop invariant is that there are indices $I \leq i \leq j \leq J$ for which:

1. The values in $A[I], \dots, A[i - 1]$ are less than or equal to the pivot element.
2. The values in $A[j + 1], \dots, A[J]$ are greater than the pivot element.
3. The pivot element has been removed and is on the side, leaving an empty entry either at $A[i]$ or at $A[j]$.
4. The other elements in $A[i], \dots, A[j]$ have not been considered.

The loop invariant is established by setting $i = I$ and $j = J$, making $A[i]$ empty by putting the element in $A[i]$ where the pivot element is and putting the pivot element aside.

If the loop invariant is true and $i < j$, then there are four possible cases (see Figure 12.4):

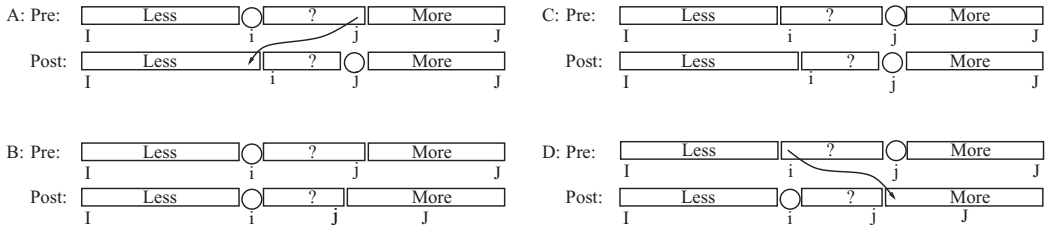


Fig. 12.4 The four cases of how to iterate are shown.

Case A. $A[i]$ is empty and $A[j] \leq \text{pivot}$: $A[j]$ belongs on the left, so move it to the empty $A[i]$. Now $A[j]$ is empty. Increase the left side by increasing i by one.

Case B. $A[i]$ is empty and $A[j] > \text{pivot}$: $A[j]$ belongs on the right and is already there. Increase the right side by decreasing j by one.

Case C. $A[j]$ is empty and $A[i] \leq \text{pivot}$: $A[i]$ belongs on the left and is already there. Increase the left side by increasing i by one.

Case D. $A[j]$ is empty and $A[i] > \text{pivot}$: $A[i]$ belongs on the right, so move it to the empty $A[j]$. Now $A[i]$ is empty. Increase the right side by decreasing j by one.

In each case, the loop invariant is maintained. Progress is made because $j - i$ decreases.

When $i = j$, the list is split as needed, leaving $A[i]$ empty. Put the pivot there. The postcondition follows.

Exercise 12.1.1 (See solution in Exercise Solutions.) Consider the algorithm that puts minimal effort into splitting the list into one of size $n - 1$ and one of size one, but puts lots of effort into recombining the sublists. Also consider the algorithm that puts lots of effort into splitting the list into one of size $n - 1$ and one of size one, but puts minimal effort into recombining the sublists. What are these two algorithms?

Exercise 12.1.2 (See solution in Exercise Solutions.) One-friend recursion vs. iteration.

1. Your task is to accept a tuple $\langle a_1, a_2, \dots, a_n \rangle$ and return the reversed tuple $\langle a_n, a_{n-1}, \dots, a_1 \rangle$. Being lazy, you will only strip off an element from one end or add an element back onto one end. But you have recursive friends to help you. Provide both a paragraph containing the friends' explanation of the algorithm, and the recursive code.
2. Now suppose that you have a stack, but friends. (See Chapter 3.) Quickly sketch an iterative program that solves this same problem. Be sure to include loop invariants and other key steps required for describing an iterative algorithm.
3. Trace each of these two programs. Step by step, compare and contrast their computations on a computer.

Exercise 12.1.3 Exercise 4.4.1 asks for an iterative algorithm for searching within a matrix $A[1 \dots n, 1 \dots m]$ in which each row is sorted and each column is sorted. This requires that $T(n, m) = n + m - 1$ of the matrix entries be examined. Exercise 7.0.7 proves that this is tight when $n = m$. But it is clearly too big when $m \gg n$, given one can do binary search in each row in time $n \log m \ll n + m - 1$. The goal now is to design a recursive algorithm that accesses $T(n, m) \approx n \log_2(\frac{m}{n})$ entries. As a huge hint, the recurrence relation will be $T(n, m) = \max_{n' \in [1, n]} T(n', \frac{m}{2}) + T(n - n', \frac{m}{2}) + \log_2 n$. You must look at the recursive tree in order to get some intuition to why the time is $T(n, m) \approx n \log_2(\frac{m}{n})$. You can also plug $T(n, m) = n \log_2(\frac{m}{n}) + 2n - \log(n) - 2$ into this recurrence relation and see that it satisfies it.

12.2 Operations on Integers

Raising an integer to a power b^N , multiplying $x \times y$, and matrix multiplication each have surprising divide-and-conquer algorithms.

Example 12.2.1 (b^N)

Suppose that you are given two integers b and N and want to compute b^N .

The Iterative Algorithm: The obvious iterative algorithm simply multiplies b together N times. The obvious recursive algorithm recurses with $\text{Power}(b, N) = b \times \text{Power}(b, N - 1)$. This requires the same N multiplications.

The Straightforward Divide-and-Conquer Algorithm: The obvious divide-and-conquer technique cuts the problem into two halves using the property that $b^{\lceil \frac{N}{2} \rceil} \times b^{\lfloor \frac{N}{2} \rfloor} = b^{\lceil \frac{N}{2} \rceil + \lfloor \frac{N}{2} \rfloor} = b^N$. This leads to the recursive algorithm $\text{Power}(b, N) = \text{Power}(b, \lceil \frac{N}{2} \rceil) \times \text{Power}(b, \lfloor \frac{N}{2} \rfloor)$. Its recurrence relation gives $T(N) = 2T(\frac{N}{2}) + 1$ multiplications. The technique in Chapter 39 notes that $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$ and $f(N) = \Theta(N^0)$, so $c = 0$. Because $\frac{\log a}{\log b} > c$, the technique concludes that time is dominated by the base cases and $T(N) = \Theta(N^{(\log a)/(\log b)}) = \Theta(N)$. This is no faster than the standard iterative algorithm.

Reducing the Number of Recursions: This algorithm can be improved by noting that the two recursive calls are almost the same and hence need only to be made once. The new recurrence relation gives $T(N) = 1T(\frac{N}{2}) + 1$ multiplications. Here $\frac{\log a}{\log b} = \frac{\log 1}{\log 2} = 0$ and $f(N) = \Theta(N^0)$, so $c = 0$. Because $\frac{\log a}{\log b} = c$, we conclude that the time is dominated by all levels and $T(N) = \Theta(f(N) \log N) = \Theta(\log N)$ multiplications.

Code

algorithm $\text{Power}(b, N)$

<pre-cond>: $N \geq 0$ (N and b not both 0)

<post-cond>: Outputs b^N .

```

begin
  if(N = 0) then
    return(1)
  else
    half =  $\lfloor \frac{N}{2} \rfloor$ 
    p = Power(b, half)
    if(2 * half = N) then
      return(p * p) % if N is even,  $b^N = b^{\lfloor N/2 \rfloor} \cdot b^{\lfloor N/2 \rfloor}$ 
    else
      return(p * p * b) % if N is odd,  $b^N = b \cdot b^{\lfloor N/2 \rfloor} \cdot b^{\lfloor N/2 \rfloor}$ 
    end if
  end if
end algorithm

```

Tree of Stack Frames (Figure 12.5)

Running Time

Input Size: One is tempted to say that the first two $\Theta(N)$ algorithms require a linear number of multiplications and that the last $\Theta(\log N)$ one requires a logarithmic number. However, in fact the first two require exponential $\Theta(2^n)$ number and the last a linear $\Theta(n)$ number in the size of the input, which is typically taken as the number of bits, $n = \log N$, needed to represent the number.

Operation: Is it fair to count the multiplications and not the bit operations in this case? I say not! The output b^N contains $\Theta(N \log b) = 2^{\Theta(n)}$ bits, and hence it will take this many bit operations to simply output the answer. Given this, it is not really fair to say that the time complexity is only $\Theta(n)$.

Mod a Prime: In order to avoid this blowup in the length of the integer, it is common given $\langle b, N, p \rangle$ to instead compute $b^N \equiv_{\text{mod } p} c$ using the above fast algorithm. The inverse operation, however, is hard, i.e., given $\langle b, c, p \rangle$ finding N . This is called *discrete log*. The best known algorithm requires $\Theta(N) = \Theta(2^n)$ multiplications. It is the basis of various cryptographic systems. See [Exercise 12.2.1](#), item 4, for more.

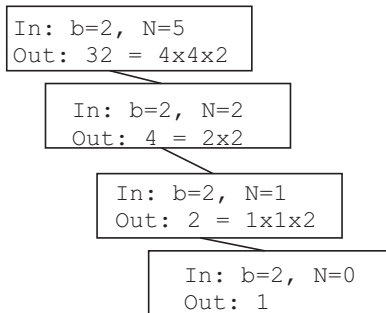


Fig. 12.5 Tree of Stack Frames in an execution of b^N from [Example 12.2.1](#).

Example 12.2.2 $(x \times y)$

The time complexity of [Example 12.2.1](#) was measured in terms of the number of multiplications. This ignores the question of how quickly one can multiply.

The input for the next problem consists of two strings of n digits each. These are viewed as two integers x and y , either in binary or in decimal notation. The problem is to multiply them.

The Iterative Algorithm: The standard elementary school algorithm considers each pair of digits, one from x and the other from y , and multiplies them together. These n^2 products are shifted appropriately and summed. The total time is $\Theta(n^2)$. It is hard to believe that one could do it faster.

$$\begin{array}{r}
 \begin{array}{r}
 8 \ 2 \ 7 \\
 5 \ 9 \ 6 \\
 \hline
 4 \ 2 \\
 1 \ 2 \\
 4 \ 8 \\
 6 \ 3 \\
 1 \ 8 \\
 7 \ 2 \\
 3 \ 5 \\
 1 \ 0 \\
 4 \ 0 \\
 \hline
 4 \ 9 \ 2 \ 8 \ 9 \ 2
 \end{array}
 \end{array}$$

The Straightforward Divide-and-Conquer Algorithm: Let us see how well the divide-and-conquer technique can work. Split each sequence of digits in half, and consider each half as an integer. This gives $x = x_1 \times 10^{n/2} + x_0$ and $y = y_1 \times 10^{n/2} + y_0$. Multiplying these symbolically gives

$$\begin{aligned}
 x \times y &= \left(x_1 \times 10^{\frac{n}{2}} + x_0\right) \times \left(y_1 \times 10^{\frac{n}{2}} + y_0\right) \\
 &= (x_1 y_1) \times 10^n + (x_1 y_0 + x_0 y_1) \times 10^{\frac{n}{2}} + (x_0 y_0).
 \end{aligned}$$

The obvious divide-and-conquer algorithm would recursively compute the four subproblems $x_1 y_1, x_1 y_0, x_0 y_1$, and $x_0 y_0$, each of $\frac{n}{2}$ digits. This would take $4T(\frac{n}{2})$ time. Then these four products are shifted appropriately and summed. Note that additions can be done in $\Theta(n)$ time. See [Section 2.2](#). Hence, the total time is $T(n) = 4T(\frac{n}{2}) + \Theta(n)$. Here $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2$ and $f(n) = \Theta(n^1)$, so $c = 1$. Because $\frac{\log a}{\log b} > c$, the technique concludes that the time is dominated by the base cases and $T(n) = \Theta(n(\log a)/(\log b)) = \Theta(n^2)$. This is no improvement in time.

Reducing the Number of Recursions: Suppose that we could find a trick so that we only needed to recurse three times instead of four. Intuitively, you might think that this would only provide a linear time saving, but in fact the saving is much more: $T(n) = 3T(\frac{n}{2}) + \Theta(n)$. Now $\frac{\log a}{\log b} = \frac{\log 3}{\log 2} = 1.58\dots$, which is still bigger than c . Hence,

time is still dominated by the base cases, but now this is $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{1.58\dots})$. This is a significant improvement over $\Theta(n^2)$.

The Trick: The first step is to multiply x_1y_1 and x_0y_0 recursively as required. This leaves us only one more recursive multiplication.

If you review the symbolic expansion for $x \times y$, you will see that we do not actually need to know the values of x_1y_0 and x_0y_1 . We only need to know their sum. Symbolically, we can observe the following:

$$\begin{aligned} x_1y_0 + x_0y_1 &= [x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0] - x_1y_1 - x_0y_0 \\ &= [(x_1 + x_0)(y_1 + y_0)] - x_1y_1 - x_0y_0. \end{aligned}$$

Hence, the sum $x_1y_0 + x_0y_1$ that we need can be computed by adding x_1 to x_0 and y_1 to y_0 ; multiplying these sums; and subtracting off the values x_1y_1 and x_0y_0 that we know from before. This requires only one additional recursive multiplication. Again, we use the fact that additions are fast, requiring only $\Theta(n)$ time.

Code

algorithm *Multiply*(x, y)

<pre-cond>: x and y are two integers represented as an array of n digits.

<post-cond>: The output consists of their product represented as an array of $n + 1$ digits.

```
begin
  if( $n = 1$ ) then
    result( $x \times y$ ) % product of single digits
  else
     $\langle x_1, x_0 \rangle =$  high- and low-order  $\frac{n}{2}$  digits of  $x$ 
     $\langle y_1, y_0 \rangle =$  high- and low-order  $\frac{n}{2}$  digits of  $y$ 
     $A = \text{Multiply}(x_1, y_1)$ 
     $C = \text{Multiply}(x_0, y_0)$ 
     $B = \text{Multiply}(x_1 + x_0, y_1 + y_0) - A - C$ 
    result( $A \times 10^n + B \times 10^{\frac{n}{2}} + C$ )
  end if
end algorithm
```

It is surprising that this trick reduces the time from $\Theta(n^2)$ to $\Theta(n^{1.58})$.

Dividing into More Parts: The next question is whether the same trick can be extended to improve the time even further. Instead of splitting each of x and y into two pieces, let's split them each into d pieces. The straightforward method recursively multiplies each of the d^2 pairs of pieces together, one from x and one from y . The total time is $T(n) = d^2 T(\frac{n}{d}) + \Theta(n)$. Here $a = d^2$, $b = d$, $c = 1$, and $\frac{\log d^2}{\log d} = 2 > c$. This gives $T(n) = \Theta(n^2)$. Again, we are back where we began.

Reducing the Number of Recursions: The trick now is to do the same with fewer recursive multiplications. It turns out it can be done with only $2d - 1$ of them. This

gives time of only $T(n) = (2d - 1)T(\frac{n}{d}) + \Theta(n)$. Here $a = 2d - 1$, $b = d$, $c = 1$, and $\frac{\log(2d-1)}{\log(d)} \approx \frac{\log(d)+1}{\log(d)} = 1 + \frac{1}{\log(d)} \approx c$. By increasing d , the times for the top stack frame and for the base cases become closer and closer to being equal. Recall that when this happens, we must add an extra $\Theta(\log n)$ factor to allow for the $\Theta(\log n)$ levels of recursion. This gives $T(n) = \Theta(n \log n)$, which is a surprising running time for multiplication.

Section 12.4 gives a $\Theta(n \log n)$ time algorithm for multiplying two n digit numbers.

Example 12.2.3 (Strassen's Matrix Multiplication)

The next problem is to multiply two $n \times n$ matrices.

The Iterative Algorithm: The obvious iterative algorithm computes the $\langle i, j \rangle$ entry of the product matrix by multiplying the i th row of the first matrix with the j th column of the second. This requires $\Theta(n)$ scalar multiplications. Because there are n^2 such entries, the total time is $\Theta(n^3)$.

The Straightforward Divide-and-Conquer Algorithm: When designing a divide-and-conquer algorithm, the first step is to divide these two matrices into four submatrices each. Multiplying these symbolically gives the following:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae + bf & ag + bh \\ ce + df & cg + dh \end{pmatrix}.$$

Computing the four $\frac{n}{2} \times \frac{n}{2}$ submatrices in this product in this way requires recursively multiplying eight pairs of $\frac{n}{2} \times \frac{n}{2}$ matrices. The total computation time is given by the recurrence relation $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n(\log 8)(\log 2)) = \Theta(n^3)$. This is no faster than the standard iterative algorithm.

Reducing the Number of Recursions: Strassen found a way of computing the four $\frac{n}{2} \times \frac{n}{2}$ submatrices in this product using only seven such recursive calls. This gives $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n(\log 7)(\log 2)) = \Theta(n^{2.8073})$. I will not include the details of the algorithm.

Exercise 12.2.1 (See solution in Exercise Solutions) Recursive GCD.

1. Write a recursive program to find the GCD of two numbers. The program should mirror the iterative algorithm found in Chapter 6.
2. Rewrite this recursive algorithm to solve the following more general problem. The input still consists of two integers, a and b . The output consists of three integers, g, u , and v , such that $ua + vb = g = \text{GCD}(a, b)$. For example, on $a = 25$ and $b = 15$ the algorithm outputs $\langle 5, 2, -3 \rangle$, because $2 \times 25 - 3 \times 15 = 50 - 45 = 5 = \text{GCD}(25, 15)$. Provide both a paragraph containing the friend's explanation of the algorithm, and the recursive code.
3. Write an algorithm for the following problem. The input consist of three integers a , b , and w . Assume that you live in a country that has two types of coins, one worth a dollars and the other b dollars. Both you and the storekeeper have a pocket full

of each. You must pay them w dollars. You can give them any number of coins, and they may give you change with any number of coins. Your algorithm must determine whether or not this is possible and, if so, describe some way of doing it (not necessarily the optimal way). [Hint: Compute $GCD(a, b)$, and use the three values g, u , and v . Consider the two cases when g divides w and when it does not. (If you want to find the optimal number of coins, basically you change a solution by using the fact that $(\frac{b}{g}) \cdot a - (\frac{a}{g}) \cdot b = 0$.)]

4. Designing an algorithm that, when given a prime p and an integer $x \in [1, p-1]$, outputs an *inverse* y such that $x \cdot y \equiv_{\text{mod } p} 1$. [Hint: First show that $GCD(p, x) = 1$. Then compute $GCD(p, x)$ and use the values u and v . Proving that every x has such an inverse proves that the integers modulo a prime form a field. Note that the integers mod n is not a field when n is not prime, because if $a \cdot b = n$, then $a \cdot b = 0 \text{ mod } n$. In a field this should only be true if a or b are zero.]
5. Use this algorithm to find that the inverse of 20 in Z_{67} is $-10 = 57$ because $20 * (-10) = -200 = 1 - 3 * 67 = 1 - 201$.

12.3 Ackermann's Function

If you are wondering just how slowly a program can run, consider the algorithm below. Assume the input parameters n and k are natural numbers.

Algorithm

```

algorithm  $A(k, n)$ 
  if  $(k = 0)$  then
    return  $(n + 1 + 1)$ 
  else
    if  $(n = 0)$  then
      if  $(k = 1)$  then
        return  $(0)$ 
      else
        return  $(1)$ 
    else
      return  $(A(k - 1, A(k, n - 1)))$ 
    end if
  end if
end algorithm

```

Recurrence Relation: Let $T_k(n)$ denote the value returned by $A(k, n)$. This gives $T_0(n) = 2 + n$, $T_1(0) = 0$, $T_k(0) = 1$ for $k \geq 2$, and $T_k(n) = T_{k-1}(T_k(n-1))$ for $k > 0$ and $n > 0$.

Solving

$$\begin{aligned}
 T_0(n) &= 2 + n. \\
 T_1(n) &= T_0(T_1(n-1)) = 2 + T_1(n-1) = 4 + T_1(n-2) \\
 &= 2i + T_1(n-i) = 2n + T_1(0) = 2n.
 \end{aligned}$$

$$T_2(n) = T_1(T_2(n-1)) = 2 \cdot T_2(n-1) = 2^2 \cdot T_2(n-2) = 2^i \cdot T_2(n-i) = 2^n \cdot T_2(0) = 2^n.$$

$$T_3(n) = T_2(T_3(n-1)) = 2^{T_3(n-1)} = 2^{2^{T_3(n-2)}} = \left[\underbrace{2^{2^{2^{\dots^2}}}}_i \right]^{T_3(n-i)} = \left[\underbrace{2^{2^{2^{\dots^2}}}}_n \right]^{T_3(0)} = \underbrace{2^{2^{2^{\dots^2}}}}_n.$$

$$T_4(0) = 1. T_4(1) = T_3(T_4(0)) = T_3(1) = \underbrace{2^{2^{2^{\dots^2}}}}_1 = 2.$$

$$T_4(2) = T_3(T_4(1)) = T_3(2) = \underbrace{2^{2^{2^{\dots^2}}}}_2 = 2^2 = 4.$$

$$T_4(3) = T_3(T_4(2)) = T_3(4) = \underbrace{2^{2^{2^{\dots^2}}}}_4 = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65,536.$$

Note that

$$\underbrace{2^{2^{2^{\dots^2}}}}_5 = 2^{65,536} \approx 10^{21,706},$$

while the number of atoms in the universe is less than 10^{100} . We have

$$T_4(4) = T_3(T_4(3)) = T_3(65,536) = \underbrace{2^{2^{2^{\dots^2}}}}_{65,536}.$$

Ackermann's function is defined to be $A(n) = T_n(n)$. We see that $A(4)$ is bigger than any number in the natural world. $A(5)$ is unimaginable.

Running Time: The only way that the program builds up a big number is by continually incrementing it by one. Hence, the number of times one is added is at least as huge as the value $T_k(n)$ returned.

Crashing: Programs can stop at run time because of (1) overflow in an integer value; (2) running out of memory; (3) running out of time. Which is likely to happen first? If the machine's integers are 32 bits, then they hold a value that is about 10^{10} . Incrementing up to this value will take a long time. However, much worse than this, each two increments need another recursive call creating a stack of about this many recursive stack frames. The machine is bound to run out of memory first.

Exercise 12.3.1 Design the algorithm and compute the running time when $d = 3$.

12.4 Fast Fourier Transformations

Fourier Transformations are hugely important, not only in signal processing but also in so many applications. The input is some signal with strength $F(t)$ at each time $t \in \{0, \dots, n-1\}$. The goal is to decompose this signal into the linear combination of cosine and sine functions of n different frequencies $f \in \{0, \dots, n-1\}$. In general, such transformations require time $O(n^2)$, but we will now define an amazing algorithm *Fast Fourier Transformations* (FFT) that does it in time $O(n \log n)$. Strangely, we will start by developing an algorithm for evaluating a polynomial $P(x)$ of degree $n-1$ at n values

for x . Then we will make it work for sine and cosine functions. We will conclude with an amazing algorithm that is able to multiply two n digit numbers in time $O(n \log n)$.

My input: A vector of coefficients $\langle a_0, a_1, \dots, a_{n-1} \rangle$ and a value x . Assume n is a power of 2. Oddly we will index a_f with frequency f .

My output: The evaluation of the polynomial $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$ on the given value for x .

Interpolations: The inverse of this starts with the value of an unknown polynomial $P(x)$ at n values for x and *interpolates* to determine the coefficients of the unique polynomial going through these points.

Splitting the Input: We need to split the input into two halves to give to our friends.

$$\begin{aligned} P(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \\ &= a_0 + a_2x^2 + a_4x^4 + \dots + a_{n-2}x^{n-2} \\ &\quad + x(a_1 + a_3x^2 + a_5x^4 + \dots + a_{n-1}x^{n-2}) \\ &= P_{\text{even}}(x^2) + x(P_{\text{odd}}(x^2)), \end{aligned}$$

where

$$\begin{aligned} P_{\text{even}}(z) &= a_0 + a_2z + a_4z^2 + \dots + a_{n-2}z^{\frac{n}{2}-1}, \\ P_{\text{odd}}(z) &= a_1 + a_3z + a_5z^2 + \dots + a_{n-1}z^{\frac{n}{2}-1}. \end{aligned}$$

Recursion: The “even” friend is given the even coefficients $\langle a_0, a_2, \dots, a_{n-2} \rangle$ and the value x^2 . The “odd” friend is given the odd coefficients $\langle a_1, a_3, \dots, a_{n-1} \rangle$ and the value x^2 . They give me $P_{\text{even}}(x^2)$ and $P_{\text{odd}}(x^2)$ and I return $P(x) = P_{\text{even}}(x^2) + x(P_{\text{odd}}(x^2))$. The running time is $T(n) = 2T(\frac{n}{2}) + O(1) = O(n)$, which is not exciting because we could have done that with a simpler algorithm.

n Values of x : Let’s redo the problem except we are given n different values of x , namely, $\langle x_0, x_1, \dots, x_{n-1} \rangle$. Oddly, we will index x_t with time t . Repeating the previous algorithm n times takes $O(n^2)$ time. Consider the obvious modification to the recursive algorithm in which we are given n values of a_f and n values of x_t and we give each of our two friends $\frac{n}{2}$ values of a_f and n values of x_t . When doing recursion, it is important to be clear about your definition of the “size” of your input and to be sure that it gets smaller. Because the number of x_t values does not decrease, their instance is not really “smaller.” We can fix this problem by having four friends each given half the a s and half the x s, but this would also take $T(n) = 4T(\frac{n}{2}) + O(n) = O(n^2)$. Somehow, we need to decrease the number of friends to two so that the running time is only $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

Special Values: Recall that we do not give the values $\{x_t\}$ to our friends, but the squares $\{(x_t)^2\}$ of these values. Suppose our set of n values of x was $\{\dots, -3, -2, -1, 1, 2, 3, \dots\}$. We give the friends the squares of these numbers, i.e., $\{\dots, 9, 4, 1, 1, 4, 9, \dots\}$. Note the set collapses to $\{1, 4, 9, \dots\}$, which has half as many

elements. This means we can give each of our two friends only $\frac{n}{2}$ values. The problem is that if our friends get the set of values $\{1, 4, 9, \dots\}$, then their friends get the set of values $\{1, 16, 81, \dots\}$, which did not collapse to half as many values. Let's say a set of values is *special* if when you square each value you get half as many values and these values are also special so that when you square them you get a quarter as many values, and so on. For example, the set $\{-i, -1, 1, i\}$, where $i = \sqrt{-1}$, is special because when you square each value, the set collapses to $\{-1, 1\}$, which is special because when you square each value, it collapses to $\{1\}$.

n th Roots of Unity Are Special: ω is said to be an n^{th} root of unity, if $\omega^n = 1$. We will see that complex numbers have n of them. A common but complicated way to define the first one is $\omega = e^{\frac{2\pi i}{n}}$. (See below for a discussion about this.) Then the set consists of $\{1, \omega, \omega^2, \omega^3, \dots, \omega^{n-1}\}$. Each of these is an n^{th} root of unity because for each $t \in \{0, n-1\}$, we have that $(\omega^t)^n = (\omega^n)^t = (1)^t = 1$. When n is a power of 2, then this set is special because when you square one of them, it becomes an $\frac{n}{2}^{\text{th}}$ root of unity because $(\omega^2)^{\frac{n}{2}} = \omega^n = 1$. We know that there are half as many of these because squaring ω^t gives $(\omega^t)^2 = \omega^{2t}$, which has an even exponent. If $2t \geq n$, then this wraps around to being $\omega^{2t-n} = \omega^{2t} \cdot \omega^n = \omega^{2t}$, which also has an even exponent and only half of the exponents in $\{0, 1, \dots, n-1\}$ are even.

Code

Algorithm $P\text{-FFT}(A, \omega, n)$

```

⟨ pre-cond ⟩:  $A = \langle a_0, a_1, \dots, a_{n-1} \rangle$  defining polynomial
 $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$ ,  $\omega = e^{\frac{2\pi i}{n}}$ , i.e., an  $n^{\text{th}}$  root of unity,
defining the set  $X = \langle \omega^0, \omega^1, \dots, \omega^{n-1} \rangle$ .
⟨ post-cond ⟩:  $Y = \langle y_0, y_1, \dots, y_{n-1} \rangle$  where for each  $t$ ,  $y_t = P(\omega^t)$ .
begin
    if(  $n=1$  )
        return(  $\langle a_0 \rangle$  )
    else
        % Separate even and odd indices
         $A_{\text{even}} = \langle a_0, a_2, \dots, a_{n-2} \rangle$ 
         $A_{\text{odd}} = \langle a_1, a_3, \dots, a_{n-1} \rangle$ 
        % Recurse
         $Y_{\text{even}} = P\text{-FFT}(A_{\text{even}}, \omega^2, \frac{n}{2})$ 
         $Y_{\text{odd}} = P\text{-FFT}(A_{\text{odd}}, \omega^2, \frac{n}{2})$ 
        % Combining
        For  $t = 0$  to  $n-1$ 
            %  $P(x) = P_{\text{even}}(x^2) + x(P_{\text{odd}}(x^2))$ .
            %  $(x_t)^2 = (\omega^t)^2 = \omega^{2t}$  or  $\omega^{2t-n}$ 
            % which is the  $t'^{\text{th}}$  of the  $(\frac{n}{2})^{\text{th}}$  roots of unity
            % where  $t' = \frac{2t}{2} = t$  or  $\frac{2t-n}{2}$ 
             $Y[t] = Y_{\text{even}}[t'] + \omega^t \cdot Y_{\text{odd}}[t']$ 
        return( $Y$ )
    end if

```

Complex Numbers: Complex numbers are very useful.

Practical: If you solve some equation and get a complex number $x + iy$, then often the imaginary part represents a practical concept. For example, if it measures the voltage of a system, then y would be the phase's shift between the voltage and the current.

Rotating the Plane: Defining i to be $\sqrt{-1}$ seems a bit arbitrary. Let's define it to be something different and from that definition prove that $i^2 = -1$. The real numbers lie in a one-dimensional line. Let's define the *complex numbers* to be the set of points $\langle x, y \rangle$ in the two-dimensional Euclidean plane. The x -axis will be the real numbers, namely the point $\langle 0, 0 \rangle$ will be zero, moving one to the right, $\langle 1, 0 \rangle$ will be one, and to the left, $\langle -1, 0 \rangle$ will be minus one. Moving one up to $\langle 0, 1 \rangle$ should be important, so let's call it i .

Since the point $\langle x, y \rangle$ in Euclidean notation is x times the vector $\langle 1, 0 \rangle$ plus y times the vector $\langle 0, 1 \rangle$, this complex number can be written as $x + iy$.

Addition is easy because $\langle x_1, y_1 \rangle + \langle x_2, y_2 \rangle = \langle (x_1 + x_2), (y_1 + y_2) \rangle$. However, we need to define multiplication. Imagine that the numbers are written both on the table and on a rubber sheet lying on top. To multiply 3 and 7, fix the rubber zero $\langle 0, 0 \rangle$ to the table $\langle 0, 0 \rangle$. Pick up the rubber one at $\langle 1, 0 \rangle$ and pull it to the table three at $\langle 3, 0 \rangle$.

Being a magic rubber sheet, this stretches it uniformly in all directions by a factor of three. This effectively multiplies each value by three. The rubber seven at $\langle 7, 0 \rangle$ moves to table twenty-one at $\langle 21, 0 \rangle$. This is how you determine that $3 \times 7 = 21$. Similarly, the rubber i at $\langle 0, 1 \rangle$ moves to table $3i$ at $\langle 0, 3i \rangle$.

Let's do the same to multiply i by i . Pick up the rubber one at $\langle 1, 0 \rangle$ and move it to the table i at $\langle 0, 1 \rangle$. This does not stretch the rubber sheet at all but rotates it 90 degrees counter clockwise. This effectively multiplies each value by i . The rubber i value at $\langle 0, 1 \rangle$ moves to table negative one at $\langle -1, 0 \rangle$. And hence as we suspected $i \times i = -1$.

Let's do a third example. Consider the point in the plain that has distance r from the origin and angle θ counter clockwise from the x -axis, i.e., polar notation. To multiply by it, pick up the rubber one at $\langle 1, 0 \rangle$ and move it to the table at this point. This rotates the rubber sheet by angle θ clockwise and stretches it in all directions by a factor of r . Hence, one multiplies two complex numbers by multiplying their distance r from the origin and adding their rotation angles θ . In 1740, Euler represented this complex number as $re^{i\theta}$ because oddly enough $[r_1 e^{i\theta_1}] \times [r_2 e^{i\theta_2}] = (r_1 r_2) e^{i(\theta_1 + \theta_2)}$.

Polar and Euler: This gives three representations of the same complex number, i.e., $x + iy = r \cos(\theta) + ir \sin(\theta) = re^{i\theta}$, where $x = r \cos(\theta)$, $y = r \sin(\theta)$, $r = \sqrt{x^2 + y^2}$, and $\theta = \cos^{-1}(\frac{x}{r})$. The reason for introducing the $e = 2.71 \dots$ is so that derivatives of these representations are consistent, namely that the derivative of $r \cos(\theta) + ir \sin(\theta)$ with respect to θ is $r(-\sin(\theta)) + ir \cos(\theta) = i \cdot [r \cos(\theta) + ir \sin(\theta)]$. Similarly, the derivative of $re^{i\theta}$ is $i \cdot [re^{i\theta}]$. If you do

this twice, you get $f''(\theta) = i^2 f(\theta) = -f(\theta)$, which is needed in the spring in [Example 1.4.6](#). The function $f(t) = e^{t+it}$ spirals moving away from the origin at a rate of $r=e^t$ and rotating at a constant rate from the e^{it} .

n^{th} Roots of Unity: Place n points evenly spaced around the circle of radius one centered at the origin. This puts an angle of $\Theta = \frac{2\pi}{n}$ radians between each pair. The first point will be one at $\langle 1, 0 \rangle$. The second one will be denoted ω . Each time you multiply by ω rotates the rubber sheet by an angle of Θ . Hence, for $t \in \{0, n-1\}$, we have that ω^t is the t^{th} point at an angle $t \cdot \Theta = t \cdot \frac{2\pi}{n}$. Multiplying by ω n times gives ω^n at an angle $n \cdot \Theta = n \cdot \frac{2\pi}{n} = 2\pi$, which is all the way around the circle back to one.

t^{th} Value of x and f^{th} Term/Frequency: Recall that the above algorithm evaluates the polynomial $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$ at $x = \omega^t$. Consider the f^{th} term x^f . Plugging in $x = \omega^t$ gives $x^f = (\omega^t)^f = \omega^{tf}$. We have seen that the exponent tf rotates this point by an angle of $tf \cdot \Theta = tf \cdot \frac{2\pi}{n}$. This point is located in the plane at $\langle x, y \rangle = \langle \cos(tf \cdot \frac{2\pi}{n}), \sin(tf \cdot \frac{2\pi}{n}) \rangle$. Thinking of the integer one as the vector from the origin to $\langle 1, 0 \rangle$ and i as the vector from the origin to $\langle 0, 1 \rangle$, we can do vector algebra to get that $\omega^{tf} = \cos(tf \cdot \frac{2\pi}{n}) + i \cdot \sin(tf \cdot \frac{2\pi}{n})$. Let us define the signal $F(t)$ to be what you get when you plug in $x = \omega^t$ into the polynomial $P(x) = \sum_{f=0}^{n-1} a_f x^f$, namely, $F(t) = P(\omega^t) = \sum_{f=0}^{n-1} a_f (\omega^t)^f = \sum_{f=0}^{n-1} a_f [\cos(tf \cdot \frac{2\pi}{n}) + i \cdot \sin(tf \cdot \frac{2\pi}{n})]$. Note that $\sin(t \cdot \frac{2f\pi}{n})$ is a sine function of t with frequency f because as time t increases from 0 to n , the angle increases from 0 to $f \cdot 2\pi$, which is f times around the circle. Hence, $F(t)$ is a linear combination of cosine and sine functions of n different frequencies $f \in \{0, \dots, n-1\}$. The only restriction (which we will ignore) is that it gives $\sin(t \cdot \frac{2f\pi}{n})$ the coefficient $a_f \cdot i$ that is i times that given to $\cos(t \cdot \frac{2f\pi}{n})$.

Inverse Sin-Cos-FFT Our original goal had its input be some signal $F(t)$ and its output be this cos-sin-decomposition. It seems that what we just accomplished is the inverse task, namely, given as input a vector of coefficients $\langle a_0, a_1, \dots, a_{n-1} \rangle$ representing the signal function $F(t) = \sum_{f=0}^{n-1} a_f [\cos(t \cdot \frac{2f\pi}{n}) + i \cdot \sin(t \cdot \frac{2f\pi}{n})]$, we evaluate $F(t)$ at values $t \in \{0, \dots, n-1\}$.

FFT(FFT($F(t)$)) = $n \cdot F(t)$: We saw a key piece of the algorithm is the term ω^{tf} , which is symmetric in time t and frequency f . With a little work, one can see that from this it follows that the algorithm for forward FFT is identical to that for the inverse FFT, except for multiplying everything by n .

Multiplying: Amazingly, one can multiply two n digit numbers in $O(n \log n)$ time using FFT. Let A be an integer with digits $a_{n-1}, \dots, a_1 a_0$. Note that $A = a_{n-1} \cdot 10^{n-1} + \dots + a_1 \cdot 10 + a_0$. Replacing the 10 with a variable x gives the polynomial $P_A(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$. Doing an FFT will evaluate this at the n^{th} roots of unity, ω^t , giving $P_A(\omega^t)$. Doing the same for a second integer B gives the values $P_B(\omega^t)$. Multiplying two polynomials is complicated and takes $O(n^2)$ time, namely, $P_{A \times B}(x) = [P_A \times P_B](x)$

$$= [a_{n-1}x^{n-1} + \cdots + a_1x + a_0] \times [b_{n-1}x^{n-1} + \cdots + b_1x + b_0] = \sum_f [\sum_i a_i b_{f-i}] x^f.$$
 However, the evaluations $[P_A \times P_B](\omega^t)$ are done in time $O(1)$, each with a simple multiplication, namely, $[P_A \times P_B](\omega^t) = P_A(\omega^t) \times P_B(\omega^t)$. Knowing these evaluations, we can do the inverse FFT to recover the coefficients of the polynomial $P_{A \times B}(x)$. Plugging in $x = 10$ gives the integer product $A \times B$.

12.5 Exercise

Exercise 12.5.1 Review the problem of iterative cake cutting ([Section 2.3](#)). You are now to write a recursive algorithm for the same problem. You will, of course, need to make the pre- and postconditions more general so that when you recurse, your subinstances meet the preconditions. As in moving from insertion sort to merge sort, you need to make the algorithm faster by cutting the problem in half.

1. You will need to generalize the problem so that the subinstance you would like your friend to solve is a legal instance according to the preconditions and so that the postconditions state the task you would like them to solve. Make the new problem, however, natural. Do not, for example, pass the number n of players in the original problem or the level of recursion. The input should simply be a set of players and a subinterval of cake. The postcondition should state the requirements on how this subinterval is to be divided among these players. To make the problem easier, assume that the number of players is $n = 2^i$ for some integer i .
2. Give recursive pseudocode for this algorithm. As a big hint, toward designing a recursive algorithm, I will tell you the first things that the algorithm does. Each player specifies where they would cut if they were to cut the cake in half. Then one of these spots is chosen. You need to decide which one and how to create two subinstances from this.
3. Prove that if your instance meets the preconditions, then your two subinstances also meet the preconditions.
4. Prove that if your friend's solutions meet the postconditions, then your solution meets the postcondition.
5. Prove that your solution for the base case meets the postconditions.
6. Give and solve the recurrence relation for the running time of this algorithm.
7. Now suppose that n is not 2^i for any integer i . How would we change the algorithm so that it handles the case when n is odd? I have two solutions: one that modifies the recursive algorithm directly, and one that combines the iterative algorithm and the recursive algorithm. You only need to do one of the two (as long as it works and does not increase the BigOh of the running time.)

13 Recursion on Trees

One key application of recursive algorithms is to perform actions on trees, because trees themselves have a recursive definition. Terminology for trees is summarized in the following table:

Term	Definition
Root	Node at the top
<i>RootInfo(tree)</i>	The information stored at the root node
Child of node u	One of the nodes just under node u
Parent of node u	The unique node immediately above node u
Siblings	Nodes with same parent
Ancestors of node u	The nodes on the unique path from the root to the node u
Descendants of node u	All the nodes below node u
Leaf	A node with no children
Height of tree	The maximum level. Some definitions say that a tree with a single node has height 0, others say height 1. It depends on whether you count nodes or edges.
Depth of node u	The number of nodes (or edges) on the path from the root to u .
Binary tree	Each node has at most two children. Each of these is designated as either the right child or the left child.
<i>leftSub(tree)</i>	Left subtree of root
<i>rightSub(tree)</i>	Right subtree of root

Recursive Definition of Tree: A tree is either:

- an empty tree (zero nodes), or
- a root node with some subtrees as children.

A *binary tree* is a special kind of tree where each node either has a right and a left subtree or is a leaf ([Figure 13.1](#)).

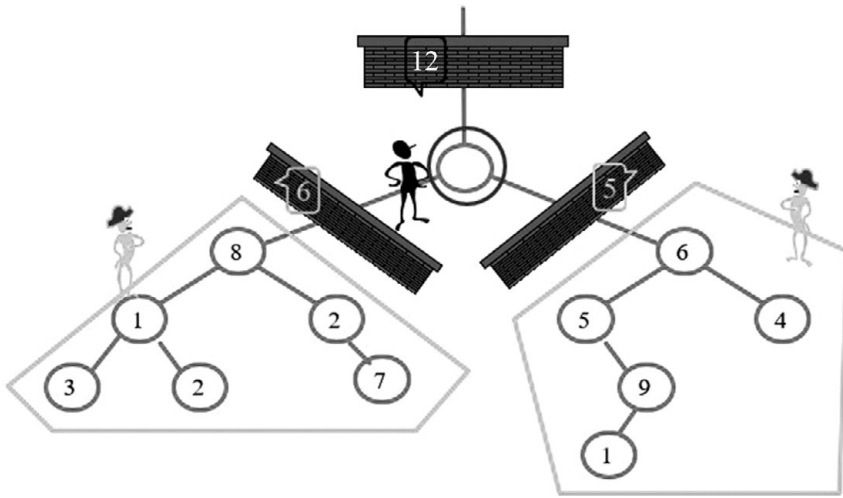


Fig. 13.1 Recursively computing the number of nodes in a binary tree.

Example 13.0.1 (Number of Nodes in a Binary Tree)

We will now develop a recursive algorithm that will compute the number of nodes in a binary tree.

Specifications

Preconditions: The input is any binary tree. Trees with an empty subtree are valid trees. So are trees consisting of a single node and the empty tree.

Postconditions: The output is the number of nodes in the tree.

Size: The *size* of an instance is the number of nodes in it.

General Input: Consider a large binary tree with two complete subtrees.

Magic: We assume that by *magic* a friend is able to count the nodes in any tree that is strictly smaller than ours.

Subinstances: The subinstances of our instance tree will be the tree's left and its right subtree. These are valid instances that are strictly smaller than ours because the root (and the other subtree) have been removed.

Subsolutions: We ask one friend to recursively count the number of nodes in the left subtree and another friend to do so in the right subtree.

Solution: The number of nodes in our tree is the number in its left subtree plus the number in its right subtree plus one for the root.

Other Instances: Suppose the instance is a tree with the right subtree missing. Surprisingly, the algorithm still works. The number of nodes in our tree's right subtree is zero. This is the answer that our friend will return. Hence, the algorithm returns the number in the left subtree plus zero plus one for the root. This is the

correct answer. Similarly, the algorithm works when the left subtree is empty or when the instance consists of a single leaf node.

The remaining instance is the empty tree. The algorithm does not work in this case, because it does not have any subtrees. Hence, the algorithm can handle all trees except the empty tree with one piece of code.

Base Cases: The empty tree is sufficiently small that we can solve it in a brute force way. The number of nodes in it is zero.

The Tree of Stack Frames: There is one recursive stack frame for each node in the tree, and the tree of stack frames directly mirrors the structure of the tree.

Running Time: There is one recursive stack frame for each node in the tree (and one for each empty tree hanging off the leaves). Generally, each stack frame does a constant amount of work. Hence, the total time is linear in the size of the tree, i.e., $T(n) = \Theta(n)$. Proved another way, let c be the amount of work done in each stack frame. Then the recurrence relation is $T(n) = T(n_{\text{left}}) + T(n_{\text{right}}) + c$. Plugging in the guess $T(n) = c \cdot n$, gives $c \cdot n = c \cdot n_{\text{left}} + c \cdot n_{\text{right}} + c$. Dividing by c gives $n = n_{\text{left}} + n_{\text{right}} + 1$, which is correct because this is the count of the number of nodes.

Code:

algorithm *NumberNodes(tree)*

<pre-cond>: *tree* is a binary tree.

<post-cond>: Returns the number of nodes in the tree.

begin

 if(*tree* = *emptyTree*) then

 result(0)

 else

 result(*NumberNodes*(*leftSub*(*tree*))

 + *NumberNodes*(*rightSub*(*tree*)) + 1)

 end if

end algorithm

Data Structure: The variable *tree* passed to a recursive program is not of type *tree object*, but it points to the root of the tree. It is thought of in two different ways: on one hand, it can be thought of as the single node it is pointing to and it can be thought of as subtree rooted at the node pointed at. If each node object has fields *left* and *right* pointing to the node's left and right children, then *tree.left* can be thought both as the root node's left child node and as the left subtree of *tree*. Instead of *tree.left*, the pseudocode in this book uses *leftSub(tree)* to denote the left subtree of the tree *tree*.

Problem with the Single-Node Base Case: Many people are tempted to use trees with a single node as the base case. A minor problem with this is that it means that the routine no longer works for the empty tree, i.e., the tree with zero nodes. A bigger problem is

that the routine no longer works for trees that contain a node with a left child but no right child, or vice versa. This tree is not a base case, because it has more than one node. However, when the routine recurses on the right subtree, the new subinstance consists of the empty tree. The routine, however, no longer works for this tree. See [Exercise 13.2.1](#) for more on this.

Answer for the Empty Tree: A common mistake is to provide the wrong answer for the empty tree. When in doubt as to what answer should be given for the empty tree, consider an instance with the left or right subtree empty. What answer do you need to receive from the empty tree to make this tree's answer correct?

Height: For example, a tree with one node can either be defined to have height 0 or height 1. It is your choice. However, if you say that it has height 0, then be careful when defining the height of the empty tree.

Definition Binary Search Tree: Another example is that people often say that the empty tree is not a binary search tree ([Section 3.1](#)). However, it is. A binary tree fails to be a binary search tree when certain relationships between the nodes exist. Because the empty tree has no nodes, none of these violating conditions exist. Hence, by default it is a binary search tree.

Max: What is the maximum value within an empty list of values? One might think 0 or ∞ . However, a better answer is $-\infty$. When adding a new value, one uses the code *newMax* = max(*oldMax*, *newValue*). Starting with *oldMax* = $-\infty$ gives the correct answer when the first value is added.

Example 13.0.2 (Many Children)

We will again count the number of nodes, but in a tree that has an arbitrary number of children. Here *tree.numberChildren* denotes the number of children of the root of tree *tree*.

Solution: The number of nodes in our tree is the sum of those in each of its subtrees plus one for the root.

Code:

```

algorithm NumberNodes(tree)
  <pre-cond>: tree is an arbitrary rooted tree (not empty).
  <post-cond>: Returns the number of nodes in the tree.
  begin
    n = 0
    loop i = 1 to tree.numberChildren
      n = n + NumberNodes(tree.child[i])
    end loop
    result( n + 1 )
  end algorithm

```

Base Case: It is surprising that this code does not seem to have a base case. Unlike the code for binary trees, it does not handle the empty tree. So how does this code

stop? If the root of the tree has at least one child, then the code legitimately recurses. If the root has no children, then this tree is itself a leaf node. This case is effectively a base case because the code loops and recurses zero times and correctly return the answer one.

Running Time: If a node in the tree has r children, then its stack frame does $O(r)$ work, which is not constant time. The total time is still $O(n)$ because this algorithm does a constant amount of time per edge. The number of edges in a tree is always one less than the number of nodes, because even though a node can have many edges going down from it, each node (except the root) has exactly one edge going up.

Communication: I have been passed a tree or a subtree. I have some computation problems to solve. I will have a recursive friend for each of my subtrees. Each will solve the same problem. I am one stack frame. Being lazy, I will only consider the root node of my tree. We combine these to get our answer.

Running Time: We want the running time to be constant $O(1)$ per node. As one stack frame, considering only one node, I personally can only spend a constant amount of time. Hence, I do not have time to look at my children's subtrees. I will entrust them to my friends.

Walls: Assume that there is a wall between me and my boss and another between me and each of my friends though which only a constant $O(1)$ amount of information can be passed. See [Figure 13.1](#).

Precondition: I know nothing about the outside world (e.g., whether my tree is part of a bigger tree) except via what my boss tells me as specified by the precondition. Generally, the precondition only states that a tree is being passed to the routine. If I need more information about the outside world in order to solve the required question about my subtree, I can get it from my boss simply having the designer change the precondition. For example, I might want to know if I am the left or the right child of my parent. Note, however, my friends have the same precondition. Hence, I must tell each of them this new information.

Postcondition: What I know about the subtrees below me is only what friends tell me via the postcondition. The postcondition for the problem dictates the required answer. Because my friends have the same postcondition, I will learn this information for each of my subtrees from them. If I need more information about these subtrees in order for me to solve my required question about my tree, then I can get it from my friends simply by having the designer change the postcondition. For example, if what I need to know about my tree is whether it is balanced, then I might ask each of my friends for the depth of their subtrees. However, if postconditions change, then I must fill this contract with my boss too.

Too Much Time: In contrast, suppose you already have a routine that finds the max depth of a tree and you innocently use it to find that about your

subtrees, i.e., a whole separate program is called on your subtrees. [Example 13.0.3](#) discusses how the running time of your routine may blow up to $O(n^2)$.

Example 13.0.3 (Is the Tree a Binary Search Tree?)

The required algorithm returns whether or not the given tree is a binary search tree (BST).

An Inefficient Algorithm:

algorithm *IsBSTtree(tree)*

(pre-cond): *tree* is a binary tree.

(post-cond): The output indicates whether it is a binary search tree.

```
begin
  if(tree = emptyTree) then
    return Yes
  else if(IsBSTtree(leftSub(tree)) and IsBSTtree(rightSub(tree))
    and Max(leftSub(tree)) ≤ rootKey(tree) ≤ Min(rightSub(tree))) then
    return Yes
  else
    return No
  end if
end algorithm
```

Running Time: For each node in the input tree, the above algorithm computes the minimum or the maximum value in the node's left and right subtrees. Though these operations are relatively fast for binary search trees, performing them for each node increases the time complexity of the algorithm, because each node may be traversed by either the *Min* or the *Max* routine many times. Suppose, for example, that the input tree is completely unbalanced, i.e., a single path. For node i , computing the max of its subtree involves traversing to the bottom of the path and takes time $n - i$. Hence, the total running time is $T(n) = \sum_{i=1..n} (n - i) = \Theta(n^2)$. This is far too slow.

13.1 Tree Traversals

A task one needs to be able to perform on a binary tree is to traverse it, visiting each node once, in one of three defined orders. Before one becomes familiar with recursive programs, one tends to think about computation iteratively, "I visit this node first, then this one, then this one, and so on." Each iteration, the program says "I just visited this node, so now let me find the next node to visit." Surprisingly, such a computation is hard to code. The reason is that binary trees by their very nature have a recursive structure. At the end of this section, I include code that traverses a binary tree in an iterative way, but only to convince you that this is much harder than doing it recursively and should be avoided.

Recursion, on the other hand, provides a very easy and slick algorithm for traversing a binary tree. Such a tree is composed of three parts. There is the root node, its left subtree, and its right subtree. You, being lazy, get one friend to traverse the left and another to traverse the right. You, feeling that you need to do some work yourself, visit the root. The order in which the three of you perform your tasks dictates the order in which the nodes get visited. The three classic orders to visit the nodes of a binary tree are *prefix*, *infix*, and *postfix*, in which the root is visited before, between, or after its left and right subtrees are visited.

algorithm *PreFix(tree)*

<pre-cond>: *tree* is a binary tree.
<post-cond>: Visits the nodes
 in prefix order.

```
begin
    if(tree ≠ emptyTree) then
        put rootInfo(tree)
        PreFix(leftSub(tree))
        PreFix(rightSub(tree))
    end if
end algorithm
```

algorithm *InFix(tree)*

<pre-cond>: *tree* is a binary tree.
<post-cond>: Visits the nodes
 in infix order.

```
begin
    if(tree ≠ emptyTree) then
        InFix(leftSub(tree))
        put rootInfo(tree)
        InFix(rightSub(tree))
    end if
end algorithm
```

algorithm *PostFix(tree)*

<pre-cond>: *tree* is a binary tree.
<post-cond>: Visits the nodes
 in postfix order.

```
begin
    if(tree ≠ emptyTree) then
        PostFix(leftSub(tree))
        PostFix(rightSub(tree))
        put rootInfo(tree)
    end if
end algorithm
```

The following order is produced if you tracing out these computations on the two trees displayed in [Figure 13.2](#).

These three orders have different applications. In math, the typical order to put operators is infix notation as in $3 + 4 * 7$. However, printed like this produces the wrong order of operations. What is required is $(3 + 4) * 7$. Using pre- and postfix order, the precedence of the operators is correctly determined even without brackets. For this reason, Hewlett Packard's first calculator used postfix notations $3\ 4\ +\ 7\ *$. It was called reverse Polish notation after Jan Łukasiewicz, who developed it in the 1920s. These were a bit of a pain, but luckily for me, the technology improved enough by the time

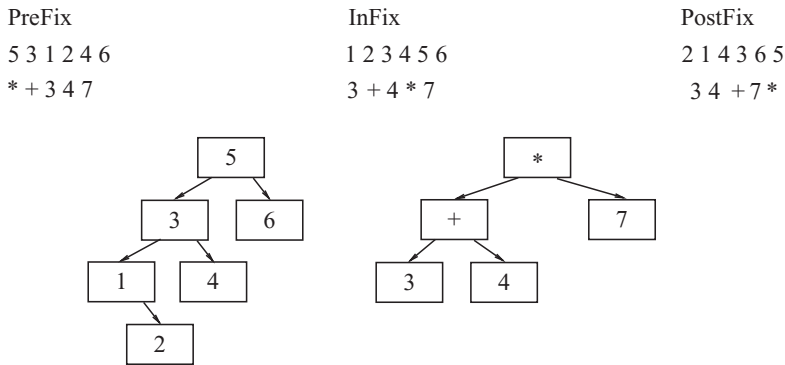


Fig. 13.2 Three traversal orders through a binary tree.

I started high school in 1977 so that I did not need to use one. However, such calculators still have their partisans, and Hewlett Packard still makes them.

PreFix visits the nodes in the same order that a depth-first search finds the nodes. See [Section 20.4](#) for the iterative algorithm for doing depth-first search of a more general graph, and [Section 20.5](#) for the recursive version of the algorithm.

Below is the iterative program for visiting the nodes in infix order. As said, it is needlessly complex and is included only to show you what to avoid.

algorithm *IterativeTraversal(tree)*

<pre-cond>: *tree* is a binary tree. As usual, each node has a value and pointers to the roots of its left and right subtrees. In addition, each node has a pointer to its parent.

<post-cond>: Does an infix traversal of *tree*.

begin

element = root(*tree*) % Current node in traversal

count = zero % Current count of nodes

loop

<loop-invariant>: *element* is some node in the tree, and some nodes have been visited.

if(*element* has a left child and it has not been visited) then

element = leftChild(*element*)

elseif(*element* has no left child or its left child have been visited and *element* has not been visited) then

visit *element*

elseif(*element*'s left subtree and *element* itself has been visited and *element* has a right child and it has not been visited) then

element = rightChild(*element*)

elseif(*element*'s left subtree, *element* itself, and right subtree have been visited and *element* has a parent) then

```

        element = parent(element)
    elseif( Everything has been visited and element is the root of the
           global tree) then
        exit
    end if
end loop
end algorithm

```

13.2 Simple Examples

Here is a list of problems involving binary trees.

1. Return the maximum of data fields of nodes.
2. Return the height of the tree.
3. Return the number of leaves in the tree. (A harder one.)
4. Copy the tree.

Try them first on your own. See [Figure 13.3](#).

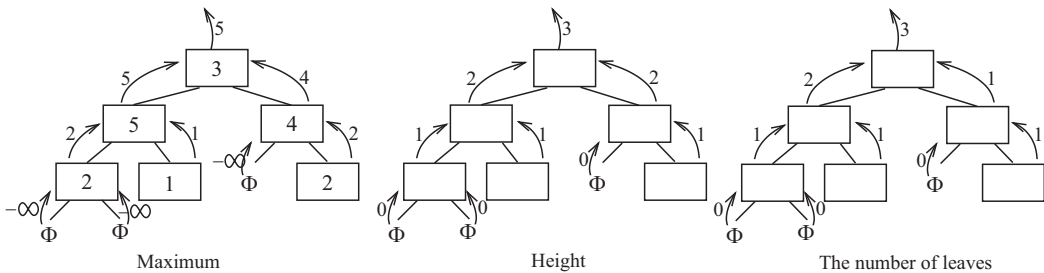


Fig. 13.3 The result returned on each subtree is provided.

Maximum: Given a binary tree, your task is to determine its maximum value. The first step is to decide how to create subinstances for your friends. As said, when the input instance is a binary tree, the most natural subinstances are its left and right subtrees. Your friends must solve the same problem that you do. Hence, assume that they provide you with the maximum value within each of these trees. Luckily, the maximum value within a tree is either the maximum on the left, the maximum on the right, or the value at the root. Our only job then is to determine which of these three is the maximum. As described in the beginning of this chapter, the maximum of the empty list is $-\infty$.

algorithm *Max(tree)*

<pre-cond> *tree* is a binary tree.

<post-cond> Returns the maximum of data fields of nodes.


```

begin
  if(tree = emptyTree) then
    result( $-\infty$ )
  else
    result(max(Max(leftSub(tree)), Max(rightSub(tree)), rootData(tree)))
  end if
end algorithm

```

Height: In this problem, your task is to find the height of your binary tree. Again, your friends can easily give you the height of your left and right subtrees. The height of your tree is determined by the deeper of its subtrees. Given their heights, you determine which is deeper and add one to take the root into account. The height of the empty tree is discussed in the beginning of this chapter.

algorithm *Height*(*tree*)

(pre-cond): *tree* is a binary tree.

(post-cond): Returns the height of the tree measured in nodes, e.g., a tree with one node has height 1.

```

begin
  if(tree = emptyTree) then
    result(0)
  else
    result(max(Height(leftSub(tree)), Height(rightSub(tree))) + 1)
  end if
end algorithm

```

Exercise 13.2.3 considers another version of this algorithm.

Number of Leaves: This problem is harder than the previous ones. We start by considering a tree with both a left and a right subtree. For this, the number of leaves in the entire tree is the sum of the numbers in the left and right subtrees. If the tree has one subtree, but the other is empty, then this same algorithm still works. If the tree is empty, then it has zero leaves. However, if this were all of the code, then the answer returned would always be zero. The case that still needs to be considered is the tree consisting of a root with no children. This root is a leaf. We need to count it.

algorithm *NumberLeaves*(*tree*)

(pre-cond): *tree* is a binary tree.

(post-cond): Returns the number of leaves in the tree.

```

begin
  if(tree = emptyTree) then
    result(0)
  else if(leftSub(tree) = emptyTree and rightSub(tree) = emptyTree) then
    result(1)
  else

```

```

        result(NumberLeaves(leftSub(tree)) + NumberLeaves(rightSub(tree)))
    end if
end algorithm

```

Copy Tree: If you want to make a copy of a tree, you might be tempted to use the code *treeCopy = tree*. However, the effect of this will only be that both the variables *treeCopy* and *tree* refer to the same tree data structure that *tree* originally did. This would be sufficient if you only want to have read access to the data structure from both variables. However, if you want to modify one of the copies, then you need a completely separate copy. To obtain this, the copy routine must allocate memory for each of the nodes in the tree, copy over the information in each node, and link the nodes together in the appropriate way. The following simple recursive algorithm, *treeCopy = Copy(tree)*, accomplishes this.

algorithm *Copy(tree)*

⟨*pre-cond*⟩: *tree* is a binary tree.

⟨*post-cond*⟩: Returns a copy of the tree.

```

begin
    if(tree = emptyTree) then
        result(emptyTree)
    else
        treeCopy = allocate memory for one node
        rootInfo(treeCopy) = rootInfo(tree)    % copy overall data in root node
        leftSub(treeCopy) = Copy(leftSub(tree))    % copy left subtree
        rightSub(treeCopy) = Copy(rightSub(tree))    % copy right subtree
        result(treeCopy)
    end if
end algorithm

```

Exercise 13.2.1 Many texts that present recursive algorithms for trees do not consider the empty tree to be a valid input instance, but by not considering empty trees the algorithm requires many more cases. Redesign the algorithm of Example 13.1 to return the number of nodes in the input tree without considering the empty tree.

Exercise 13.2.2 Develop an algorithm that returns the sum of the values within the nodes of a binary tree.

Exercise 13.2.3 We have given a recursive algorithm for finding the height of a binary tree measured in nodes, e.g., a tree with one node has height 1. Rewrite this algorithm so that the height is measured in edges, e.g., a tree with one node has height 0.

Exercise 13.2.4 If the computer system does not have garbage collection, then it is the responsibility of the programmer to deallocate the memory used by all the nodes of a tree when the tree is discarded. Develop a recursive algorithm, *Deallocate(tree)*, that accomplishes this. How much freedom is there in the order of the lines of the code?

Exercise 13.2.5 Develop an algorithm that searches for a key within a binary search tree.

13.3 Heap Sort and Priority Queues

Heap sort is a fast sorting algorithm that is easy to implement. Like quick sort, it has the advantage of being done in place in memory, whereas merge and radix counting sorts require an auxiliary array of memory to transfer the data to. I include heap sort in this chapter because it is implemented using recursion within a tree data structure.

Completely Balanced Binary Tree: We will visualize the values being sorted as stored in a binary tree that is completely balanced, i.e., every level of the tree is completely full except for the bottom level, which is filled in from the left.

Array Implementation of a Balanced Binary Tree: Because the tree always has this balanced shape, we do not have to bother with the overhead of having nodes with pointers. In actuality, the values are stored in a simple array $A[1, n]$. See Figure 13.4. The mapping between the visualized tree structure and the actual array structure is done by indexing the nodes of the tree $1, 2, 3, \dots, n$, starting with the root of the tree and filling each level in from left to right.

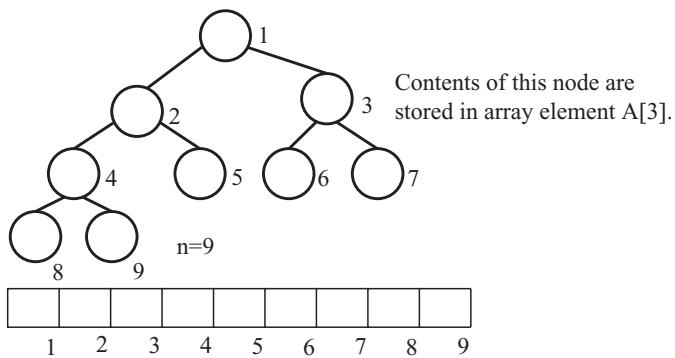


Fig. 13.4 The mapping between the nodes in a balanced binary tree and the elements of an array.

- The root is stored in $A[1]$.
- The parent of $A[i]$ is $A[\lfloor \frac{i}{2} \rfloor]$.
- The left child of $A[i]$ is $A[2 \cdot i]$.
- The right child of $A[i]$ is $A[2 \cdot i + 1]$.
- The node in the far right of the bottom level is stored in $A[n]$.
- If $2i + 1 > n$, then the node does not have a right child.

Definition of a Heap: A heap imposes a partial order (see Section 20.6) on the set of values, requiring that the value of each node be greater than or equal to that of each of the node's children. There are no rules about whether the left or the right child is larger. See Figure 13.5.

Maximum at Root: An implication of the heap rules is that the root contains the maximum value. The maximum may appear repeatedly in other places as well.

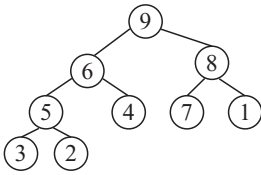


Fig. 13.5 An example of nodes ordered into a heap.

Exercise 13.4.1 gives you more practice understanding this definition.

The Heapify Problem

Specifications

Precondition: The input is a balanced binary tree such that its left and right subtrees are heaps. (That is, it is a heap except that its root might not be larger than that of its children.)

Postcondition: Its values are rearranged in place to make it complete heap.

Recursive Algorithm: The first task in making this tree into a heap is to put its maximum value at the root. See Figure 13.6. Because the left and right subtrees are heaps, the maxima of these trees are at their roots. Hence, the maximum of the entire tree is either at the root, at its left child node, or at its right child node. You find the maximum among these three. If the maximum is at the root, then you are finished. Otherwise, for the purpose of discussion, assume that the maximum is in the root's left child. Swap this maximum value with that of the root. The root and the right subtree now form a heap, but the left subtree might not. You will give the subtask of making your left subtree into a heap to a recursive friend. Before you can do this, you need to make sure that this subinstance meets the preconditions of the problem. By our precondition, our left subtree was a heap when we received it. We changed its root. After this change, it still has the property that its left and right subtrees are heaps. Hence, the preconditions of our problem are met and you can give your left subtree to your friend. By the postcondition, the friend makes this subtree into a heap. Your entire tree is now a heap.

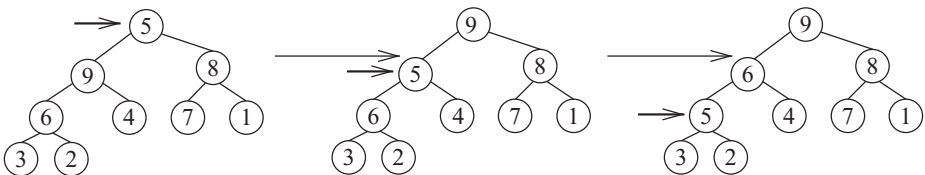


Fig. 13.6 An example computation of *Heapify*.

Code

algorithm *Heapify*(*r*)

<pre-cond>: The balanced binary tree rooted at $A[r]$ is such that its left and

right subtrees are heaps.

(post-cond): Its values are rearranged in place to make it complete heap.

```
begin
  if(A[rightchild(r)] is max of {A[r], A[rightchild(r)],
    A[leftchild(r)]}) then
    swap(A[r], A[rightchild(r)])
    Heapify(rightchild(r))
  elseif(A[leftchild(r)] is max of {A[r], A[rightchild(r)],
    A[leftchild(r)]}) then
    swap(A[r], A[leftchild(r)])
    Heapify(leftchild(r))
  else % A[r] is max of {A[r], A[rightchild(r)], A[leftchild(r)]}
    exit
  end if
end algorithm
```

Running Time: $T(n) = 1 \cdot T(n/2) + \Theta(1)$. From [Chapter 39](#) we know that $\frac{\log a}{\log b} = \frac{\log 1}{\log 2} = 0$ and $f(n) = \Theta(n^0)$, so $c = 0$. Because $\frac{\log a}{\log b} = c$, we conclude that time is dominated by all levels and $T(n) = \Theta(f(n) \log n) = \Theta(\log n)$.

Because this algorithm recurses only once per call, it can easily be made into an iterative algorithm.

Iterative Algorithm: A good loop invariant would be “The entire tree is a heap except that node i might not be greater or equal to both of its children. As well, the value of i ’s parent is at least the value of i and of i ’s children.” When i is the root, this is the precondition. The algorithm proceeds as in the recursive algorithm. Node i follows one path down the tree to a leaf. When i is a leaf, the whole tree is a heap.

Code

algorithm *Heapify(r)*

(pre-cond): The balanced binary tree rooted at $A[r]$ is such that its left and right subtrees are heaps.

(post-cond): Its values are rearranged in place to make it complete heap.

```
begin
   $i = r$ 
  loop
    (loop-invariant): The entire tree rooted at  $A[r]$  is a heap except that
    node  $i$  might not be greater or equal to both of its children. As well,
    the value of  $i$ ’s parent is at least the value of  $i$  and of  $i$ ’s children.

    exit when  $i$  is a leaf
    if(A[rightchild(i)] is max of {A[i], A[rightchild(i)],
      A[leftchild(i)]}) then
```

```

        swap(A[i], A[rightchild(i)])
        i = rightchild(i)
    elseif(A[leftchild(i)] is max of {A[i], A[rightchild(i)],
    A[leftchild(i)]}) then
        swap(A[i], A[leftchild(i)])
        i = leftchild(i)
    else % A[i] is max of {A[i], A[rightchild(i)], A[leftchild(i)]}
        exit
    end if
end loop
end algorithm
Running Time:  $T(n) = \Theta(\text{height of tree}) = \Theta(\log n)$ .

```

The MakeHeap Problem

Specifications

Precondition: The input is an array of numbers, which can be viewed as a balanced binary tree of numbers.

Postcondition: Its values are rearranged in place to make it heap.

Recursive Algorithm: The obvious recursive algorithm is to recursively make $\lceil \frac{n-1}{2} \rceil$ of the numbers into a heap, make another $\lceil \frac{n-1}{2} \rceil$ into a heap, and put the remaining number at the root of a tree with these two heaps as children. This now meets the precondition for *Heapify*, which turns the whole thing into a heap.

Running Time: $T(n) = 2T(\frac{n}{2}) + \Theta(\log n)$. Again from [Chapter 39](#), $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$ and $f(n) = \Theta(n^0 \log n)$, so $c = 0$. Because $\frac{\log a}{\log b} > c$, we conclude that time is dominated the base cases and $T(n) = \Theta(n^{\log a / \log b}) = \Theta(n)$.

The structure of the recursive tree for this algorithm is very predictable, so it can easily be made into an iterative algorithm, which calls *Heapify* on exactly the same nodes though in a slightly different order.

Iterative Algorithm: See [Figure 13.7](#). The loop invariant is that all subtrees of height i are heaps. Initially, the leaves of height $i = 1$ are already heaps. Suppose that all subtrees of height i are heaps. The subtrees of height $i + 1$ have the property that their left and right subtrees are heaps. Hence, we can use *Heapify* to make them into heaps. This maintains the loop invariant while increasing i by one. The postcondition clearly follows from the loop invariant and the exit condition that $i = \log n$.

Code

algorithm *MakeHeap*()

⟨pre-cond⟩: The input is an array of numbers, which can be viewed as a balanced binary tree of numbers.

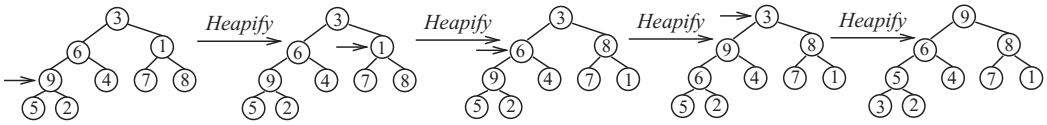


Fig. 13.7 An example of the iterative version of *MakeHeap*.

(post-cond): Its values are rearranged in place to make it a heap.

begin

 loop $k = \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor - 1, \lfloor \frac{n}{2} \rfloor - 2, \dots, 2, 1$

Heapify(k)

 end loop

end algorithm

Running Time: The number of subtrees of height i is $2^{(\log n) - i}$, because each such tree has its root at level $(\log n) - i$ in the tree. Each takes $\Theta(i)$ to heapify. This gives a total time of $T(n) = \sum_{i=1}^{\log n} (2^{(\log n) - i})i$. This sum is geometric. Hence, its total is Theta of its maximum term. The first term with $i = 1$ is $(2^{(\log n) - i})i = \Theta(2^{\log n}) = \Theta(n)$. The last term with $i = \log n$ is $(2^{(\log n) - i})i = (2^0) \log n = \log n$. The first term is the biggest, giving a total time of $\Theta(n)$. See [Chapter 38](#) for more on approximation summations.

The HeapSort Problem

Specifications

Precondition: The input is an array of numbers.

Postcondition: Its values are rearranged in place to be in sorted order.

Algorithm: The loop invariant is that for some $i \in [0, n]$, the $n - i$ largest elements have been removed and are sorted on the side, and the remaining i elements form a heap. See [Figures 13.8](#) and [13.9](#). The loop invariant is established for $i = n$ by forming a heap from the numbers using the *MakeHeap* algorithm. When $i = 0$, the values are sorted.

Suppose that the loop invariant is true for i . The maximum of the remaining values is at the root of the heap. Remove it and put it in its sorted place on the left end of the sorted list. Take the bottom right-hand element of the heap, and fill the newly created hole at the root. This maintains the correct shape of the tree. The tree now has the property that its left and right subtrees are heaps. Hence, you can use *Heapify* to make it into a heap. This maintains the loop invariant while decreasing i by one.

Array Implementation: The heap sort can occur in place within the array. As the heap gets smaller, the array entries on the right become empty. These can be used to store the sorted list that is on the side. Putting the root element where it belongs, putting the bottom left element at the root, and decreasing the size of the heap can be accomplished by swapping the elements at $A[1]$ and at $A[i]$ and decrementing i .

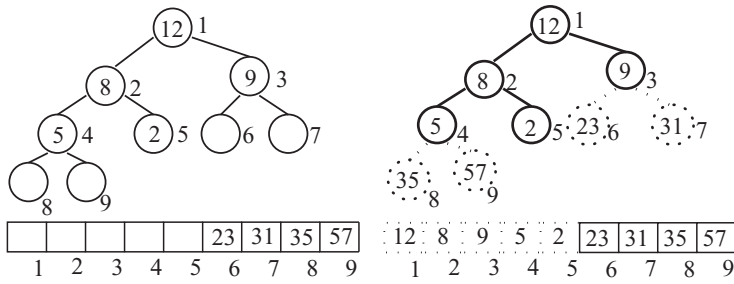


Fig. 13.8 The left diagram shows the loop invariant with $n - i = 9 - 5 = 4$ of the largest elements in the array and the remaining $i = 5$ elements forming a heap. The right diagram emphasizes the fact that though a heap is viewed as being stored in a tree, it is actually implemented in an array. When some of the elements are still in the tree and some are in the array, these views overlap.

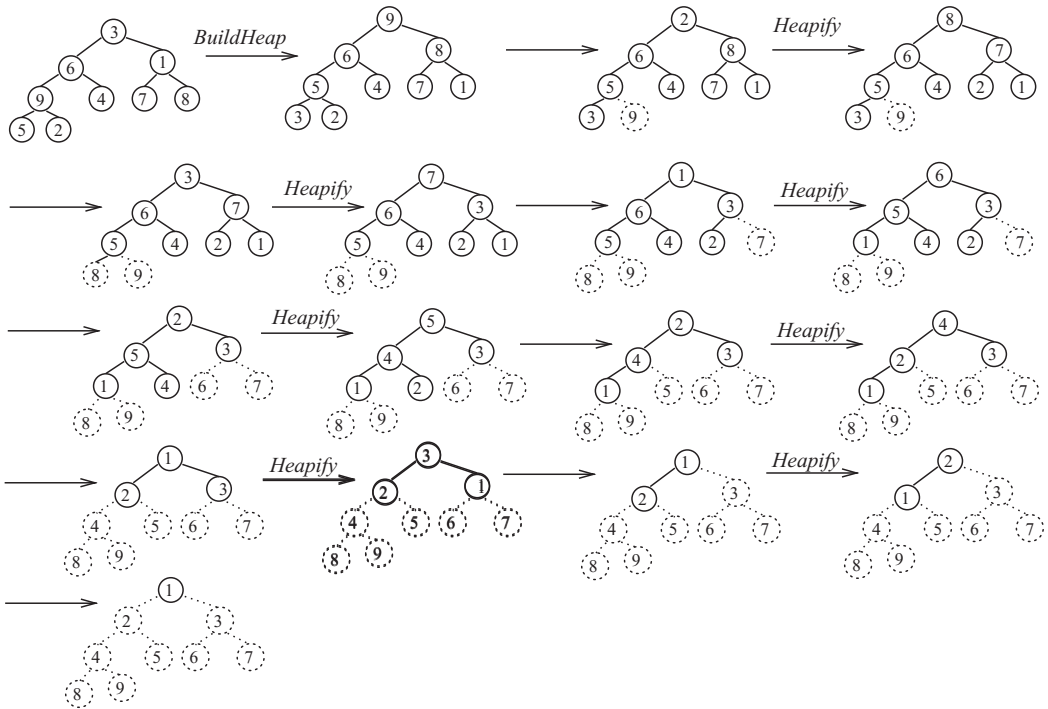


Fig. 13.9 An example computation of *HeapSort*.

Code

algorithm *HeapSort*()

$\langle$ *pre-cond* $\rangle$: The input is an array of numbers.

$\langle$ *post-cond* $\rangle$: Its values are rearranged in place to be in sorted order.


```

begin
    MakeHeap()
     $i = n$ 
    loop
        ⟨loop-invariant⟩: The  $n - i$  largest elements have been
        removed and are sorted in  $A[i + 1, n]$ , and the remaining  $i$  elements
        form a heap in  $A[1, i]$ .
        exit when  $i = 1$ 
        swap( $A[root]$ ,  $A[i]$ )
         $i = i - 1$ 
        Heapify( $root$ )      % On a heap of size  $i$ .
    end loop
end algorithm

```

Running Time: *MakeHeap* takes $\Theta(n)$ time, heapifying a tree of size i takes time $\log(i)$, for a total of $T(n) = \Theta(n) + \sum_{i=n}^1 \log i$. This sum behaves like an arithmetic sum. Hence, its total is n times its maximum value, i.e., $\Theta(n \log n)$.

Common Mistakes When Describing These Algorithms: Statements that are *always* true, such as “The root is the max of any heap,” give no information about the state of the program within the loop. For *Heapify*, “The left subtree and the right subtree of the current node are heaps” is useful. However, in the end the subtree becomes a leaf, at which point this loop invariant does not tell you that the whole tree is a heap. For *HeapSort*, “The tree is a heap” is good, but how do you get a sorted list from this in the end? Do not run routines without making sure that their preconditions are met, such as having *HeapSort* call *Heapify* without being sure that the left and right subtrees of the given node are heaps.

Priority Queues: Like stacks and queues, priority queues are an important ADT.

Definition: A *priority queue* consists of:

Data: A set of elements, each of which is associated with an integer that is referred to as the *priority* of the element.

Operations

Insert an Element: An element, along with its priority, is added to the queue. Coding this is left for [Exercise 13.3.2](#).

Change Priority: The priority of an element already in the queue is changed. The routine is passed a pointer to the element within the priority queue and its new priority. Coding this is left for [Exercise 13.3.3](#).

Remove an Element: Removes and returns an element of the highest priority from the queue.

Implementations

Implementation	Insert Time	Change Time	Remove Time
Sorted in an array or linked list by priority	$O(n)$	$O(n)$	$O(1)$
Unsorted in an array or linked list separate queue for each priority level	$O(1)$	$O(1)$	$O(n)$
(To add, go to correct queue; to delete, find first nonempty queue)	$O(1)$	$O(1)$	$O(\text{No. of priorities})$
Heaps	$O(\log n)$	$O(\log n)$	$O(\log n)$

Heap Implementation: The elements of a priority queue are stored in a heap ordered according to the priority of the elements.

Operations

Remove an Element: The element of the highest priority is at the top of the heap. It can be removed, and the heap then should be reheapified as done in *HeapSort*.

Insert an Element: Place the new element in the lower right corner of the heap, and then bubble it up the heap until it finds the correct place according to its priority.

Change Priority: The routine is passed a pointer to the element whose priority is changing. After making the change, this element is bubbled either up or down the heap, depending on whether the priority has increased or decreased.

Exercise 13.3.1 (See solution in Exercise Solutions.) Consider a heap storing the values $1, 2, 3, \dots, 15$.

1. Where in the heap can the value 1 go?
2. Which values can be stored in entry $A[2]$?
3. Where in the heap can the value 15 go?
4. Where in the heap can the value 6 go?

Exercise 13.3.2 Design an algorithm to insert a new element into the heap implementation of the priority queue.

Exercise 13.3.3 Design an algorithm to change the priority of an element in the heap implementation of the priority queue.

13.4 Representing Expressions with Trees

We will now consider how to represent multivariate expressions using binary trees. We will develop the algorithms to evaluate, copy, differentiate, simplify, and print such an

expression. Though these are seemingly complex problems, they have simple recursive solutions.

Recursive Definition of an Expression:

- Single variables x, y , and z and single real values are themselves expressions.
- If f and g are expressions, then $f + g$, $f - g$, $f * g$, and f / g are also expressions.

Tree Data Structure: The recursive definition of an expression directly mirrors that of a binary tree. Because of this, a binary tree is a natural data structure for storing an expression. (Conversely, you can use an expression to represent a binary tree.)

Example 13.4.1 (Evaluate Expression)

This routine evaluates an expression that is represented by a tree (Figure 13.10). For example, it can evaluate $f = x * (y + 7)$, with $xvalue = 2$, $yvalue = 3$, and $zvalue = 5$, and return $2 * (3 + 7) = 20$.

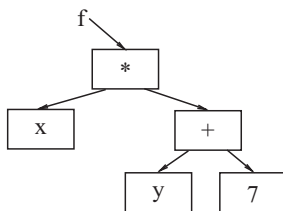


Fig. 13.10 Tree from Example 13.1 representing $f = x * (y + 7)$.

Code

algorithm *Eval*($f, xvalue, yvalue, zvalue$)

<pre-cond> f is an expression whose only variables are x, y , and z , and $xvalue$, $yvalue$, and $zvalue$ are the three real values to assign to these variables.

<post-cond> The returned value is the evaluation of the expression at these values for x, y , and z . The expression is unchanged.

begin

```

  if( $f$  = a real value) then
    result( $f$ )
  else if( $f$  = "x") then
    result( $xvalue$ )
  else if( $f$  = "y") then
    result( $yvalue$ )
  else if( $f$  = "z") then
    result( $zvalue$ )
  else if( $rootOp(f)$  = "+" ) then
    result( $Eval(leftSub(tree), xvalue, yvalue, zvalue)$ )

```

```

        + Eval(rightSub(tree), xvalue, yvalue, zvalue))
    else if(rootOp(f) = " - ") then
        result(Eval(leftSub(tree), xvalue, yvalue, zvalue)
            - Eval(rightSub(tree), xvalue, yvalue, zvalue))
    else if(rootOp(f) = " * ") then
        result(Eval(leftSub(tree), xvalue, yvalue, zvalue)
            × Eval(rightSub(tree), xvalue, yvalue, zvalue))
    else if(rootOp(f) = " / ") then
        result(Eval(leftSub(tree), xvalue, yvalue, zvalue)
            / Eval(rightSub(tree), xvalue, yvalue, zvalue))
    end if
end algorithm

```

Example 13.4.2 (Differentiate Expression)

This routine computes the derivative of a given expression with respect to an indicated variable.

Specification:

Preconditions: The input consists of $\langle f, x \rangle$, where f is an expression represented by a tree and x is a string giving the name of a variable.

Postconditions: The output is the derivative $f' = df/dx$. This derivative should be an expression represented by a tree whose nodes are separate from those of f . The data structure f should remain unchanged. See [Figure 13.11](#).

Coding this is left for [Exercise 13.4.1](#), and tracing it for [Exercise 13.4.2](#).

Example 13.4.3 (Simplify Expression)

This routine simplifies a given expression. For example, the derivative of $x * y$ with respect to x will be computed to be $1 * y + x * 0$. This should be simplified to y .

Specification

Preconditions: The input consists of an expression f represented by a tree.

Postconditions: The output is another expression that is a simplification of f . Its nodes should be separate from those of f , and f should remain unchanged.

Code

algorithm *Simplify*(f)

<pre-cond> f is an expression.

<post-cond> The output is a simplification of this expression.

begin

```

    if( $f$  = a real value or a single variable) then
        result( Copy( $f$ ) )
    else %  $f$  is of the form ( $g'$  op  $h'$ )

```

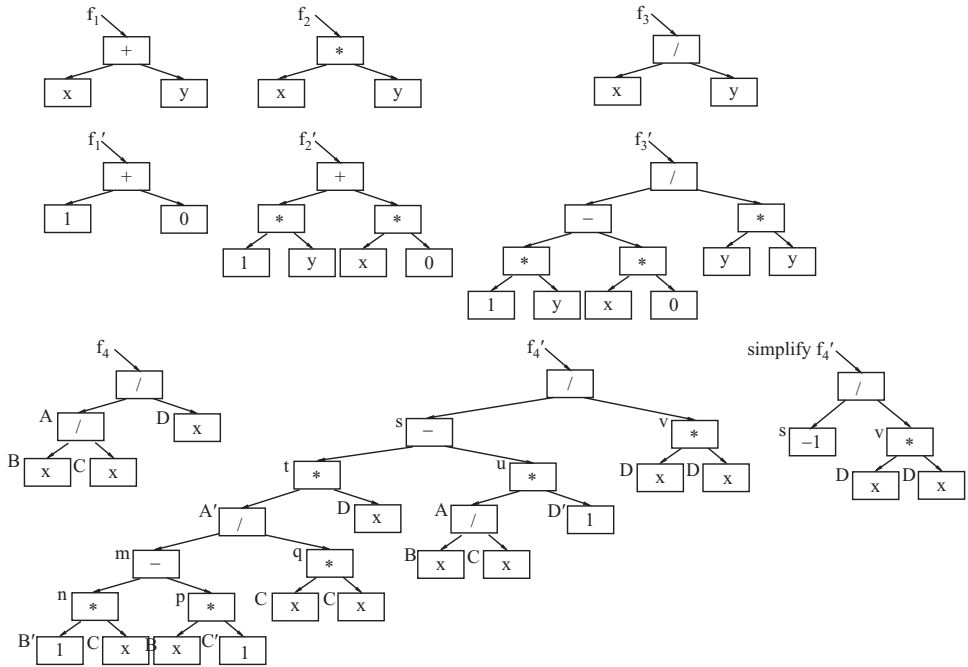


Fig. 13.11 Four functions and their derivatives. The fourth derivative has been simplified.

```

g = Simplify(leftSub(f))
h = Simplify(rightSub(f))
if(one of the following forms applies:
    1 * h = h    g * 1 = g    0 * h = 0    g * 0 = 0
    0 + h = h    g + 0 = g    g - 0 = g    x - x = 0
    0/h = 0      g/1 = g      g/0 = ∞      x/x = 1
    6 * 2 = 12   6/2 = 3     6 + 2 = 8     6 - 2 = 4)) then
    result(the simplified form)
else
    result( g op h)
end if
end if
end algorithm

```

This is traced out in [Exercise 13.4.3](#).

Exercise 13.4.1 (See solution in Exercise Solutions.) Describe the algorithm for the derivative. Do not give the complete code. Only give the key ideas.

Exercise 13.4.2 Trace out the execution of the derivative algorithm on the instance $f = (x/x)/x$ given above. In other words, draw a tree with a box for each time a routine is called. For each box, include only the function f passed and derivative returned.

Exercise 13.4.3 (See solution in Exercise Solutions.) Trace out the execution of `Simplify` on the derivative f' obtained in [Exercise 13.4.1](#), where $f = (x/x)/x$. In other words, draw a tree with a box for each time a routine is called. For each box, include only the function f passed and the simplified expression returned.

14 Recursive Images

Recursion can be used to construct very complex and beautiful pictures. We begin by combining the same two fixed images recursively over and over again. This produces fractal-like images whose substructures are identical to the whole. Next we will generate random mazes by using randomness to slightly modify these two images so that the substructures are not identical.

14.1 Drawing a Recursive Image from a Fixed Recursive and a Base Case Image

Drawing an Image: An image is specified by a set of lines, circles, and arcs and by two points A and B that are referred to as the *handles*. Before such an image can be drawn on the screen, its location, size, and orientation on the screen need to be specified. We will do this by specifying two points A and B on the screen. Then a simple program can translate, rotate, scale, and draw the image on the screen in such a way that the two handle points of the image land on these two specified points on the screen.

Specifying a Recursive Image: A recursive image is specified by the following:

1. a *base case* image,
2. a *recurse* image,
3. a set of places within the recurse image to recurse,
4. the two points A and B on the screen at which the recursive image should be drawn, and
5. an integer n .

The Base Case: If $n = 1$, then the base case image is drawn.

Recurring: If $n > 1$, then the recursive image is drawn on the screen at the location specified. Included in the recursive image are a number of *places to recurse*. These are each depicted by an arrow, $\longrightarrow$. When the recursive image is translated, rotated, scaled, and drawn on the screen, these arrows are located somewhere on the screen. The arrows themselves are not drawn. Instead, the same picture is drawn recursively at these locations, but with the value $n - 1$.

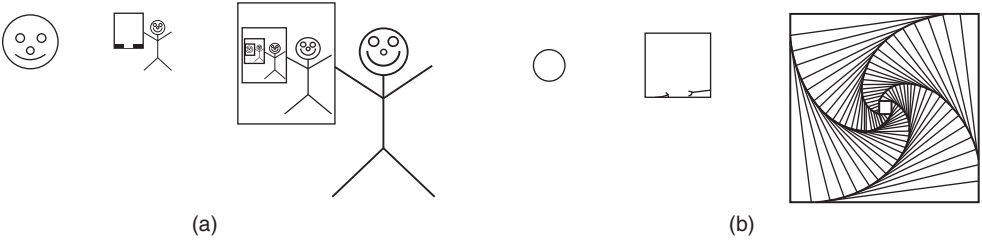


Fig. 14.1 (a) Man recursively framed; (b) rotating square.

Examples

Man Recursively Framed: See Figure 14.1.a. The base case for this construction consists of a happy face. When $n = 1$, this face is drawn. The recursive image consists of a man holding a frame. There is one place to recurse within the frame. Hence, when $n = 2$, this man is drawn with the $n = 1$ happy face inside it. For $n = 3$, the man is holding a frame containing the $n = 2$ image of a man holding a framed $n = 1$ happy face. The recursive image provided is with $n = 5$. It consists of a man holding a picture of a man holding a picture of a man holding a picture of $\dots$ a face. In general, the recursive image for n contains $R(n) = R(n - 1) + 1 = n - 1$ men and $B(n) = B(n - 1) = 1$ happy faces.

Rotating Square: See Figure 14.1.b. This image is constructed similarly to the previous one. Here, however, the $n = 1$ base case consists of a circle. The recursive image consists of a single square with the $n - 1$ image shrunk and rotated within it. The squares continue to spiral inward until the base case is reached.

Birthday Cake: See Figure 14.2. The birthday cake recursive image is different in that it recurses in two places. The $n = 1$ base case consists of a single circle. The recursive image consists of a single line with two smaller copies of the $n - 1$ image drawn above it. In general, the recursive image for n contains $R(n) = 2R(n - 1) + 1 = 2^{n-1} - 1$ lines from the recursive image and $B(n) = 2B(n - 1) = 2^{n-1}$ circles from the base case image.

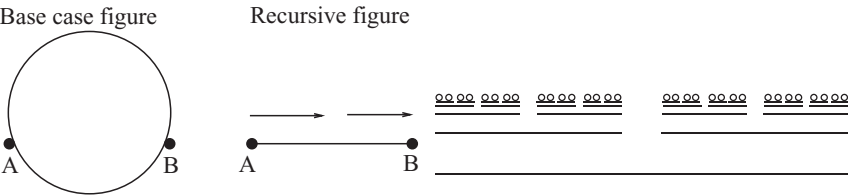


Fig. 14.2 Birthday cake.

Leaf: See Figure 14.3. A leaf consists of a single stem plus eight subleaves along it. Each subleaf is an $n - 1$ leaf. The base case image is empty, and the recursive

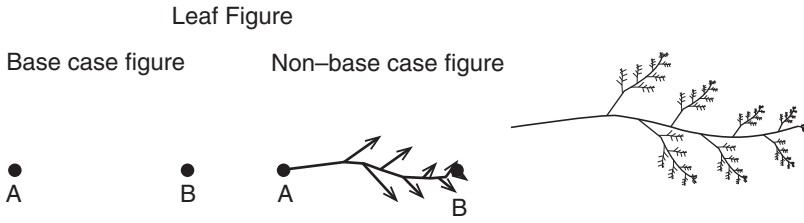


Fig. 14.3 Leaf.

image consists of the stem plus the eight places to recurse. Hence, the $n = 1$ image is blank. The $n = 2$ image consists of a lone stem. The $n = 3$ image is a stem with eight stems for leaves, and so on. In general, the recursive image for n contains $R(n) = 8R(n - 1) + 1 = \frac{1}{7}(8^{n-1} - 1)$ stems from the recursive image.

Fractal: See Figure 14.4. This recursive image is a classic. The base case is a single line. The recursive image is empty except for four places to recurse. Hence, $n = 1$ consists of the line. $n = 2$ consists of four lines, forming a line with an equilateral triangle jutting out of it. As n becomes large, the image becomes a snowflake. It is a fractal in that every piece of it looks like a copy of the whole.

The classic way to construct it is slightly different from what is done here. In the classical method, we are allowed the following operation. Given a line, divide it into three equal parts. Replace the middle part with the two equal-length lines forming an equilateral triangle. Starting with a single line, construct the fractal by repeatedly applying this operation to all the lines that appear.

In general, the recursive image for n contains $B(n) = 4B(n - 1) = 4^{n-1}$ base case lines. The length of each of these lines is $L(n) = \frac{1}{3}L(n - 1) = (\frac{1}{3})^{n-1}$. The total length of all these lines is $B(n) \cdot L(n) = (\frac{4}{3})^{n-1}$. As n approaches infinity, the fractal becomes a curve of infinite length.

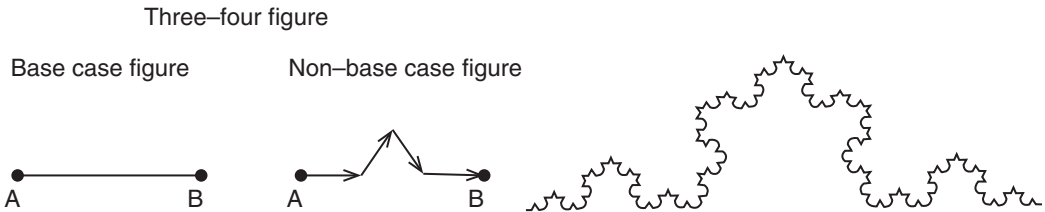


Fig. 14.4 Fractal.

Exercise 14.1.1 (See solution in Exercise Solutions.) See Figure 14.5.a. Construct the recursive image that arises from the base case and recursive image for some large n . Describe what is happening.

Exercise 14.1.2 (See solution in Exercise Solutions.) See Figure 14.5.b. Construct the recursive image that arises from the base case and recursive image for some large n .

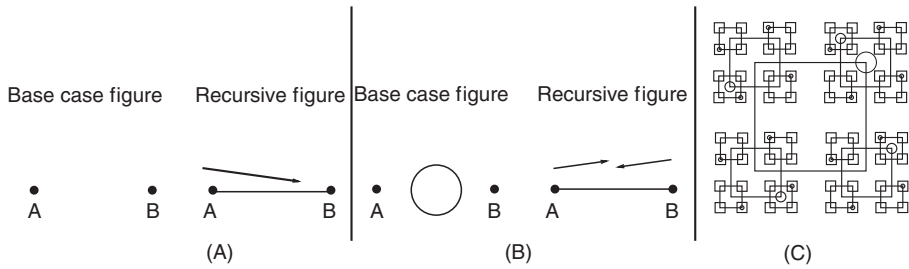


Fig. 14.5 Three more examples.

Note that one of the places to recurse is pointing opposite the other. To line the image up with these arrows, the image must be rotated 180° . The image cannot be flipped.

Exercise 14.1.3 See Figure 14.5.c. This construction looks simple enough. The difficulty is keeping track of at which corners the circle is. Construct the base case and the recursive image from which the given recursive image arises. Describe what is happening.

14.2 Randomly Generating a Maze

We will use similar methods to generate a random maze. The maze M will be represented by an $n \times m$ two-dimensional array with entries from $\{\text{brick}, \text{floor}, \text{cheese}\}$. Walls consist of lines of bricks. A mouse will be able to move along floor squares in any of the eight directions. The maze generated will not contain corridors as such, but only many small rectangular rooms. Each room will have either one door in one corner of the room or two doors in opposite corners. The *cheese* will be placed in a room that is chosen randomly from among the rooms that are far enough from the start location.

Precondition: The routine *AddWalls* is passed a matrix representing the maze as constructed so far and the coordinates of a room within it. The room will have a surrounding wall except for one door in one of its corners. The room will be empty of walls. The routine is also passed a flag indicating whether or not cheese should be added somewhere in the room.

Postcondition: The output is the same maze with a randomly chosen submaze added within the indicated room and cheese added as appropriate.

Initial Conditions: To meet the preconditions of *AddWalls*, the main routine first constructs the four outer walls with the top right corner square left as a floor tile to act as a door into the maze and as the start square for the mouse. Calling *AddWalls* on this single room completes the maze.

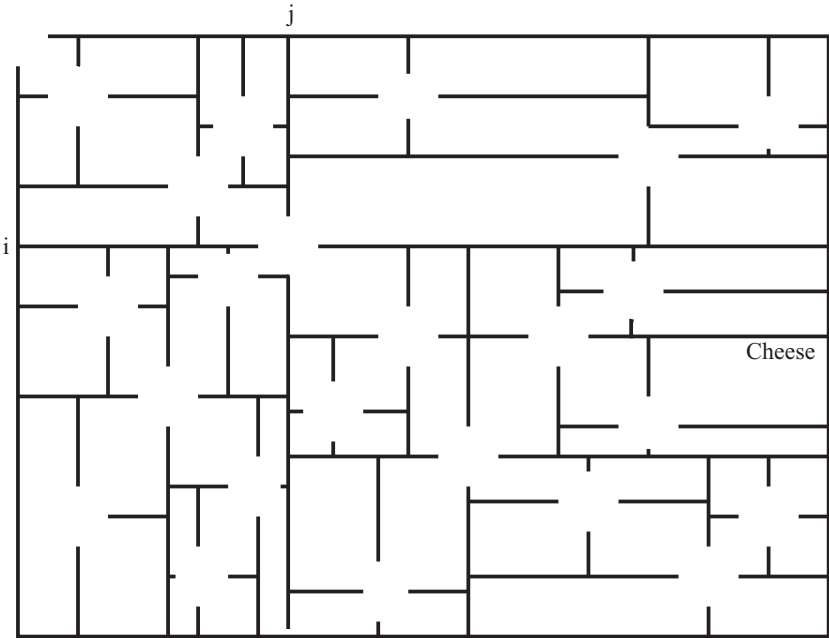


Fig. 14.6 A maze containing cheese.

Subinstances: If the indicated room has height and width of at least 3, then the routine *AddWalls* will choose a single location (i,j) uniformly at random from all those in the room that are not right next to one of its outer walls. (The (i,j) chosen by the top stack frame in Figure 14.6 is indicated.) A wall is added within the room all the way across row i and all the way down column j , subdividing the room into four smaller rooms. To act as a door connecting these four rooms, the square at location (i,j) remains a floor tile. See Figure 14.7. Then four friends are asked to fill a maze into each of these four smaller rooms. If our room is to have cheese, then one of the three rooms not containing the door to our room is selected to contain the cheese.

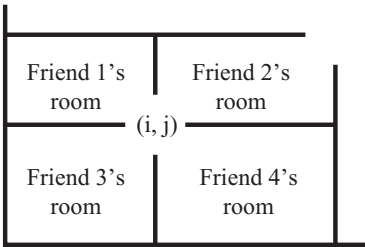


Fig. 14.7 Partitioning a room of the maze.

Running Time: The time required to construct an $n \times n$ maze is $\Theta(n^2)$. This can be seen two ways. For the easy way, note that a brick is added at most once to any entry of

the matrix and that there are $\Theta(n^2)$ entries. The hard way solves the recurrence relation $T(n) = 4T(n/2) + \Theta(n) = \Theta(n^2)$.

Searching the Maze: One way of representing a maze is by a graph. [Chapter 20](#) presents a number of iterative algorithms for searching a graph. [Section 20.5](#) presents the recursive version of the depth-first search algorithm. All of these could be used by a mouse to find the cheese.

Exercise 14.2.1 Write the code for generating a maze of rooms.

Exercise 14.2.2 Tiling: The precondition to the problem is that you are given three integers $\langle n, i, j \rangle$, where i and j are in the range 1 to 2^n . You have a 2^n by 2^n square board of squares. You have a sufficient number of tiles each with the shape . Your goal is to place nonoverlapping tiles on the board to cover each of the $2^n \times 2^n$ tiles except for the single square at location $\langle i, j \rangle$. Give a recursive algorithm for this problem in which you place one tile yourself then have four friends help you. What is your base case?

15 Parsing with Context-Free Grammars

An important computer science problem is parsing a string according to a given context-free grammar. A *context-free grammar* is a means of describing which strings of characters are contained within a particular language. It consists of a set of rules and a start *nonterminal* symbol. Each rule specifies one way of replacing a nonterminal symbol in the current string with a string of terminal and nonterminal symbols. When the resulting string consists only of terminal symbols, we stop. We say that any such resulting string has been *generated* by the grammar.

Context-free grammars are used to understand both the syntax and the semantics of many very useful languages, such as mathematical expressions, Java, and English. The *syntax* of a language indicates which strings of tokens are valid sentences in that language. The *semantics* of a language involves the meaning associated with strings. In order for a compiler or natural-language recognizers to determine what a string means, it must *parse* the string. This involves deriving the string from the grammar and, in doing so, determining which parts of the string are noun phrases, verb phrases, expressions, and terms.

Some context-free grammars have a property called *look ahead one*. Strings from such grammars can be parsed in linear time by what I consider to be one of the most amazing and magical recursive algorithms. This algorithm is presented in this chapter. It demonstrates very clearly the importance of working within the friends level of abstraction instead of tracing out the stack frames: Carefully write the specifications for each program, believe by magic that the programs work, write the programs calling themselves as if they already work, and make sure that as you recurse, the instance being input gets smaller.

In [Chapter 25](#) we will analyze an elegant dynamic programming algorithm that parses a string from any context-free grammar, not just look ahead one, in $\Theta(n^3)$ time.

The Grammar: We will look at a very simple grammar that considers expressions over $\times$ and $+$. In this grammar, a *factor* is either a simple integer or a more complex expression within brackets; a *term* is one or more factors multiplied together; and an *expression* is one or more terms added together. More precisely:

$$\text{exp} \Rightarrow \text{term}$$
$$\Rightarrow \text{term} + \text{term} + \cdots + \text{term}$$
$$\text{term} \Rightarrow \text{fact}$$
$$\Rightarrow \text{fact} * \text{fact} * \cdots * \text{fact}$$

$$\begin{aligned} \text{fact} &\Rightarrow \text{int} \\ &\Rightarrow (\text{exp}) \end{aligned}$$

Nonterminals, Terminals, and Rules: More generally, a grammar is defined by a set of *nonterminals*, a set of *terminals*, a *start nonterminal*, and a set of rules. Here the nonterminals are ‘exp,’ ‘term,’ and ‘fact.’ The terminals are integers, the character ‘+’, and the character ‘*’. The start nonterminal is ‘exp.’ The preceding display gives the list of rules for this grammar.

A Derivation of a String: A grammar defines a *language* of strings that can be derived in the following way. A derivation of a string starts with the start symbol (a nonterminal). Then each rule, like those just given, says that you can replace the nonterminal on the left with the string of terminals and nonterminals on the right.

A Parsing of an Expression: Let s be a string consisting of terminals. A parsing of this string is a tree. Each internal node of the tree is labeled with a nonterminal symbol, and the root with the start nonterminal. Each internal node must correspond to a rule of the grammar. For example, for rule $A \Rightarrow BC$, the node is labeled A and its two children are labeled B and C . The leaves of the tree, read left to right, give the input string s of terminals. Figure 15.1 is an example.

The Parsing Abstract Data Type: The following is an example of where it is useful not to give the full implementation details of an abstract data type. In fact, we will even leave the specification of parsing structure open for the implementer to decide.

For our purposes, we will only say the following: When p is a variable of type parsing, we will use “ $p = 5$ ” to indicate that it is assigned a parsing of the expression ‘5’. We will go on to *overload* the operations $*$ and $+$ as operations that join two parsings into one. For example, if p_1 is a parsing of the expression ‘ $2 * 3$ ’ and p_2 of ‘ $5 * 7$ ’, then we will use $p = p_1 + p_2$ to denote a parsing of the expression ‘ $2 * 3 + 5 * 7$ ’.

The implementer defines the structure of a parsing by specifying in more detail what these operations do. For example, if the implementer wants a parsing to be a binary tree representing the expression, then $p_1 + p_2$ would be the operation of constructing a binary tree with the root being a new ‘+’ node, the left subtree being the binary tree p_1 , and the right subtree being the binary tree p_2 . On the other hand, if the implementer wants a parsing to be simply an integer evaluation of the expression, then $p_1 + p_2$ would be the integer sum of the integers p_1 and p_2 .

Specifications for the Parsing Algorithm

Precondition: The input consists of a string of tokens s . The possible tokens are the characters ‘*’ and ‘+’ and arbitrary integers. The tokens are indexed as $s[1], s[2], s[3], \dots, s[n]$.

Postcondition: If the input is a valid *expression* generated by the grammar, then the output is a *parsing* of the expression. Otherwise, an error message is output.

The algorithm consists of one routine for each nonterminal of the grammar: *GetExp*, *GetTerm*, and *GetFact*.

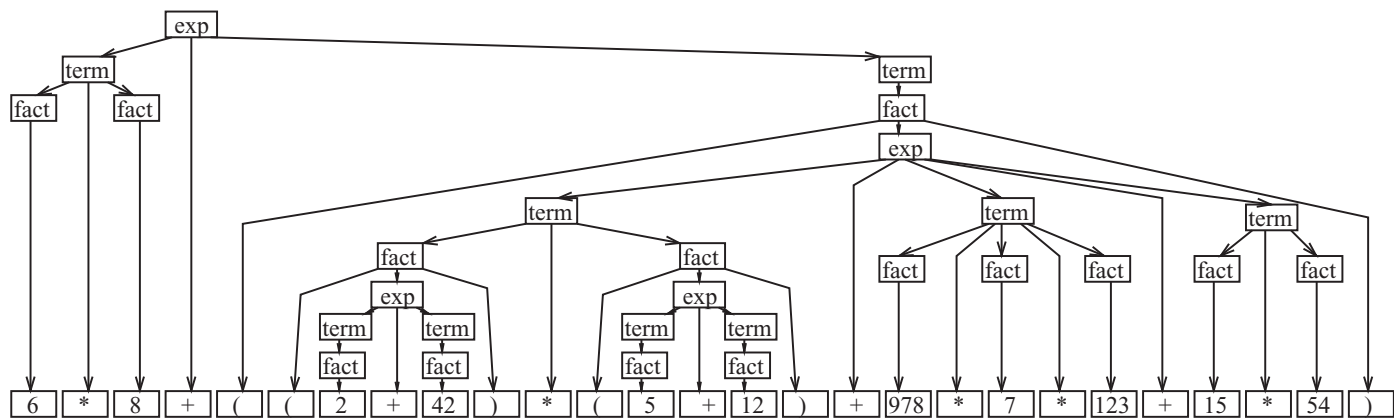


Fig. 15.1 A parse tree for the string $s = 6 * 8 + ((2 + 42) * (5 + 12) + 987 * 7 * 123 + 15 * 54)$.

Specifications for *GetExp*

Precondition: The input of *GetExp* consists of a string s of tokens and an index i that indicates a starting point within s .

Postcondition: The output consists of a parsing of the longest substring $s[i], s[i + 1], \dots, s[j - 1]$ of s that starts at index i and is a valid expression. The output also includes the index j of the token that comes immediately after the parsed expression.

If there is no valid expression starting at $s[i]$, then an error message is output.

The specifications for *GetTerm* and *GetFact* are the same as for *GetExp*, except that they return the parsing of the longest term or factor starting at $s[i]$ and ending at $s[j - 1]$.

Examples of *GetExp*, *GetTerm*, and *GetFact*: See [Figure 15.2](#).

Reasoning for *GetExp*: Consider some input string s and some index i . The longest substring $s[i], \dots, s[j - 1]$ that is a valid expression consists of some number of terms added together. In all of these cases, it begins with a term. By magic, assume that the *GetTerm* routine already works. Calling *GetTerm*(s, i) will return p_{term} and j_{term} , where p_{term} is the parsing of this first term and j_{term} indexes the token immediately after this term. Specifically, if the expression has another term then j_{term} indexes the ‘+’ that is between these terms. Hence, we can determine whether there is another term by checking $s[j_{term}]$. If $s[j_{term}] = ‘+’$, then *GetExp* will call *GetTerm* again to get the next term. If $s[j_{term}]$ is not a ‘+’ but some other character, then *GetExp* is finished reading in all the terms. *GetExp* then constructs the parsing consisting of all of these terms added together.

The reasoning for *GetTerm* is the same.

GetExp Code

algorithm *GetExp*(s, i)

⟨pre-cond⟩: s is a string of tokens, and i is an index that indicates a starting point within s .

⟨post-cond⟩: The output consists of a parsing p of the longest substring $s[i], s[i + 1], \dots, s[j - 1]$ of s that starts at index i and is a valid expression. The output also includes the index j of the token that comes immediately after the parsed expression.

begin

 if ($i > |s|$) return “Error: Expected characters past end of string.”

 end if

$\langle p_{\langle term, 1 \rangle}, j_{\langle term, 1 \rangle} \rangle = \text{GetTerm}(s, i)$

$k = 1$

 loop

⟨loop-invariant⟩: The first k terms of the expression have been read.

GetExp:

$$\begin{aligned}
 s &= \boxed{(2 * 8 + 42 * 7) * 5 + 8} \\
 s &= (\boxed{2 * 8 + 42 * 7}) * 5 + 8 \\
 s &= (2 * 8 + \boxed{42 * 7}) * 5 + 8 \\
 s &= (2 * 8 + 42 * 7) * \boxed{5 + 8} \\
 s &= (2 * 8 + 42 * 7) * 5 + \boxed{8}
 \end{aligned}$$

GetTerm:

$$\begin{aligned}
 s &= \boxed{(2 * 8 + 42 * 7) * 5} + 8 \\
 s &= (\boxed{2 * 8} + 42 * 7) * 5 + 8 \\
 s &= (2 * 8 + \boxed{42 * 7}) * 5 + 8 \\
 s &= (2 * 8 + 42 * 7) * \boxed{5} + 8 \\
 s &= (2 * 8 + 42 * 7) * 5 + \boxed{8}
 \end{aligned}$$

GetFact:

$$\begin{aligned}
 s &= \boxed{(2 * 8 + 42 * 7)} * 5 + 8 \\
 s &= (\boxed{2} * 8 + 42 * 7) * 5 + 8 \\
 s &= (2 * \boxed{8} + 42 * 7) * 5 + 8 \\
 s &= (2 * 8 + \boxed{42} * 7) * 5 + 8 \\
 s &= (2 * 8 + 42 * \boxed{7}) * 5 + 8 \\
 s &= (2 * 8 + 42 * 7) * \boxed{5} + 8 \\
 s &= (2 * 8 + 42 * 7) * 5 + \boxed{8}
 \end{aligned}$$

Fig. 15.2 Example input instances are given for *GetExp*, *GetTerm*, and *GetFact*. The string s is the same for all examples. The beginning of the rectangle indicates the input index i at which the parsing should begin. The contents of the rectangle indicates resulting parsing. The end of the rectangle indicates the output index j at which the parsing ends.

```

    exit when  $s[j_{\langle term, k \rangle}] \neq '+'$ 
     $\langle p_{\langle term, k+1 \rangle}, j_{\langle term, k+1 \rangle} \rangle = \text{GetTerm}(s, j_{\langle term, k \rangle} + 1)$ 
     $k = k + 1$ 
end loop
 $p_{exp} = p_{\langle term, 1 \rangle} + p_{\langle term, 2 \rangle} + \dots + p_{\langle term, k \rangle}$ 
 $j_{exp} = j_{\langle term, k \rangle}$ 
return  $\langle p_{exp}, j_{exp} \rangle$ 
end algorithm

```

GetTerm Code**algorithm** *GetTerm*(s, i)

⟨pre-cond⟩: s is a string of tokens, and i is an index that indicates a starting point within s .

⟨post-cond⟩: The output consists of a parsing p of the longest substring $s[i], s[i + 1], \dots, s[j - 1]$ of s that starts at index i and is a valid term. The output also includes the index j of the token that comes immediately after the parsed term.

begin

if $(i > |s|)$ return “Error: Expected characters past end of string.” end if

$\langle p_{\langle fact, 1 \rangle}, j_{\langle fact, 1 \rangle} \rangle = \text{GetFact}(s, i)$

$k = 1$

loop

⟨loop-invariant⟩: The first k facts of the term have been read.

exit when $s[j_{\langle fact, k \rangle}] \neq '*'$

$\langle p_{\langle fact, k+1 \rangle}, j_{\langle fact, k+1 \rangle} \rangle = \text{GetFact}(s, j_{\langle fact, k \rangle} + 1)$

$k = k + 1$

end loop

$p_{term} = p_{\langle fact, 1 \rangle} * p_{\langle fact, 2 \rangle} * \dots * p_{\langle fact, k \rangle}$

$j_{term} = j_{\langle fact, k \rangle}$

return $\langle p_{term}, j_{term} \rangle$

end algorithm

Reasoning for GetFact: The longest substring $s[i], \dots, s[j - 1]$ that is a valid factor has one of the following two forms:

$$\text{fact} \Rightarrow \text{int}$$

$$\text{fact} \Rightarrow (\text{exp})$$

Hence, we can determine which form the factor has by testing $s[i]$.

If $s[i]$ is an integer, then we are finished. p_{fact} is a parsing of this single integer $s[i]$, and $j_{fact} = i + 1$. The $+1$ moves the index past the integer.

If $s[i] = '('$, then for s to be a valid factor there must be a valid expression starting at $j_{term} + 1$, followed by a closing bracket $'$ '. We can parse this expression with $\text{GetExp}(s, j_{term} + 1)$, which returns p_{exp} and j_{exp} . The closing bracket after the expression must be in $s[j_{exp}]$. Our parsed factor will be $p_{fact} = (p_{exp})$ and $j_{fact} = j_{exp} + 1$. The $+1$ moves the index past the $'$ '.

If $s[i]$ is neither an integer nor a $'($ ', then it cannot be a valid factor. Give a meaningful error message.

GetFact Code**algorithm** *GetFact*(s, i)

⟨pre-cond⟩: s is a string of tokens and i is an index that indicates a

starting point within s .

$\langle \text{post-cond} \rangle$: The output consists of a parsing p of the longest substring $s[i], s[i+1], \dots, s[j-1]$ of s that starts at index i and is a valid factor. The output also includes the index j of the token that comes immediately after the parsed factor.

begin

 if($i > |s|$) return “Error: Expected characters past end of string.”

 end if

 if($s[i]$ is an int)

$p_{fact} = s[j]$

$j_{fact} = i + 1$

 return $\langle p_{fact}, j_{fact} \rangle$

 else if ($s[i] = '('$)

$\langle p_{exp}, j_{exp} \rangle = \text{GetExp}(s, i + 1)$

 if ($s[j_{exp}] = ')'$)

$p_{fact} = (p_{exp})$

$j_{fact} = j_{exp} + 1$

 return $\langle p_{fact}, j_{fact} \rangle$

 else

 Output “Error: Expected ‘)’ at index j_{exp} ”

 end if

 else

 Output “Error: Expected integer or ‘(’ at index i ”

 end if

end algorithm

Tree of Stack Frames: *GetExp* calls *GetTerm*, which calls *GetFact*, which may call *GetExp*, and so on. If one were to draw out the entire tree of stack frames showing who calls whom, this would exactly mirror the parse tree that it created. See [Exercise 15.0.1](#).

Running Time: We prove that the running time of this entire computation is linear in the size of the parse tree produced, which in turn is linear in the size $\Theta(n)$ of the input string.

To prove the first, it is sufficient to prove that the running time of each stack frame either is constant or is linear in the number of children of the node in the parse tree that this stack frame produces. For example, if the stack frame for *GetFact* finds an integer, then its node in the parse tree has no children, but *GetFact* uses only a constant amount of time. In contrast, if a stack frame for *GetExp* reads in t terms, then its running time will be some constant times t , and its node in the parse tree will have t children.

We now prove that the size of the parse tree produced is linear in the size $\Theta(n)$ of the input string. If the grammar is such that every nonterminal goes to at least one terminal or at least two nonterminals, then each node in the parse tree either is a leaf or has at

least two children. It follows that the number of nodes in the parse tree will be at most some constant times the number of leaves, which is the size of the input string. In our grammar, however, an expression might go to a single term, which can go to a single factor. This creates a little path of outdegree one. It cannot, however, be longer than this, because a factor either is a leaf or has three children: one is ‘(’, the second an expression, and the third ‘)’. Such little paths can only increase the size of the parse tree by a factor of 3.

In conclusion, the running time is $\Theta(n)$.

Proof of Correctness: To prove that a recursive program works, we must consider the *size* of an instance. The routine need only consider the postfix $s[i], s[i+1], \dots$, which contains $|s| - i + 1$ characters. Hence, we will define the size of the instance $\langle s, i \rangle$ to be $|\langle s, i \rangle| = |s| - i + 1$. Let $H(n)$ be the statement “Each of *GetFac*, *GetTerm*, and *GetExp* works on the instance $\langle s, i \rangle$ when $|\langle s, i \rangle| = |s| - i + 1 \leq n$.” We prove by induction that $\forall n \geq 0, H(n)$.

If $|\langle s, i \rangle| = 0$, then $i > |s|$: There is no valid expression, term, or factor starting at $s[i]$, and all three routines return an error message. It follows that $H(0)$ is true.

If $|\langle s, i \rangle| = 1$, then there is one remaining token: For this to be a factor, term, or expression, it must be a single integer. *GetFac* is written to give the correct answer in this situation. *GetTerm* gives the correct answer, because it calls *GetFac*. *GetExp* gives the correct answer, because it calls *GetTerm*, which in turn calls *GetFac*. It follows that $H(1)$ is true.

Assume $H(n-1)$ is true, that is, that each of *GetFac*, *GetTerm*, and *GetExp* works on instances of size at most $n-1$.

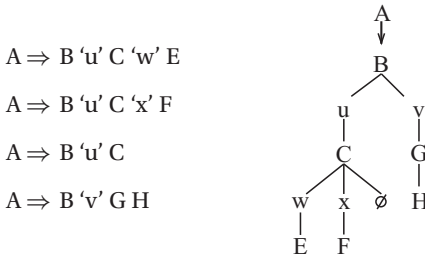
Consider *GetFac*(s, i) on an instance of size $|s| - i + 1 = n$. It makes at most one subroutine call, *GetExp*($s, i+1$). The size of this instance is $|s| - (i+1) + 1 = n-1$. Hence, by assumption, this subroutine call returns the correct answer. Because all of *GetFac*(s, i)’s subroutine calls return the correct answer, it follows that *GetFac*(s, i) works on all instances of size n .

Now consider *GetTerm*(s, i) on an instance of size $|s| - i + 1 = n$. It calls *GetFac* some number of times. The input instance for the first call *GetFac*(s, i) still has size n . Hence, the induction hypothesis $H(n-1)$ does *not* claim that it works. However, the previous paragraph proves that this routine does in fact work on instances of size n . The remaining calls are on smaller instances.

Finally, consider *GetExp*(s, i) on an instance $\langle s, i \rangle$ of size $|s| - i + 1 = n$. We use the previous paragraph to prove that this first subroutine call *GetTerm*(s, i) works.

In conclusion, all three work on all instances of size n and hence on $H(n)$. This completes the induction step.

Look Ahead One: A grammar is said to be *look ahead one* if, given any two rules for the same nonterminal, the first place that the rules differ is a difference in a terminal. (Equivalently, the rules can be viewed as paths down a tree.) This feature allows our parsing algorithm to look only at the next token in order to decide what to do next. Thus the algorithm runs in linear time. An example of a good set of rules would be



(Actually, even this grammar could also be problematic if when $s = 'bbbucccweee,'$ B could either be parsed as $'bbb'$ or as $'bbbu.'$ Having B eat the $'u'$ would be a problem.)

An example of a bad set of rules would be

$$A \Rightarrow BC$$

$$A \Rightarrow DE$$

With such a grammar, you would not know whether to start parsing the string as a B or a D . If you made the wrong choice, you would have to back up and repeat the process.

Exercise 15.0.1 (See solution in Exercise Solutions.) Consider $s = (((1) * 2 + 3) * 5 * 6 + 7)$.

1. Give a derivation of the expression s .
2. Draw the tree structure of the expression s .
3. Trace out the execution of your program on $GetExp(s, 1)$. In other words, draw a tree with a box for each time a routine is called. For each box, include only whether it is an expression, term, or factor and the string's $[i], \dots, s[j - 1]$ that is parsed.

Exercise 15.0.2 Consider a grammar G that includes the four look ahead rules for A . Give the code for $GetA(s, i)$ that is similar to that for $GetExp(s, i)$. We can assume that it can be parsed, so do not bother with error detection.

Exercise 15.0.3 If you are feeling bold, try to write a recursive program for a generic parsing algorithm. The input is $\langle G, T, s, i \rangle$, where G is a look ahead one grammar, T is a nonterminal of G , s is a string of terminals, and i is an index. The output consists of a parsing of the longest substring $s[i], s[i + 1], \dots, s[j - 1]$ of s that starts at index i and is a valid T according to the grammar G . In other words, the parsing starts with nonterminal T and ends with the string $s[i], s[i + 1], \dots, s[j - 1]$. The output also includes the index j of the token that comes immediately after the parsed expression. For example, $GetExp(s, i)$ is the same as calling this algorithm on $\langle G, exp, s, i \rangle$ where G is the grammar given above.

The loop invariant is that you have parsed a prefix $s[i], s[i + 1], \dots, s[j' - 1]$ of s , producing a partial parsing, p and the rest of the string, $s[j'], s[j' + 1], \dots, s[j - 1]$, will be parsed using one of the partial rules in the set R . For example, suppose the grammar G includes the four look ahead rules for A given above, we are starting with the nonterminal $T = A$, and we are parsing the string $s = 'bbbucccweee'$. Initially, we have parsed nothing, and R contains all of each of the four rules, namely, $R =$

$\{BuCwE, BuCxF, BuC, BvGH\}$. After two iterations, we have parsed ‘ $bbbu$ ’ using a parsing p_B for ‘ bbb ’ followed by the character ‘ u ’. We must parse the rest of the string $cccweee$ using one of the rules in $R = \{CwE, CxF, C\}$. Note that the used-up prefix Bu from the consistent rules and the inconsistent rules were deleted. Because the grammar is look ahead one, we know that either the first token in each rule of R is the same nonterminal B , or each rule of R begins with a terminal or is the empty rule. These are the two cases your iteration needs to deal with.

16 Key Concepts Summary: Recursion

People do have a hard time understanding recursion. But then a light switches on and they get it. I always feel that it is a matter of them trusting the friends (see [Section 11.1](#)).

The Steps: The following are the steps that need to be *considered* (but not necessarily written) when you are designing a recursive algorithm within the friends paradigm. (Be sure that the code that you produce does not indicate that you missed any of these steps.)

1. Carefully write the specifications for the problem.
 - (a) Precondition: Define the set of legal instances (inputs).
To be sure that we solve the problem for every legal instance.
So that we know what we can give to a friend.
 - (b) Postcondition: Required output.
So that we know what is expected of us.
So that we know what we can expect from our friend.
2. Focus on only one step. Do not trace out the entire computation.
3. Consider your input instance.
 - (a) Remember that you know nothing other than what is given to you by the preconditions.
 - (b) Be sure each path in your code returns what is required by the postcondition.
4. If your instance is sufficiently small, solve it yourself as a base case.
5. Construct one or more subinstances.
 - (a) It must be an instance to the same problem, i.e., meet the precondition.
 - (b) It must be smaller according to some definition of size.
6. Assume by magic (strong induction) your friends give you a correct solution for each of these.
7. Use their solutions for their subinstances to help you find a solution to your own instance.
8. Do not worry about who your boss is or how your friends solve their instance. No global variable or effects.
9. If you want more from your friends, change the pre- and/or postconditions, but
 - (a) Be sure to document it.
 - (b) Be sure you and all of your friends are using the same pre-/postconditions.
 - (c) Be sure to handle these changes yourself.

10. The code does not need to be much more complex than the following.

algorithm $Alg(a, b, c)$

<pre-cond>: Here a is a tuple, b an integer, and c a binary tree.

<post-cond>: Outputs x , y , and z which are useful objects.

begin

 if($\langle a, b, c \rangle$ is a sufficiently small instance)

 return($\langle [0], 0, emptytree \rangle$)

$\langle a_{sub1}, b_{sub1}, c_{sub1} \rangle$ = a part of $\langle a, b, c \rangle$

$\langle x_{sub1}, y_{sub1}, z_{sub1} \rangle = Alg(\langle a_{sub1}, b_{sub1}, c_{sub1} \rangle)$

$\langle a_{sub2}, b_{sub2}, c_{sub2} \rangle$ = a different part of $\langle a, b, c \rangle$

$\langle x_{sub2}, y_{sub2}, z_{sub2} \rangle = Alg(\langle a_{sub2}, b_{sub2}, c_{sub2} \rangle)$

$\langle x, y, z \rangle$ = combine $\langle x_{sub1}, y_{sub1}, z_{sub1} \rangle$ and $\langle x_{sub2}, y_{sub2}, z_{sub2} \rangle$

 return($\langle x, y, z \rangle$)

end algorithm

11. Give and solve the recurrence relation for the running time of this algorithm.

Formal Proof: The following are the formal steps similar to those for an iterative algorithm needed to prove a recursive program correct.

Precond $\Rightarrow$ Precond: Prove that if your instance meets the preconditions, then each of your subinstances also meets the preconditions.

Smaller or Basecase: Prove that if your instance is *sufficiently small* according to some measure of your choosing, then it is a *base case*, in which case you do not recurse. Otherwise, the subinstances that you give to your friends are (according to this same measure) *smaller* than your instance. Also your instance has a finite size.

Postcond $\Rightarrow$ Postcond: Prove that if your friend's solutions meet the postconditions for their subinstances, then your solution meets the postcondition for your instance.

Prove that your solution for the base case meets the postconditions.

Running Time Using Recurrence Relations: Let a denote the number of friends that you have. Let b be such that each friend is given an instance of size $\frac{n}{b}$, where n is the size of your instances. Let c be such that the time that you personally spend is $O(n^c)$. Solve the recurrence relation $T(n) = aT(\frac{n}{b}) + O(n^c)$. If $\frac{\log a}{\log b} > c$, then the running time is dominated by the bases cases and is $T(n) = O(n^{\frac{\log a}{\log b}})$. If $\frac{\log a}{\log b} < c$, then the time is dominated by the work in the top stack frame and is $T(n) = O(n^c)$. If $\frac{\log a}{\log b} = c$, then all the levels require about the same amount of time and the the total time is $O(n^c \log n)$.

Trees: A common example is recursion on trees.

Subinstances: If your input is a binary tree, then surely you will have two friends, one to whom you give your left subtree and one to whom you give your right subtree.

If each node may have many children, then given a tree, a stack frame simply loops over the children of the root.

Wall around Subtrees: To save yourself some time, you must trust your two friends. Imagine building a big wall around your two subtrees. The only information the friends have about the root or about the other subtree is what you pass them in their subinstances. The only information that you have is the root and what your two friends pass back to you in their solutions. From this information, and this alone, you have to be able to solve the given problem for the entire tree. In addition to your friends solving the given problem for the subtrees, what additional information might you want them to tell you?

Cases: With binary trees, the base case should be the empty tree. The general code needs to consider the case in which you have a big left and a big right subtree. Only if absolutely necessary should you be having an extra case for the tree consisting of a single node or when one of the subtrees is empty.

If each node may have many children, then no explicit base case is needed because if the root has no children, then its stack frame will not recurse.

In general, avoid cases with *if* statements. For example, use $\max(a, b, c)$ instead of separate cases for which it is bigger.

Running Time: Compute and add up the work done in each stack in the tree of stack frames. There is one recursive stack frame for each node in the tree (and one for each empty tree hanging off the leaves). Generally, each stack frame does a constant amount of work. Hence, the total time is linear in the number of nodes of the tree, i.e., $T(n) = O(n)$.

If each node may have many children, then each stack frame generally does a constant amount of work per subtree so the total time is linear in the number of edges of the tree which is one less than the number of nodes.

Proved another way, let c be the amount of work done in each stack frame. Then the recurrence relation is $T(n) = T(n_{\text{left}}) + T(n_{\text{right}}) + c$. Plugging in the guess $T(n) = c \cdot n$, gives $c \cdot n = c \cdot n_{\text{left}} + c \cdot n_{\text{right}} + c$. Dividing by c gives $n = n_{\text{left}} + n_{\text{right}} + 1$, which is correct because this is the count of the number of nodes.

17 Additional Exercises: Part II

- II.1. **Short Answers:** What is the four-sentence generic reason to explain why a recursive algorithm works?
- II.2. **Trace Eval:** Running *Eval* (Recursion page 365) on tree f_B and $x = 1$ has its left friend return $3 + 1 = 4$, its right $4 + 1 = 5$, and then itself would return $4 \times 5 = 20$. This algorithm still works when given a directed acyclic graph (DAG).

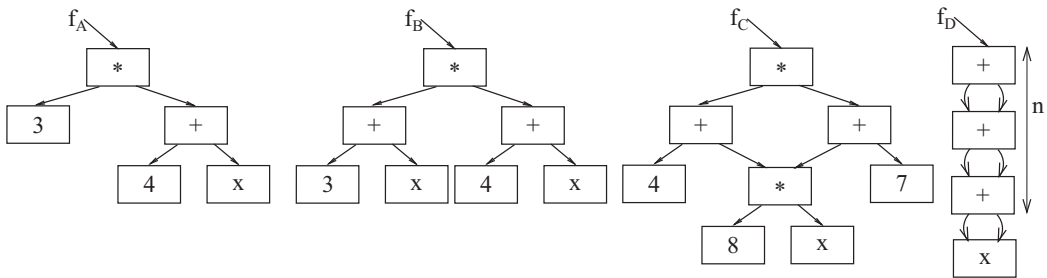


Fig. 17.1

- (a) What is the running time $T(n)$ of *Eval* on DAG f_D with a line of n additions? Give the recurrence relation and the bigOh of what it evaluates to, but don't show how you obtained this evaluation.
- (b) Give one sentence describing a change to the above algorithm to make the running time much faster.
What would the running time then be in terms of the number of nodes N and number of edges E in the DAG?

II.3. Planar Graphs

Prove the following claims about planar graphs:

- (a) In a tree $E = N - 1$. (Note $F = 1$ and $C = 1$.) A graph theory person would prove this using Structural Induction, which involves deleting nodes and/or edges to make the graph smaller. We will do exactly the same, except in computer science we will call it Recursion. Use ONE of the following two methods (the first is more familiar and the second is easier).
- i. Assume the tree G is rooted. Let r denote the number of children the root has. Write a recursive program that returns both N and E , and assures that $E = N - 1$.

- ii. Assume the tree G is not rooted. Write a recursive program that returns both N and E , and assures that $E = N - 1$. You will have only one friend. To give them something smaller, you must delete something from G .
- (b) For every connected planar graph, $E = N + F - 2$. Prove by induction on the number of faces.
- (c) For every planar graph, $E = N + F - C - 1$. Prove by induction on the number of components.
- (d) For every planar graph, $F \leq 2/3E$. Prove by building a bipartite graph between the E edges of G and the F faces with an “edge” between edge u, v and face f of G if they are “adjacent.”
- (e) The maximum degree of a node could be as big as $n-1$. However, for every planar graph, there exists a node with degree 5 or less.
- (f) The nodes of every planar graph G can be colored with six colors so that every edge is adjacent to nodes of two different colors.

Prove this by writing a recursive algorithm $SixColor(G)$ that assigns such a coloring to a given planar graph G .

Hint: If you were doing it iteratively, you would not know which node to color first. Hence, do not do it iteratively. Hence, do not say “the current node.”

Hint: Do not do a DFS or a BFS.

Hint: From our build up, you should know which node to color last.

Hint: Be sure to get and use help from at least one friend. Do not insult them by giving them a single node.

Note: Your algorithm must use that fact that G is planar, because it is NP-complete to color a graph with the minimum number of colors.

- (g) Provide a paragraph explaining why your previous algorithm works. Be sure to use the friend’s paradigm covered in [Section 11.1](#).
- (h) Change your previous algorithm so that instead of knowing the graph G is planar, the precondition is that it has at least one node with degree at most five. Does it work? If not, what goes wrong?

II.4. Abstract Recursion (Tower of Hanoi): Let’s see how we can use recursion not only by giving your friend a smaller instance but in a stranger abstract way. Review Towers of Hanoi from [Section 11.4](#). We will be generalizing it to having more than three pegs.

Any comparison based sorting code for sorting integers can be abstracted to handle real numbers. Any comparison between integers is replaced by the more complex comparisons between reals. Similarly, any code for moving normal disks can be abstracted to be able to move “super disks”. Any moving of a disk is replaced by the moving of a super disk.

For each choice of A' , k' , and n' , we will design a Towers of Hanoi algorithm $Alg_{\langle A', k', n' \rangle}(n, k)$ for moving n disks from a source to a destination peg when it has k spare pegs available. Here $A' = Alg_{k'}(n')$ can be any algorithm you like for moving n' individual disks from a source to a destination peg when it has k'

spare pegs available. When $k \leq \log_2 n$, we will plug into A' the standard three peg Towers of Hanoi algorithm. When $k \geq \log_2 n$, we will plug the obvious algorithm that has an infinite number of pegs. When $k = \log_2 n$, the two algorithms are the same.

The algorithm is recursive. Though I have hammered into you not to trace out recursion or to micro manage all the friends, we will first ask you to do this. Later questions will redo this same recursive algorithm using the friends paradigm.

- (a) Suppose you have $n = n_1 \times n_2$ disks and you partition them into n_1 super disks each containing n_2 disks. Suppose you want to move these using the standard three pegs, but we change the rules so that you are allowed to move a super disk of n_2 disks each super time step. Determine the asymptotic complexity of the number of these super steps needed in the following two ways:
 - 1) What is the time to move n_1 disks.
 - 2) If I am able to move n_2 disks in one time step, then my friend is left with $n - n_2$ pegs so the recurrence relation for you to solve is $T(n) = 2T(n - n_2) + 1$.
- (b) Now suppose you have a 1^{st} friend who has a great algorithm for moving n_1 super disks in time $T_1(n_1)$ and many 2^{nd} friends who are able to move n_2 disks in time $T_2(n_2)$. Suppose their work does not interfere with each other. How would you combine these in the obvious way? How many disks n get moved by this process and how much time $T(n)$ does it take?
- (c) Recall that in Towers of Hanoi, I must move the disks one at a time without putting a disk on top of one that is smaller. In order to implement the above algorithm, we need to ensure that there are enough spare pegs available so the work of the friends do not interfere with each other. A peg is considered spare for a friend if at the beginning of it's work the peg contains no disks smaller then the those being moved by the friend and in the end is left unchanged by the friend. On instance $\langle n, peg_{source}, peg_{dest}, k \rangle$ you will have your source peg, your destination peg and k spare ones for a total of $k+2$ pegs. Because your 1^{th} friend is solving your problem, she has the same source and destination as you. Suppose in addition, she needs k_1 spare pegs. Suppose each of your 2^{nd} friends in addition to there own source and destination have k_2 spare pegs. How many 2^{nd} friends do you have? Which of these friends can share pegs without interference? What is the total number of spare pegs k that you need for your computation?
- (d) We will now design $Alg_{\langle A', k', n' \rangle}(n, k)$ by looking at the tree of stack frames and micromanaging what each friend does. Let $A' = Alg_{k'}(n')$ denote your favorite algorithm for moving n' individual disks from a source to a destination peg when it has k' spare pegs available. Let $T' = T'(n') = Time(A')$ denote the number of disks that it moves. Let d be a parameter that we will

determine later. We generalize the previous two level idea into d levels of recursion.

For each $d'n[0, d-1]$, we will consider “super d' disks” and “super d' friends”. Each such friend could use a different algorithm to complete her task, with different numbers of disks $n_{d'}$ and different numbers of pegs $k_{d'}$. However, for simplify let's give each the same, namely each friend will use algorithm A' for moving n' super disks with k' spare pegs, in time T' .

Begin answering this question by explaining how to partition the set of individual disks into a hierarchy of super disks. Which friend handles which stack frame in the tree of stack frames? I have more or less given the answer already, but just to be sure, explain the task of each super d' friend, how she does it, and who she gets help from. Do not worry about bounding numbers, because is is done in later questions.

- (e) Now describe this same recursive algorithm $Alg_{\langle A', k', n' \rangle}(n, k)$ using the friends paradigm, namely just tell us what the top level player does and how she gets help from her friends. In order to set up the recurrence relation $T(n) = aT(\frac{n}{b}) + n^c$, determine the number of friends a she has, the factor b by which each the friends instance is smaller, and the number n^c of individual disks that she personally moves. What is the base case?
- (f) Generalize your two level answers from questions 17 and 17 to d level answers in order to determine the total number k of spare pegs used, the total number n of disks moved, and the total time $T_k(n)$ as functions of our fixed parameters k', n', T' , and d .
- (g) Questions 17 defined k, n , and $T_k(n)$ in terms of our fixed parameters k', n', T' , and d . Turn this around. Given k, n, k' , and $T'(n')$, determine d, n' , and $T_k(n)$.
- (h) Given k', n' and T' , determine the parameters a, b , and b in the recursive relation $T_k(n) = aT_k(b) + n^c$ and solve it using $T_k(n) = \theta\left(n^{\frac{\log a}{\log b}}\right) = \theta\left(a^{\frac{\log n}{\log b}}\right)$. Show that this is the same answer as obtained in a previous question.
- (i) We will now construct the algorithm $Alg_{\langle A', k', n' \rangle}(n, k)$, by plugging into A' the standard three peg, $k'=1$, Towers of Hanoi algorithm on some n' disks. Section 11.4 gives that this requires moving $T'(n') = 2^{n'} - 1 \leq 2^{n'}$ disks. Given $k, n, k'=1$, and this $T'(n')$, determine d, n' , and $T_k(n)$. Determine d, n' and $T_k(n)$ when $k=1, k=2, k=\log_2 n$, and $k > \log_2 n$ spare pegs are available. Note the ones that you should know already. Hint: $n^z = 2^{\log_2(n)z}$.
- (j) Suppose a friend needs to move n' disks from peg_{source} to peg_{dest} , but wants to do it as fast as possible using as many extra pegs as necessary. Give your algorithm. How many times $T'(n')$ do you have to move a disk and how many extra pegs k' will you need?
- (k) Finally, we will construct the algorithm $Alg_{\langle A', k', n' \rangle}(n, k)$, by plugging into A' the algorithm from questions 17.

Solving all of these equations to determine the running time is surprisingly hard. I recommend the following steps. Given k and d , determine k' . From this k' , your answer to questions 17 determines n' . Ignore any additive constants. From n' and d , determine n . Plug in your n' in terms of k and d . Solve for the significant d in terms n , k , and the insignificant d . Approximate this significant d by ignoring the insignificant d . Note that plugging this approximation of the significant d as your value of the insignificant d does not change anything significantly to worry about. Your answer to questions 17 determines $T'(n')$. Approximate $T_k(n)$.

Determine d , n' and $T_k(n)$ when $k = n$. Note whether you should know this already.

Note that when $k' = 1$, this algorithm and that from questions 17 are the same. In this case, determine k' , n' , n , d , and k . What happens to this algorithm, when the number k of pegs is smaller than this computed k ?

II.5. Recursion Inversions: Given a sequence of n distinct numbers $A = \langle a_1, a_2, \dots, a_m \rangle$, we say that a_i and a_j are inverted if $i < j$ but $a_i > a_j$. The number of inversions in the sequence A , denoted $I(A)$, is the number of pairs a_i and a_j that are inverted. $I(A)$ provides a measure of how close A is to being sorted in increasing order. For example, if A is already sorted in increasing order, then $I(A)$ is 0. At the other extreme, if A is sorted in decreasing order, every pair is inverted, and thus $I(A)$ is $\binom{n}{2}$. Describe a divide and conquer algorithm that finds $I(A)$ in time $\Theta(n \log n)$. You should assume that the sequence A is given to us in an array such that we can test the value of a_i in unit time, for any i . Note that the obvious algorithm is to test all pairs a_i and a_j ; that algorithm runs in $\Theta(n^2)$ time. Hint: You should modify merge sort to also compute $I(A)$. The work each stack frame requires is an iterative algorithm. Describe this iterative algorithm using loop invariants.

II.6. Recursive Forward and Back Propagation: See Chapter 30 to learn more about machine learning. During gradient descent, training inputs are fed forward layer by layer until the output is produced at the output layer. The derivatives are computed backward layer by layer starting at the output layer and ending at the input layer. The problem is that this process can take a lot of time and memory. Sometimes there is a trade-off between time and memory.

- I will give you an algorithm that uses lots of memory and the optimal amount of time.
- You will find an interactive (loop invariant) algorithm that uses the optimal amount of memory and lots of time.
- You will find a recursive algorithm that uses almost the optimal amount of memory and almost the optimal amount of time.

We model this with a path of n nodes, many worker gnomes, and a single queen gnome. Having a worker gnome on node i corresponds to having all $O(dm)$ of the forward propagation information stored for that node. Having the queen gnome

on node $n-i$ corresponds to having all $O(dm)$ of the backward propagation information stored for that node.

Precondition: The precondition is that there is a worker gnome on the first node and the queen is on the n^{th} node.

Postcondition: The postcondition is that the queen has moved to the first node.

The rules are as follows.

- If you have a worker gnome on node i , then in one step you move it to node $i+1$.
(This corresponds to the input information being fed forward from the i^{th} layer of the neural net to the $i+1^{st}$ layer.)
- If you have a worker gnome on node i , then in one step you can move a different worker gnome from anywhere on or off the board to join it at node i .
(This is so that one gnome can remain on node i while the other moves to node $i+1$.)
(Each gnome corresponds to a block of memory. This move corresponds to copying one such block into another.)
- You can remove any worker gnome from the board.
(This corresponds freeing up memory by deleting some information currently stored.)
- The queen needs a worker to be able to move. If you have both the queen and a worker on node i , then in one step you can move the queen to node $i-1$.
(This corresponds to the derivatives being computed backward from the i^{th} layer to the $i-1^{st}$ layer. The problem is that this computation also depends on the information computed during the forward propagation about this node.)
- Hint: There is never any need to have more than one queen.
- Hint: Given our task is to move the queen backward, you might as well delete all workers that are on nodes i and higher.
- Sorry. The workers gnomes cannot move backward and the queens cannot move forward.
(This corresponds to the fact that you can compute $y=f(x)$ from x , but you can't reconstruct x from y .)
- The game time used is the number of these steps.
The game memory used is the number of worker gnomes used (i.e., maximum number of worker gnomes on the board at one time).
(Hint: Game time/memory depends only on n . Actual time/memory is $O(dm)$ times this because there are d data items and the neural net is m nodes wide.)

0. My Algorithm – Lots of Memory and Optimal Time

If everything is remembered, then it takes n worker gnomes and $2(n-1)$ game time. First the worker gnome walks from node 1 to n , leaving a copy on each

node. Then the queen walks from node n to 1. She can do this easily because there is a worker gnome on each node.

1. Your Iterative Algorithm – Optimal Memory and Lots of Time

If the minimum is remembered, then it can be done with very, very few worker gnomes.

Loop Invariant: The loop invariant is that there is a worker gnome on the first node and the queen is on the i^{th} node.

Develop this algorithm and compute the amount of game time/space used.

1.1 Establishing the LI

1.2 Maintaining the Loop Invariant

1.3 Establishing Postcondition

1.4 Game Memory/Time: What is the bigOh of the total time and memory used by your algorithm for the game? Is this memory optimal?

2. Your Recursive Algorithm – Almost Optimal Memory and Almost Optimal Time:

Precondition: The precondition that there is a worker on node i and the queen is on a later (or same) node j .

Postcondition: The postcondition is that the queen has moved from node j to node i .

Memory/Time: Denote the game memory used by this friend by $M(j - i + 1)$ and the game time used by $T(j - i + 1)$ for appropriate functions $M(n)$ and $T(n)$.

Your job is to compete the following steps.

Code is not needed.

2.1 Base Case

$\langle \text{Instance} - \text{Pre} \rangle \ \& \ \langle \text{too-small} \rangle \ \& \ \langle \text{code?} \rangle \Rightarrow \langle \text{Instance-Post} \rangle$

2.2 Maintaining Precondition

$\langle \text{Instance-Pre} \rangle \ \& \ \text{not } \langle \text{too-small} \rangle \ \& \ \langle \text{code?} \rangle \Rightarrow \langle \text{SubInstances-Pre} \rangle \ \& \ \langle \text{SubInstances-Smaller} \rangle$

2.3 Trust Your Friend

$\langle \text{SubInstances-Pre} \rangle \ \& \ \langle \text{SubInstances-Smaller} \rangle \ \& \ \langle \text{code?} \rangle \Rightarrow \langle \text{SubInstances-Post} \rangle$

2.4 Maintaining Postcondition

$\langle \text{SubInstances-Post} \rangle \ \& \ \langle \text{code?} \rangle \Rightarrow \langle \text{Instance-Post} \rangle$

2.5 Time: Give the recurrence relation for T and explain where it comes from.

2.6 Memory: Give the recurrence relation for M and explain where it comes from.

2.7 Solving Recurrences: Solve the recurrence relations for M and T using the master theorem.

Are these “Almost Optimal”?

II.7. Nuts and Bolts: You have a bag of n nuts and a bag of n bolts. The nuts and bolts are all of different sizes, but for each nut there is a matching bolt (and vice versa). What you want to do is to match them. Unfortunately, you can't compare a nut with a nut or a bolt with a bolt (your eyes are too weak).

- (a) I want you to argue how many comparisons the best algorithm needs when the only thing you can do is to compare a nut with a bolt and see yes or no whether or not they match.
- (b) Now, suppose what you can do is to compare a nut with a bolt and see if the bolt is too small, too big, or just right. I want you to describe a simple algorithm that uses randomness and recursion. It should be very similar to Quick Sort, which partitions with a pivot. See [Example 12.1.2](#) for the algorithm and for its $O(n \log n)$ running time.

Hint: Use the fact that every nut matches some bolt.

II.8. Iterative Pebbles (Loop Invariants): (Problems II.8, II.9, II.10 should be done together.)

For many computational problems there is a trade-off between the time and the amount of memory needed. There are algorithms that use minimal time but lots of memory and algorithms that use minimal memory but lots of time. One way of modeling this is by a game of pebbling the nodes of a directed acyclic graph (DAG). Each node represents a value that needs to be computed. Each pebble corresponds to a memory register. Having a pebble on a node represents the fact that that value is stored in that memory register. Pebbles can be removed from a node and placed elsewhere. This corresponds to replacing the value in the corresponding register with a newly computed value. However, there is a restriction. If there are edges from each of the nodes $In(v) = \{u_1, u_2, \dots, u_d\}$ to v , then there needs to be a pebble on these nodes $u_1, u_2, \dots, u_d$ before you can put a pebble on v . This is modeling the fact that you need the values associated with $u_1, u_2, \dots, u_d$ in order to be able to compute the value associated with v . In contrast, a pebble can be placed at any time on a leaf of the DAG because these nodes have no incoming edges. These correspond to the easy-to-compute values. The input to the pebbling problem specifies a DAG and one of its nodes. The goal is to place a pebble on this node.

As a working example, consider the DAG consisting of a rectangle of nodes w wide and h high. Here h is much smaller than w . For each node v (not on the bottom row), there will be d nodes $In(v) = \{u_1, u_2, \dots, u_d\}$ on the previous row with directed edges to v . Here, d is some small value. See [Figure 17.2](#).

- (a) Describe an extremely simple iterative algorithm that puts a pebble on the specified node as quickly as possible. The algorithm may use as many pebbles (memory) as needed, but does not redo any work. Give the key steps for developing an iterative algorithm as outlined in the class and handouts, i.e., start by defining the loop invariant.
- (b) How much time do you need for this iterative programming algorithm? Given that you are able to reuse pebbles, how many do you really need?

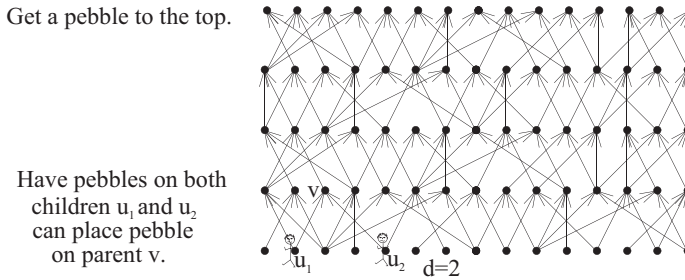


Fig. 17.2

II.9. **Recursive Pebbles:** The problem is identical to that in question II.8. Your task is to place a pebble on some node v which is r rows from the bottom. See Figure 17.3.

- Give **code** for a recursive algorithm that solves this problem. Your algorithm can redo work as necessary, but should use as few pebbles as possible. Your code should describe not only when a pebble should be placed on a node, but also when a pebble should be removed from node. Remember to get help from your friends.
- Let $Time(r)$ be the time used by your algorithm to go from there being no pebbles on the DAG to placing a pebble on one node r rows from the bottom. Give and solve a recurrence relation for this.
- Let $Pebbles(r)$ be the numbers of pebbles used by your algorithm. Argue why the recurrence relation is $Pebbles(r) = Pebbles(r-1) + (d-1)$. Solve this recurrence relation.
- Suppose the input for your recursive algorithm is not the matrix as described above but is a tree with n nodes. As before, each node (except leaves) has d children pointing at it. The height is h . What is the running time of this recursive algorithm?

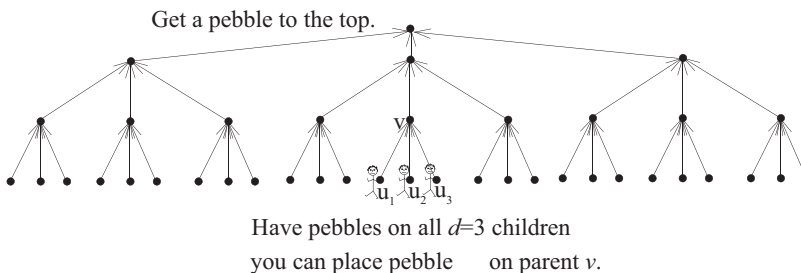


Fig. 17.3

II.10. **Dynamic Programming Pebbles:** The problem here is just slightly different from that in questions II.8 and II.9. The input is the same rectangle graph except each edge has a positive weight. The *volume* of a path from a node v to the

bottom is the *product* of the weights along the path. The task, given a node v , is to find the path from v to the bottom with largest volume.

- (a) Give the bird and friend algorithm for this. To make things more consistent with the earlier questions, ask some questions about the beginning of the solution.

Give code for the dynamic programming algorithm.

- (b) Suppose the weights on the edges could be negative. Does the same dynamic programming algorithm still work? Explain why. In contrast, does the dynamic programming algorithm for shortest weight in a level graph work when the edges could be negative? Explain why.

If you said yes, give a formal proof that it works. If you said no, give a simple way of fixing the dynamic programming algorithm to return the correct answer.

- II.11. **Recursive Probability:** I posted on Facebook that I had a set of speakers to give away. Three friends, Kunle, Hilary, and Michael, said they wanted them. I decided to flip a coin to fairly decide who gets them. The question is how to do this. A dice would have been easier. Why? If you are not up on probability, think about the space of probability one as being a piece of paper of area one. I have a way of taking any piece of paper and cutting its area in exactly half. This gives pieces of area $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots, \frac{1}{2^r}$. Then I must give each of my friends a pile of pieces whose area sums to $\frac{1}{3}$. If, for example, I cut each piece r times, then I have 2^r pieces of the same size. But 2^r can't be divided evenly into three. Problem. To warm up, let's pretend that Sam responded too. The following then would be the solution.

algorithm *Choose4*()

<pre-cond>: There are no inputs, but we do flip coins.

<post-cond>: We output Kunle, Hilary, Michael, or Sam with equal probability

begin

 flip a coin twice getting one of HH, HT, TH, TT.

 select:

 get HH: return(Kunle)

 get HT: return(Hilary)

 get TH: return(Michael)

 get TT: return(Sam)

end algorithm

- (a) Change this program into a recursive program that chooses fairly between the first three people. Hint: If Sam gets it, he can give it away.
- (b) What is the worst case number of coins that gets flipped? What is the probability that this happens.
- (c) If I get \$3 with probability p and \$7 with probability $1 - p$, then my Expected[profit] is $3 \times p + 7 \times (1 - p)$. Give a recurrence relation involving Expected[number of coin flips]. Solve this.

- (d) Suppose the program is

algorithm *Choose3*()

$\langle$ *pre-cond* $\rangle$: There are no inputs, but we do flip coins.

$\langle$ *post-cond* $\rangle$: We output Kunle, Hilary, or Michael with equal probability

begin

 flip a coin twice getting one of HH, HT, TH, TT.

 if(get HH) return(Kunle)

 else if(get TT)

 flip a coin twice getting one of HH, HT, TH, TT.

 if(get HH) return(Kunle)

 ... nothing else for Kunle

end algorithm

then Kunle is returned with the sequences HH and TTHH. The probability of this is $\frac{1}{4} + \frac{1}{4} \times \frac{1}{4} = \frac{1}{4} + \frac{1}{16} = \frac{5}{16}$. List the sequences of coin flips for which Kunle is returned in your program. What is the total probability of this? It should be $\frac{1}{3}$.

- (e) The last calculation was a little tricky. Make it easier by giving a recurrence relation involving $\text{Pr}[\text{Kunle}]$.

II.12. **Recursion Smallest:** Explain the recursive algorithm given in [Example 12.1.3](#) for finding the k^{th} smallest element in an array. Explain what the running time is.

II.13. **MaxDistance(tree, x, y):** Define the distance between two nodes in a binary tree to be the number of edges in the unique path from the one node to their lowest common ancestor and then back down to the other.

Write a recursive program that is given a binary tree and two integers x and y and outputs the maximum distance between any two nodes where one is labeled by integer x and the other by y .

For example, $\text{MaxDistance}(\text{tree}, 1, 6) = 4$ because of the 4 edge path $\langle 1, 3, 4, 9, 6 \rangle$ is longer than the 3 edge path $\langle 1, 3, 5, 6 \rangle$. Be very careful that the 6 edge path $\langle 1, 3, 5, 3, 4, 9, 6 \rangle$ is NOT considered.

The answer will be $-\infty$ if such nodes do not appear.

Be sure to check that your code works on all the examples in [Figure 17.4](#).

$\text{MaxDistance}(\text{tree}, 4, 6) = 3$
 $\text{MaxDistance}(\text{tree}, 6, 4) = 3$
 $\text{MaxDistance}(\text{tree}, 6, 6) = 5$
 $\text{MaxDistance}(\text{tree}, 5, 4) = 2$
 $\text{MaxDistance}(\text{tree}, 5, 5) = 0$
 $\text{MaxDistance}(\text{tree}, 2, 4) = 3$
 $\text{MaxDistance}(\text{tree}, 2, 10) = -\infty$

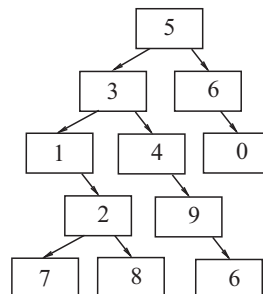


Fig. 17.4

Hint: Considering the entire tree instance, define the following concepts. They are not routines but useful definitions. These variables do not have values unless YOU assign them in your code.

- if($root.item == x$) $rootX = 0$ else $rootX = -\infty$.
 - Define $depthX_{left}$ to be the maximum depth of any node labeled by x that is in the left subtree or at the root, i.e., the number of edges in the path from the root to this node. For example, $depth(4)_{left} = 2$, $depth(4)_{right} = -\infty$, and $depth(5)_{left} = 0$.
 - You must also define values $depthX$ and $depthX_{leftsub}$.
 - Focus on the relationships between these variable values.
 - The required variable values are the maximums from a bunch of cases. Hence, use the “max” routine.
- Do not have lots of “if” statements.
- Why will I want to give you zero if you say “if x appears in the left subtree”?
 - Your code should be symmetric with respect to x and y .
 - Your code should return **three** values.

(a) (Flip between answering this problem and writing your code.)

Prove that your code $MaxDistance(tree, 2, 0)$ returns $3 + 2 = 5$ for the given tree.

Prove that your code $MaxDistance(tree, 2, 4)$ does not return $3 + 2 = 5$.

Why will I be upset and want to give you zero if you explain that

$MaxDistance(tree, 2, 4)$ returns $2 + 1 = 3$?

Prove that your code $MaxDistance(tree, 2, 4)$ returns 3.

Hint: Don’t prove that these are the required answers. Prove that **your** code returns these answers! I want to learn about **your** code, not about the problem.

(b) Give exactly one sentence arguing about the running time of your algorithm on input instance of size n .

(c) Give the recursive program.

II.14. Halting

algorithm $A(a, b)$

<pre-cond>: a and b are integers.

<post-cond>: Returns ??

begin

if($a = b$) then

return(0)

elseif($a < b$) then

return($-A(b, a)$)

else

return($1 + A(a - 1, b)$)

endif

end algorithm

algorithm $A(a, b)$

<pre-cond>: a and b are integers.

<post-cond>: Returns ??

begin

if($a = b$) then

return(0)

elseif($a < b$) then

return($-A(b, a)$)

else

return($A(a - 1, b - 1)$)

endif

end algorithm

(a) Circle the output of this program. i. $a + b$ ii. $a - b$ iii. $\max(a, b)$; iv. Does not halt. (b) Prove either that this program halts or that it does not. (c) If the program halts, prove that its output is what you claim.	(a) Circle the output of this program. i. $a + b$ ii. $a - b$ iii. $\max(a, b)$; iv. Does not halt. (b) Prove either that this program halts or that it does not. (c) If the program halts, prove that its output is what you claim.

II.15. Binary Trees

- Give the definition of a binary tree.
- What additional requirements do binary search trees have?
- Given a set of n integers, how long does it take to put them into a binary search tree? Why?
- What additional requirements do heaps have?
- Given a set of n integers, how long does it take to put them into a heap? Why?
- Give two applications of heaps.

II.16. Postfix Transversals: Suppose you are deep within the computation of “Postfix Traversal of a Binary Tree.”

- Give the output of a postfix traversal of the tree in the next question.
- Who stores a pointer to the current node in the tree?
- Who stores pointers that allow you to move down a path in the tree?
- Who stores pointers that allow you to move up the tree to the root?

II.17. AVL Tree: Recall that an AVL tree is a binary search tree with the property that every node has a balance factor of -1, 0, or 1, where its balance factor is the difference between the height of its left and its right subtrees. You are to give pseudocode for an *recursive* algorithm that is given a binary tree and returns whether or not it is an AVL tree. You do not have to prove its correctness. Also describe the algorithm using a paragraph including all required aspects of loop invariants, of recursive/friends, and of subroutines.

II.18. Min in Max Path: Consider a binary tree of integers T . Let $P(T)$ denote the set of paths p from the root of T to a leaf of T (a leaf is a node with no children). For each such path p , let m_p denote the minimum element on the path. We define the MaxMin value of T to be the maximum m_p for all such paths p . More formally,

$$\text{MaxMin}(T) = \max_{p \in P(T)} \min_{v \in p} \text{Val}(v).$$

Examples (Figure 17.5):

- In Tree (a), the smallest element on the first path $\langle 7, 3, 5 \rangle$ is 3. The smallest element on the second path $\langle 7, 3, 2 \rangle$, is 2. The smallest element on the third path $\langle 7, 4, 8 \rangle$ and fourth path $\langle 7, 4, 6 \rangle$ is 4. The largest of these four smallest values $\{3, 2, 4, 4\}$ is 4.

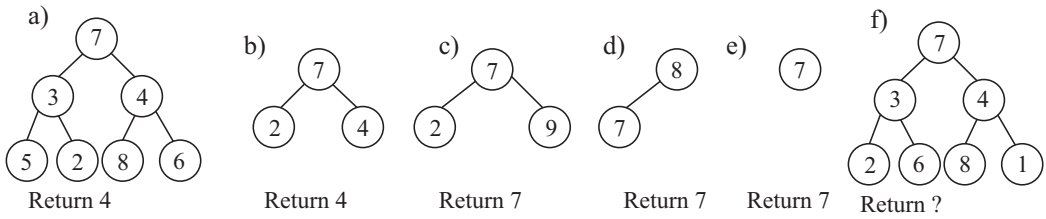


Fig. 17.5

- In Tree (b), the smallest element on the path $\langle 7, 2 \rangle$ is 2 and the smallest element on the path $\langle 7, 4 \rangle$, is 4. The largest of these two smallest values $\{2, 4\}$ is 4.
 - In Tree (c), the smallest element on the path $\langle 7, 2 \rangle$ is 2 and the smallest element on the path $\langle 7, 9 \rangle$, is 7. The largest of these two smallest values $\{2, 7\}$ is 7.
 - In Tree (d), there is only one path $\langle 8, 7 \rangle$. Its minimum is 7. The maximum of this one value is 7. Note that there is no path to the right because there are no leaves that way.
 - In Tree (e), there is only one path $\langle 7 \rangle$. Its minimum is 7. The maximum of this one value is 7.
- (a) What is the maxmin value for Tree (f)?
- (b) Give the code for $MaxMin(T)$.
- (c) Provide a quick paragraph explaining why the algorithm works.
- (d) Check your code on these inputs.

II.19. **Recursion on Trees:** Design a recursive algorithm $SearchBSTforList(tree, keyToFind, a, b)$ for the following problem. An input instance is $\langle tree, keyToFind, a, b \rangle$, where $tree$ is a binary search tree, $keyToFind$ is a key that may or may not be in the tree, and a and b are both integers greater or equal to zero. The output is $\langle A, B \rangle$ where A is the sorted list containing the a largest elements in $tree$ (if they exist) that are less than or equal to $keyToFind$ and B is the sorted list containing the b smallest elements in $tree$ (if they exist) that are greater than $keyToFind$. As an example, suppose that $tree$ contains the elements $[1, 3, 4, 7, 8, 11, 14, 21, 25, 29]$, then $SearchBSTforList(tree, 11, 3, 2)$ returns $A = [7, 8, 11]$ and $B = [14, 21]$ and $SearchBSTforList(tree, 50, 3, 7)$ returns $A = [21, 25, 29]$ and $B = []$.

Structure: I find it easier to forget about the tree structure of $tree$ and as done here think of $tree$ as a list of elements. Here, if $rootKey(tree) = 14$, then $leftSub(tree) = [1, 3, 4, 7, 8, 11]$ and $rightSub(tree) = [21, 25, 29]$.

Hand in the following:

- (a) Give the code.
- (b) The number of nodes n in $Tree$, the depth d of $Tree$, and the size $a + b$ of the required output all play a role in the running time of your algorithm.

Here we will assume that the operations $|A'|$, $A' + A''$, and $A' + e$ all take constant time.

- i. Argue about the running time of your algorithm on input instance $\langle tree, keyToFind, \infty, \infty \rangle$.
- ii. Similarly bound the running time when input instance is $\langle tree, keyToFind, 1, 0 \rangle$.
- iii. The running time has to be at least the size of the output which is to be $a + b$. If the input b is given as a $\log(b)$ bit string, then is this time complexity exponential?
- iv. Let $q = |A| + |B|$ be the number of values in the output on instance $\langle tree, keyToFind, a, b \rangle$. Express an upper bound on q in terms of n , d , a , and b .
Let T be the number of stack frames in the computation. Express (guess) an upper bound on T in terms of n , d , a , b , and q . No explanation needed.

II.20. Heap

- (a) Define a heap.
- (b) The routine *Heapify* takes a binary tree such that the left and right subtrees are heaps and returns a heap. Use invariants to sketch a recursive algorithm for *Heapify* and to justify its correctness. (No code.)
- (c) A “Priority Queue” is an abstract data type. Just like Stacks and Queues, this type consists of a list of items. Unlike Stacks and Queues, each item in a Priority Queue has a “Priority” associated with it. The operations on this type are:
 - i. Put a new item along with its priority into the Priority Queue.
 - ii. Remove the item from the Priority Queue that has the highest priority.
 Answer the following:
 - i. How would you use a heap to implement a Priority Queue? In English, describe the algorithm for:
 - A. Inserting.
 - B. Deleting.
 - ii. Using BigOh notation, what is the average time complexity of inserting and deleting an item from such a Priority Queue?
- (d) What is the ordering imposed on the elements of a max heap?
- (e) Given a binary max heap A , provide concise pseudocode for an $O(1)$ algorithm that prints the biggest 3 elements in the heap in non-increasing order. You may make the following assumptions:
 - i. The heap is implemented as a linear array.
 - ii. Operations $LEFT(i)$ and $RIGHT(i)$ return the indices of the left and right subtree of the node stored at index i of the array.
 - iii. The heap contains at least 10 nodes.
- (f) Consider the problem of determining the *smallest* element in a binary max heap. Derive a tight lower bound for this problem and prove that it is a tight lower bound.

II.21. **Recursive Synchronize:** Consider a line of n identical finite state automata that in synced time steps pass messages left and right along the line. The number of states each has cannot depend on n , hence they can't "know" the length of the line. Being identical, they can't know where they are positioned in the line. For example, maybe each of them has 12 states. When one is in the 3rd such state, receives the message 2 from the left and 8 from the right, then they send the message 3 to the left and nothing to the right and go into state 9. Because there are only 12 states and each node is running the identical code, there can be at most 24 different types of messages sent or received. Note with 12 states, a node can count up to 12 but can't count higher. The format of the answer, however, is simple sentences. Do NOT give state transitions. Nor does the answer really need code.

- (a) As a warmup, solve the following problem. NOT RECURSIVE. The **precondition** is that the automata at one of the ends has been signaled to *start* and the one at the other end knows that it is the other end but has not been signaled to start. Each of the other automata is in its *asleep* state. The **postcondition** is that after some length of time depending only on n , the one or two automata in the middle simultaneously know that they are in the middle. All other automata go back to the *asleep* state. Below, the automata marked with a 0 are the ones in the middle.

x x x 0 x x x x x x 0 0 x x x

Hint: The starting automata sends two messages that get relayed along the line, one that travels at speed one and one at speed $\frac{1}{c}$. When the fast message hits the other end, the end automata bounces it back. What should c be, how is a message sent at speed $\frac{1}{c}$, and how is the problem solved? (My answer has four sentences and does not give anything that looks like code or DFA state transitions. Neither does it worry about n plus or minus c details such as whether there is one or two guys in the middle.)

- (b) Recursive Fire Squad: The **precondition** is the same as in the previous question. The *postcondition* is that after some length of time depending only on n , all automata simultaneously go into the *fire* state. Design a recursive algorithm for this (as emphasized in step 2). (My answer has twelve sentences and does not give anything that looks like code or DFA state transitions.)
- (c) How much time is there from when the leftmost automata is signaled to start and when they all fire? Be careful to note that some of the actions happen in parallel.

II.22. **RIL:** Define a *recursive integer list* (RIL) to be a list of a finite number of objects where each object is either an integer or is an RIL. For example, $L = [3, 2, [4, 3], [[3, 9], [], 8], 7]$ is an RIL. Give the pseudocode for a recursive algorithm *RILSum* that given an RIL L , returns the sum of all the integers appearing in it. For example, $RILSum(L) = 3 + 2 + 4 + 3 + 3 + 9 + 8 + 7$. No explanation or running time needed.

- II.23. **Recursive Find the Key:** My dear friend Chester told me this problem. I still have not looked at the solution from the video, but I found a sweet recursive solution. Then I found an even easier solution. I would love to share these with you.

Prison Game: The game has three players: The “encoding” prisoner and the “decoding” prisoner are friends. Together, they want to find the key to freedom. The prison warden wants to prevent them from escaping. But the warden does like to play games with his victims. He hides the key in one of n boxes. The encoder knows its location and must communicate this to the decoder. But how they are able to communicate is very limited. The warden labels each box with a bit, which he tells the encoder. The encoder communicates to the decoder by flipping the bit on one of the boxes. The decoder learns only the labelings with the one bit flipped. He does not learn, for example, which bit was flipped. From this alone, the decoder must then be able to determine which of the n boxes contains the key.

Your job is to write two recursive programs, Encoder and Decoder.

Encoder(n)

Precondition: The encoder, who we will refer to as `encoder( $n$ )`, learns the following:

- The number of boxes n .
- The index i of the box containing the key.
- The vector of n bits labeling the boxes as provided by the warden.

Postcondition: `Encoder( $n$ )` returns:

- The index i' of the box whose bits will be flipped.

Decoder(n)

Precondition: The decoder, who we will refer to as `decoder( $n$ )`, learns the following:

- The same vector of n bits given to `encoder( $n$ )` except with the bit specified by `encoder( $n$ )` flipped.

Postcondition: `Decoder( $n$ )` returns:

- The location i of the key, as told to `encoder( $n$ )`.

$n = 2$: With one box, you know where the key is, so let's consider when you have two.

Encode

Precondition: Let x_0 denote the bit on the zeroth box and x_1 on the first. Let $i \in \{0, 1\}$ indicate whether the key is in box 0 or in box 1.

Postcondition: If $x_1 = i$, then the `encoder(2)` says to flip bit x_0 . Otherwise, he says to flip x_1 .

Decode

Precondition: Let x'_0 and x'_1 denote bits on the boxes after the bit specified by the encoder(2) has been flipped.

Postcondition: One of your **test questions** is to specify the actions of decoder(2) and to argue that this $n = 2$ algorithm works.

Hints

- You know you are to use recursion. Given that the input consists of some n boxes, the obvious thing is to give the first half of these boxes to your first recursive friend and the second half to your second. This, however, does not work. Instead, encoder(n) will name his recursive friends encoder(n_1) and encoder(n_2), giving them, respectively, n_1 and n_2 boxes, where $n = n_1 \times n_2$. Decoder(n) will do the same. One of your tasks is to determine for which factorization $n = n_1 \times n_2$ your program with n boxes works, assuming of course that these recursive friends solve their problem correctly.
- The encoder(n) and decoder(n) will each take their n bit vector and arrange the bits into a matrix with n_1 rows and n_2 columns. Then each will determine the parity of each row and of each column. (The parity is 1 if the number of 1s is odd.) The following is an example:

0	1	1	1	0	1
1	0	1	0	0	0
0	1	0	1	0	0
1	1	0	0	1	1
0	1	0	0	1	

$n = 20, n_1 = 4 \text{ and } n_2 = 5.$

The following are your questions:

Hint: You might want to reread my recursive steps.

- Finish the $n = 2$ case and prove that it works.
- Explain what encoder(n) does with the help of encoder(n_1), and encoder(n_2).

Hint: If you panic, you will find this hard. If you simply guess, I am quite sure you will figure out the inputs and outputs to these three programs.

- Explain what decoder(n) does with the help of decoder(n_1), and decoder(n_2).

Hint: Same as last hint.

- In order to prove that the decoder(n) algorithm works, you need to prove that if the instance it is given meets its precondition, then so does the subinstance it gives to each of its recursive friends, decoder(n_1) and decoder(n_2). More specifically, suppose that as required, decoder(n) is given the same vector of n bits that were given to encoder(n) except with the bit specified by

encoder(n) flipped. Prove that as required, decoder(n_1) is given the same vector of n bits that were given to encoder(n_1) except with the bit specified by encoder(n_1) flipped.

- (e) Assuming that the recursive friends with n_1 and n_2 boxes, respectively, correctly solved their problem, argue why encoder(n) and decoder(n) together correctly locate the key. Is this true for every factorization $n = n_1 \times n_2$?

Hint: My solutions has five quite straightforward sentences.

Hint: I did not use the word “prove” in order to avoid mass panic.

- (f) Characterize the values of n for which your algorithm works and for which it does not. For example, does the algorithm work for $n = 3$? Explain why.
- (g) What is the running time for this algorithm? Does this time depend on the choice of the factorization $n = n_1 \times n_2$?
- (h) Oh dear. I was trying to understand the base cases of the above algorithm and I came up with a much easier algorithm. Instead of laying out the n boxes into an $n_1 \times n_2$ rectangle, spread it into a $\log n$ dimensional hypercube. Don't worry about this. Just name each of the n boxes with a unique $\log n$ bit binary vector. For example, if $n = 16$, then the first box will be named $\langle 0, 0, 0, 0 \rangle$ and the last box will be named $\langle 1, 1, 1, 1 \rangle$. We will add these vectors by taking the parity in each of the dimensions/columns, i.e., bit-wise xor. For example, $\langle 1, 0, 1, 0 \rangle + \langle 1, 1, 0, 0 \rangle = \langle 0, 1, 1, 0 \rangle$. One funny thing is that if you add one of these vectors to itself, you get the zero vector, namely, $\langle 1, 0, 1, 0 \rangle + \langle 1, 0, 1, 0 \rangle = \langle 0, 0, 0, 0 \rangle$. This helps to show that $X + Y = X - Y$ and $X + Y + Y = X$.
- i. Now each box has a vector name, a bit label given by the warden (one of which is flipped by encoder(n)), and maybe contains the key. Both encoder(n) and decoder(n) will add up the vectors of the boxes that have label 1. Let E and D denote these sums. Let F denote the vector name of the box whose bit encoder(n) flips. Give an equation giving D as a function of E and F .
 - ii. Let K denote the vector name of the box containing the key. Give a simple equation with which decoder(n) will determine K from D .
 - iii. Given this, give an equation with which encoder(n) will determine F from E and K .
 - iv. Characterize the values of n for which your algorithm works and for which it does not. Explain why.

18 Partial Solutions to Additional Exercises: Part II

II.1. **Short Answers:** The generic reason a recursive algorithm works is:

- i) When our instance meets the precondition, the instances that we give our friends meet the precondition and are smaller.
- ii) We then trust our friends to give us a solution to their instance that meets the post condition.
- iii) We then combine their solutions to get a solution to our instance that meets the post condition.
- iv) Unless our instance is sufficiently small, in which case we solve it ourselves.

II.2. **Trace Eval**

- (a) $T(0) = 1$ because the base case just returns x .
The recurrence is $T(n) = T(n-1) + T(n-1) + 1 = 2T(n-1) + 1 = 2^{n+1} - 1$.
The answer of $O(2^n)$ is just fine.
Not asked for but the value returned is $f(0) = x$ and $f(n) = f(n-1) + f(n-1) = 2f(n-1) = 2^n x$.
- (b) The algorithm as given does DFS but visits nodes more than once. Change the algorithm so that it drops bread crumbs, storing the answer at each node so that this answer does not need to be recomputed.
The running time would then be $\Theta(E)$ just like DFS.

II.3. **Planar Graphs**

- (a) Claim a: If G is a tree, then $E = N - 1$, $F = 1$ and $C = 1$. Two proofs:
 - (i) The base case is that, if the graph has $N = 1$ node and $E = 0$ edges, then the statement is true.
Given a rooted tree G , I sit at the root. For each i , I give my i^{th} subtree to my i^{th} recursive friend who returns N_i , E_i , and assures that $E_i = N_i - 1$. The number of nodes in G is $N = (\sum_i N_i) + 1$ when you count the root as well.
The number of edges in G is $E = \sum_i (E_i + 1)$ when you count the parent edge for each subtree as well.
Using our friend's assurances that $E_i = N_i - 1$, we get $E = \sum_i (E_i + 1) = \sum_i ((N_i - 1) + 1) = ((\sum_i N_i) + 1) - 1 = N - 1$, which is what we want to assure.
 - (ii) Again, the base case is that, if the graph has $N = 1$ node and $E = 0$ edges, then the statement is true.

We proved above that each tree G has at least one leaf. Form graph G' by deleting one such node and its one edge. Give this to my one recursive friend, who returns N' , E' , and assures that $E' = N' - 1$.

The number of nodes in G is $N = N' + 1$ because we deleted one node.

The number of edges in G is $E = E' + 1$ because we deleted one edge.

Using that our friend's assurances that $E' = N' - 1$, we get $E = E' + 1 = (N' - 1) + 1 = ((N - 1) - 1) + 1 = N - 1$ which is what we want to assure.

- (b) For every connected planar graph, $E = N + F - 2$. Proof by induction on the number of faces: If $F = 1$, then graph is a tree and $E = N - 1 = N + F - 2$ is true by the first claim. Suppose by way of induction it is true for connected planar graphs with fewer than F faces. Consider a connected planar graph G with F faces. The union of all the faces tile the entire plane. Hence, if $F > 1$, then G has two faces, f_1 and f_2 , separated by an edge u, v . Let G_1 denote the graph formed from G by deleting this edge and merging the two faces f_1 and f_2 into one face. This gives $N_1 = N$, $E_1 = E - 1$, $F_1 = F - 1$. By way of induction $E_1 = N_1 + F_1 - 2$, and hence $(E - 1) = N + (F - 1) - 2$, and hence $E = N + F - 2$ as needed.
- (c) For every planar graph, $E = N + F - C - 1$. Proof by induction on the number of components: If $C = 1$, then graph is a connected and $E = N + F - 2 = N + F - C - 1$ is true by the claim 2. Suppose by way of induction it is true for planar graphs with fewer than C components. Consider a planar graph G with C faces. Choose two of the components and form graph G_1 by adding an edge u, v between some node u in the first component and some v in the second. This gives $N_1 = N$, $E_1 = E + 1$, $F_1 = F$, and $C_1 = C - 1$. By way of induction $E_1 = N_1 + F_1 - C_1 - 1$, and hence $(E + 1) = N + F - (C - 1) - 1$, and hence $E = N + F - C - 1$ as needed.
- (d) For every planar graph, $F \leq 2/3E$.
Proof: Build a bipartite graph between the E edges of G and the F faces with an "edge" between edge u, v and face f of G if they are "adjacent." Each edge u, v has two "sides" and hence is "adjacent" to exactly two faces f . Hence, the number of "edges" in this bipartite graph is $2E$. Each face f is "adjacent" to at least three edges u, v , because the smallest number of sides a face can have is when it is a triangle. Hence, the number of "edges" in this bipartite graph greater or equal to $3F$. The result follows from $2E = \#edges \geq 3F$.
- (e) For every planar graph, there exists a node with degree 5 or less.
Proof: Combining the claims (c), (d), and that $C \geq 1$ gives that $E = N + F - C - 1 \leq N + (2/3E) - 2$. Solving this gives $13E \leq N - 2$ and that $E \leq 3N - 6$. The sum $\sum deg(u) = 2E$ because each edge u, v is adjacent to two nodes. Hence, the average degree is $2E/N \leq 6N - 12N / N < 6$. Hence, there must be at least one node with degree 5 or less.
- (f) *SixColor*(G)
 G is a planar graph
Returns a bichromatic six coloring C of G .

```

if(  $G = \text{emptyGraph}$  )
    return(  $\text{emptyColoring}$  )
else
     $u = \text{NodeMinDeg}(G)$ 
     $G1 = \text{DeleteNode}(G, u)$ 
    Deleting  $u$  and its adjacent edges from  $G$ .
     $C1 = \text{SixColor}(G1)$ 
    Let  $c$  be one color from  $\{1, 2, \dots, 6\}$  not used in  $C1$  to color one
    of  $u$ 's at most five neighbors  $\text{Neighbors}(G, u)$ .
    Let  $C$  color all the nodes in  $G$  except for  $u$ 
    in the same way as  $C1$  colors  $G1$ 
    and let  $C(u)$  be  $c$ .
    return(  $C$  )
end if

```

- (g) The key idea in a friend's algorithm is as follows. We are given a planar graph G to color. We can give our friend any instance that is smaller and meets the precondition. In this case, we give our friend the graph $G1 = \text{DeleteNode}(G, u)$ obtained by deleting one of its nodes u of minimum degree, i.e. $u = \text{NodeMinDeg}(G)$ and this node's edges. Note that $G1$ is a smaller valid instance, namely it is a planar graph with one fewer node. Hence we can trust our friend to give us the right answer for her instance, namely $C1 = \text{SixColor}(G1)$ is a valid bichromatic six coloring of $G1$. Our job is to construct the correct answer for our instance from her answer. Because her $C1$ is a valid coloring of $G1$, within $G1$ each node has a color in $\{1, 2, \dots, 6\}$ and each edge is bichromatic. Because the node u in G has degree at most 5, there is a sixth color in $\{1, 2, \dots, 6\}$ that u can be given and still have each of the edges $\{u, v_i\}$ adjacent to u be bichromatic. This proves that C is a valid coloring of G . If our input is small enough then we must do it ourselves. Here, the base case is the empty graph. We make the answer for this strange case be whatever is needed to make the bigger cases work. In this case, the coloring assigns no colors because there are no nodes to color.
- (h) It does not work. If your graph G has only one small degree node, then after deleting it, the remaining graph might not have a small degree node and it might be NP-complete to color it.

II.4. Abstract Recursion (Tower of Hanoi): Let's see how we can use recursion not only by giving your friend's a smaller instance but a stranger abstract way. Review Towers of Hanoi from Section 11.4. We will be generalizing it to having more than three pegs.

Any comparison based sorting code for sorting integers can be abstracted to handle real numbers. Any comparison between integers is replaced by the more complex comparisons between reals. Similarly, any code for moving normal disks can be abstracted to be able to move "super disks". Any moving of a disk is replaced by the moving of a super disk.

For each choice of A' , k' , and n' , we will design a Towers of Hanoi algorithm $Alg_{\langle A', k', n' \rangle}(n, k)$ for moving n disks from a source to a destination peg when it has k spare pegs available. Here $A' = Alg_{k'}(n')$ can be any algorithm you like for moving n' individual disks from a source to a destination peg when it has k' spare pegs available. When $k \leq \log_2 n$, we will plug into A' the standard three peg Towers of Hanoi algorithm. When $k \geq \log_2 n$, we will plug the obvious algorithm that has an infinite number of pegs. When $k = \log_2 n$, the two algorithms are the same.

The algorithm is recursive. Though I have hammered into you not to trace out recursion or to micro manage all the friends, we will first ask you to do this. Later questions will redo this same recursive algorithm using the friends paradigm.

- (a) Suppose you have $n = n_1 \times n_2$ disks and you partition them into n_1 super disks each containing n_2 disks. Suppose you want to move these using the standard three pegs, but we change the rules so that you are allowed to move a super disk of n_2 disks each super time step. Determine the asymptotic complexity of the number of these super steps needed in the following two ways:

1) What is the time to move n_1 disks.

2) If I am able to move n_2 disks in one time step, then my friend is left with $n - n_2$ pegs so the recurrence relation for you to solve is $T(n) = 2T(n - n_2) + 1$.

- Answer: Either way the running time is $T(n) = O(2^{n_1})$.

1) The normal algorithm takes $T(n) = O(2^{n_1})$ when given n_1 disks. These disks are now super.

2) Losing n_2 each step, the depth of recursion is $\frac{n}{n_2} = n_1$. Each step, the number of friends doubles.

- (b) Now suppose you have a 1st friend who has a great algorithm for moving n_1 super disks in time $T_1(n_1)$ and many 2nd friends who are able to move n_2 disks in time $T_2(n_2)$. Suppose their work does not interfere with each other. How would you combine these in the obvious way? How many disks n get moved by this process and how much time $T(n)$ does it take?

- Answer: You can move $n = n_1 \times n_2$ by partitioning them into n_1 super disks containing n_2 disks each. Follow the first friend's algorithm for moving n_1 super disks. Any moving of a super disk is replaced by a second friend move n_2 disks. Each of your first friend's $T_1(n_1)$ super steps actually takes time $T_2(n_2)$ for a total of $T(n) = T_1(n_1) \times T_2(n_2)$.

- (c) Recall that in Towers of Hanoi, I must move the disks one at a time without putting a disk on top of one that is smaller. In order to implement the above algorithm, we need to ensure that there are enough spare pegs available so the work of the friends do not interfere with each other. A peg is considered spare for a friend if at the beginning of it's work the peg contains no disks smaller than the those being moved by the friend and in the end is left unchanged by the friend. On instance $\langle n, peg_{source}, peg_{dest}, k \rangle$ you will have your source peg, your destination peg and k spare ones for a total of $k+2$ pegs. Because your

1th friend is solving your problem, she has the same source and destination as you. Suppose in addition, she needs k_1 spare pegs. Suppose each of your 2nd friends in addition to their own source and destination have k_2 spare pegs. How many 2nd friends do you have? Which of these friends can share pegs without interference? What is the total number of spare pegs k that you need for your computation?

- Answer: The number of 2nd friends is $T_1(n_1)$, one for each of the 1st friend's super steps. However, each of them leaves their k_2 spare pegs empty when they are complete. Hence, they can all use the same ones. Hence, the total need is $k = k_1 + k_2$.

- (d) We will now design $Alg_{\langle A', k', n' \rangle}(n, k)$ by looking at the tree of stack frames and micromanaging what each friend does. Let $A' = Alg_{k'}(n')$ denote your favorite algorithm for moving n' individual disks from a source to a destination peg when it has k' spare pegs available. Let $T' = T'(n') = Time(A')$ denote the number of disks that it moves. Let d be a parameter that we will determine later. We generalize the previous two level idea into d levels of recursion.

For each $d' \in [0, d-1]$, we will consider “super ^{d'} disks” and “super ^{d'} friends”. Each such friend could use a different algorithm to complete her task, with different numbers of disks $n_{d'}$ and different numbers of pegs $k_{d'}$. However, for simplicity let's give each the same, namely each friend will use algorithm A' for moving n' super disks with k' spare pegs, in time T' .

Begin answering this question by explaining how to partition the set of individual disks into a hierarchy of super disks. Which friend handles which stack frame in the tree of stack frames? I have more or less given the answer already, but just to be sure, explain the task of each super ^{d'} friend, how she does it, and who she gets help from. Do not worry about bounding numbers, because this is done in later questions.

- Answer: You have n disks and you partition them into n' super ^{$d-1$} disks, each which is partitioned into n' super ^{$d-2$} disks, and so on. Like any recursive program there will be a tree of stack frames. For each $d' \in [0, d]$, stack frames at level d' are executed by a super ^{d'} friend, with the single super ^{d} friend being the root and the many super⁰ friends being the leaves. Her task is to move a super ^{d'} disk from one peg to another. She interprets this as moving n' super ^{$d'-1$} disks. She does this using algorithm A' with k' extra pegs. She gets help from a super ^{$d'-1$} friend for each such super ^{$d'-1$} disk move.
- (e) Now describe this same recursive algorithm $Alg_{\langle A', k', n' \rangle}(n, k)$ using the friends paradigm, namely just tell us what the top level player does and how she gets help from her friends. In order to set up the recurrence relation $T(n) = aT(\frac{n}{b}) + n^c$, determine the number of friends a she has, the factor b by which each friend's instance is smaller, and the number n^c of individual disks that she personally moves. What is the base case?

- Answer: The root player's task is to move n disk from one peg to another. She partitions the disks into n' super disks each containing $\frac{n}{n'}$ disks. She interprets her task as moving these n' super disks. She does this using algorithm A' with k' extra peg. She gets help from a friend for each such super disk move. She needs $a = T'$ friends, one for each of the T' super moves specified by A' . The factor b by which each the friends instance is smaller is $b = n'$, because the super disks given to the friends have size $\frac{n}{n'}$. Every non-base case friends moves no individual disks, so $n^c = 0$. If $n = 1$, she just moves this disk.
- (f) Generalize your two level answers from questions 17 and 17 to d level answers in order to determine the total number k of spare pegs used, the total number n of disks moved, and the total time $T_k(n)$ as functions of our fixed parameters k' , n' , T' , and d .
- Answer: The generalization of the total number of pegs needed is $k = k_1 + k_2 + \dots + k_d = k'd$. The generalization of the total number of disks moved is $n = n_1 \times n_2 \times \dots \times n_d = n'^d$. The generalization of the total time is $T_k(n) = T_1(n_1) \times T_2(n_2) \times \dots \times T_d(n_d) = T'^d$.
- (g) Questions 17 defined k , n , and $T_k(n)$ in terms of our fixed parameters k' , n' , T' , and d . Turn this around. Given k , n , k' , and $T'(n')$, determine d , n' , and $T_k(n)$.
- Answer: Solving $k = k'd$ gives $d = \frac{k}{k'}$ levels. Solving $n = n'^d$ gives $n' = n^{\frac{1}{d}} = n^{\frac{k'}{k}}$ disks. This gives, the total time to be $T_k(n) = T'(n')^d = T'(n^{\frac{k'}{k}})^{k/k'}$.
- (h) Given k' , n' and T' , determine the parameters a , b , and b in the recursive relation $T_k(n) = aT_k(b) + n^c$ and solve it using $T_k(n) = \theta\left(n^{\frac{\log a}{\log b}}\right) = \theta\left(a^{\frac{\log n}{\log b}}\right)$. Show that this is the same answer as obtained in a previous question.
- Answer: We stated that she needs $a = T'$ friends and that inputs shrink by a factor of $b = n' = n^{\frac{k'}{k}}$ giving $\log b = \log n' = \frac{k'}{k} \log n$ and $T_k(n) = \theta\left(a^{\frac{\log n}{\log b}}\right) = \theta\left(T'^{\frac{\log n}{\log n'}}\right) = \theta\left(T'^{\frac{k}{k'}}$.
- (i) We will now construct the algorithm $Alg_{(A', k', n')}(n, k)$, by plugging into A' the standard three peg, $k'=1$, Towers of Hanoi algorithm on some n' disks. Section 11.4 gives that this requires moving $T'(n') = 2^{n'} - 1 \leq 2^{n'}$ disks. Given k , n , $k' = 1$, and this $T'(n')$, determine d , n' , and $T_k(n)$. Determine d , n' and $T_k(n)$ when $k = 1$, $k = 2$, $k = \log_2 n$, and $k > \log_2 n$ spare pegs are available. Note the ones that you should know already. Hint: $n^z = 2^{\log_2(n)z}$.
- Answer: $d = \frac{k}{k'} = k$ levels. $n' = n^{\frac{1}{d}} = n^{\frac{1}{k}}$ disks. $T'(n') = 2^{n'} - 1$.

$$T_k(n) = T'(n')^d = (2^{n'} - 1)^d = \left(2^{n^{\frac{1}{k}}} - 1\right)^k \leq 2^{kn^{\frac{1}{k}}}.$$
When $k = 1$, $n' = n^{\frac{1}{1}} = n$ and $T_1(n) = (2^n - 1)^1 = 2^n - 1$ as is expected. Having an extra peg drastically decreases these to $n' = \sqrt[n]{n}$ and $T_2(n) \leq 2^{2\sqrt[n]{n}}$. When $k = \log_2 n$, $n' = n^{\frac{1}{k}} = 2^{\frac{\log_2 n}{k}} = 2^1 = 2$ and

$T_{\log_2 n}(n) = (2^{n^{\frac{1}{k}}} - 1)^k = (2^2 - 1)^k = 3^{\log_2 n} = n^{\log_2 3}$. When $k > \log_2 n$, $n' < 2$ and the algorithm does not make sense.

- (j) Suppose a friend needs to move n' disks from peg_{source} to peg_{dest} , but wants to do it as fast as possible using as many extra pegs as necessary. Give your algorithm. How many times $T'(n')$ do you have to move a disk and how many extra pegs k' will you need?

- Answer: Move each disk from peg_{source} to a peg by itself, except for the largest which can be moved directly to peg_{dest} . Then in the reverse order stack the disks onto peg_{dest} . You must move a disk $T'(n') = 2n' - 1$ times and you need $k' = n' - 1$ extra pegs.

- (k) Finally, we will construct the algorithm $Alg_{\langle A', k', n' \rangle}(n, k)$, by plugging into A' the algorithm from questions 17.

Solving all of these equations to determine the running time is surprisingly hard. I recommend the following steps. Given k and d , determine k' . From this k' , your answer to questions 17 determines n' . Ignore any additive constants. From n' and d , determine n . Plug in your n' in terms of k and d . Solve for the significant d in terms n, k , and the insignificant d . Approximate this significant d by ignoring the insignificant d . Note that plugging this approximation of the significant d as your value of the insignificant d does not change anything significantly to worry about. Your answer to questions 17 determines $T'(n')$. Approximate $T_k(n)$.

Determine d, n' and $T_k(n)$ when $k = n$. Note whether you should know this already.

Note that when $k' = 1$, this algorithm and that from questions 17 are the same. In this case, determine k', n', n, d , and k . What happens to this algorithm, when the number k of pegs is smaller than this computed k ?

- Answer: $k = k'd$. $k' = \frac{k}{d}$. $n' = k' + 1 \approx n'd$. $n' = \frac{k}{d}$. $n = n'^d = \left(\frac{k}{d}\right)^d$.

$$d = \frac{\log(n)}{\log(k/d)} = \frac{\log(n)}{\log(k) - \log(d)} \approx \frac{\log(n)}{\log(k)}. \quad \log(d) \approx \log \log(n) - \log \log(k).$$

$$d \approx \frac{\log(n)}{\log(k) - \log \log(n) + \log \log(k)}.$$

$$T_k(n) = T'(n')^d = (2n' - 1)^d \approx (2n')^d = 2^d n \approx 2^{\frac{\log(n)}{\log(k)}} n = n^{1 + \frac{1}{\log(k)}}.$$

$$k = n, T \approx 2^{\frac{\log(n)}{\log(k)}} n = 2^1 n \text{ as needed. When } k' = 1, n' = k' + 1 = 2,$$

$n = n'^d = 2^d$. $d = \log_2 n$, and $k = k'd = \log_2 n$. When the number k of pegs is smaller than this, k' gets smaller than one, which does not make sense.

- II.5. **Recursion Inversions:** The recursive algorithm is given an array A and returns $\langle S, I \rangle$, where S is the array sorted and I is the number of inversions $I(A)$. When A contains at least two elements, the code cuts the input A into two halves A_1 and A_2 and recurses on both. Let $\langle S_1, I_1 \rangle$ and $\langle S_2, I_2 \rangle$ be their output. The code then calls an iterative algorithm that takes as input two sorted arrays S_1 and S_2 and returns $\langle S, I_{(1,2)} \rangle$, where $S = \text{merge}(S_1, S_2)$ is the merging of these and $I_{(1,2)}$ is $I(S_1 + S_2)$, which is the number of inversions in

the array containing S_1 followed by S_2 . The output of the recursive algorithm is $\langle S, I_1 + I_2 + I_{(1,2)} \rangle$.

The code for the iterative algorithm maintains three sorted arrays S'_1 , S'_2 , and S' and one integer I' . The first loop invariant is that $\text{merge}(S'_1, S'_2) + S'$ is the merged array $S = \text{merge}(S_1, S_2)$ that we need to output. The second loop invariant is that $I(S'_1 + S'_2) + I'$ is the number of inversions $I_{(1,2)} = I(S_1 + S_2)$ that we need to output. We establish this easily by setting $S'_1 = S_1$, $S'_2 = S_2$, $S' = \emptyset$, and $I' = 0$.

The step is as follows. Let $a_{\langle |S'_1|, 1 \rangle}$ be the last elements of S'_1 and $a_{\langle |S'_2|, 2 \rangle}$ be the last elements of S'_2 . There are two cases. If $a_{\langle |S'_1|, 1 \rangle} > a_{\langle |S'_2|, 2 \rangle}$, then because S_2 is sorted, $a_{\langle |S'_1|, 1 \rangle}$ is greater than all elements in S_2 . Just as in the standard merging algorithm, $a_{\langle |S'_1|, 1 \rangle}$ is removed from the end of S'_1 and added to the beginning of S . This maintains the first loop invariant. The element $a_{\langle |S'_1|, 1 \rangle}$ contributes $|S_2|$ to $I(S'_1, S'_2)$ because it creates an inversion with each of the elements of S_2 . Hence, we maintain the second loop invariant by adding $|S_2|$ to I' .

In the second case, $a_{\langle |S'_1|, 1 \rangle} \leq a_{\langle |S'_2|, 2 \rangle}$. In this case, $a_{\langle |S'_2|, 2 \rangle}$ is greater than or equal to all elements in S_1 . Just as in the standard merging algorithm, $a_{\langle |S'_2|, 2 \rangle}$ is removed from the end of S'_2 and added to the beginning of S . This maintains the first loop invariant. The element $a_{\langle |S'_2|, 2 \rangle}$ contributes zero to $I(S'_1, S'_2)$ because it creates an inversion with none of the elements of S_1 . Hence, we maintain the second loop invariant by adding nothing to I' .

The exit condition is when one of S'_1 or S'_2 is empty. This will eventually occur because the sum of their lengths decreases by one each iteration.

We establish the postcondition as follows. By the exit condition, one of S'_1 or S'_2 is empty. Assume S'_1 is empty. (The other case is the same.) This gives that $\text{merge}(S'_1, S'_2)$ is simply S'_2 and hence $\text{merge}(S'_1, S'_2) + S' = S'_2 + S'$. The first loop invariant gives that this is the required output S . Hence, the first postcondition is maintained by outputting $S = S'_2 + S'$. Having S'_1 empty and S'_2 sorted gives us that the number of inversion $I(S'_1 + S'_2)$ is zero and hence $I(S'_1 + S'_2) + I' = I'$. The second loop invariant gives that this is the number of inversions $I_{(1,2)}$ that we need to output. Hence, the second postcondition is maintained by outputting $I_{(1,2)} = I'$.

II.7. Nuts and Bolts

- (a) In an adversarial lower bound, as the algorithm compares nuts and bolts, the response is always that they do not match until some nut is compared with each and every bolt. The last bolt compared is the one matched. This is repeated for the remaining nuts and bolts. The total time is then $O(n^2)$. If you are extra smart, you might find a bug in this proof. If nut 1 and nut 2 have had a failed comparison with every bolt except the last, then they cannot be matched to both.
- (b) Choose a random nut. Compare it to each bolt, partitioning them into those that are smaller, a perfect match, and larger. At least one bolt is a

perfect match. Compare it with each nut, partitioning them into those that are smaller and larger. Match this nut and bolt. The nuts and bolts that are smaller than this pair must meet the precondition, namely, you have a bag of n' nuts and a bag of n' bolts and for each nut there is a matching bolt (and vice versa). This subinstance is also smaller. Hence, get a friend to match them. Similarly, those that are larger.

Part III

Optimization Problems

19 Definition of Optimization Problems

Many important and practical problems can be expressed as *optimization problems*. Such problems involve finding the best of an exponentially large set of solutions. It can be like finding a needle in a haystack. The obvious algorithm, considering each of the solutions, takes too much time because there are so many solutions. Some of these problems can be solved in polynomial time using network flow, linear programming, greedy algorithms, or dynamic programming. When not, recursive backtracking can sometimes find an optimal solution for some instances in some practical applications. Approximately optimal solutions can sometimes be found more easily. Random algorithms, which flip coins, sometimes have better luck. However, for most optimization problems, the best known algorithms require $2^{\Theta(n)}$ time on the worst case input instances. The commonly held belief is that there are no polynomial-time algorithms for them (though we may be wrong). NP-completeness helps to justify this belief by showing that some of these problems are universally hard among this class of problems. I now formally define this class of problems.

Ingredients: An optimization problem is specified by defining instances, solutions, and costs.

Instances: The *instances* are the possible inputs to the problem.

Solutions for Instance: Each instance has an exponentially large set of *solutions*. A solution is *valid* if it meets a set of criteria determined by the instance at hand.

Measure of Success: Each solution has an easy-to-compute *cost*, *value*, or *measure success* that is to be minimized or maximized.

Specification of an Optimization Problem

Preconditions: The input is one instance.

Postconditions: The output is one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) measure of success. (The solution to be outputted need not be unique.)

Examples

Longest Common Subsequence: This is an example for which we have a polynomial-time algorithm.

Instances: An instance consists of two sequences, e.g., $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$.

Solutions: A subsequence of a sequence is a subset of the elements taken in the same order. For example, $Z = \langle B, C, A \rangle$ is a subsequence of $X = \langle A, \underline{B}, \underline{C}, B, D, \underline{A}, B \rangle$. A solution is a sequence Z that is a subsequence both X and Y . For example, $Z = \langle B, C, A \rangle$ is solution, because it is a subsequence common to both X and Y ($Y = \langle B, D, \underline{C}, \underline{A}, B, A \rangle$).

Measure of Success: The value of a solution is the length of the common subsequence, e.g., $|Z| = 3$.

Goal: Given two sequences X and Y , the goal is to find the longest common subsequence (LCS for short). For the example given above, $Z = \langle B, C, B, A \rangle$ is longest common subsequence.

Course Scheduling: This is an example for which we do not have a polynomial-time algorithm.

Instances: An instance consists of the set of courses specified by a university, the set of courses that each student requests, and the set of time slots in that courses can be offered.

Solutions: A solution for an instance is a schedule that assigns each course a time slot.

Measure of Success: A conflict occurs when two courses are scheduled at the same time even though a student requests them both. The cost of a schedule is the number of conflicts that it has.

Goal: Given the course and student information, the goal is to find the schedule with the fewest conflicts.

Airplane: The following is an example of a practical problem.

Instances: An instance specifies the requirements of a plane: size, speed, fuel efficiency, etc.

Solutions: A solution for an instance is a specification of a plane, right down to every curve and nut and bolt.

Measure Success: The company has a way of measuring how well the specification meets the requirements.

Goal: Given plane requirements, the goal is to find a specification that meets them in an optimal way.

20 Graph Search Algorithms

An optimization problem requires finding the best of a large number of solutions. This can be compared to a mouse finding cheese in a maze. Graph search algorithms provide a way of systematically searching through this maze of possible solutions.

Another example of an optimization problem is finding the shortest path between two nodes in a graph. There may be an exponential number of paths between these two nodes. It would take too much time to consider each such path. The algorithms used to find a shortest one demonstrate many of the principles that will arise when solving harder optimization problems.

A surprisingly large number of problems in computer science can be expressed as graph theory problems. In this chapter, we will first learn a generic search algorithm that finds more and more of the graph by following arbitrary edges from nodes that have already been found. We also consider the more specific orders of depth-first and breadth-first search to traverse the graph.

Using these ideas, we are able to solve the optimization problem of discovering shortest paths between pairs of nodes and to learn about the structure of the graph.

20.1 A Generic Search Algorithm

The Reachability Problem

Preconditions: The input is a graph G (either directed or undirected) and a source node s .

Postconditions: The output consists of all the nodes u that are reachable by a path in G from s .

Basic Steps: Suppose you know that node u is reachable from s (denoted as $s \rightarrow u$) and that there is an edge from u to v . Then you can conclude that v is reachable from s (i.e., $s \rightarrow u \rightarrow v$). You can use such steps to build up a set of reachable nodes.

- s has an edge to v_4 and v_9 . Hence, v_4 and v_9 are reachable.
- v_4 has an edge to v_7 and v_3 . Hence, v_7 and v_3 are reachable.
- v_7 has an edge to v_2 and v_8 . . .

Difficulties

- Data Structure:** How do you keep track of all this?
- Exit Condition:** How do you know that you have found all the nodes?
- Halting:** How do you avoid cycling, as in $s \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow \dots$, forever?

Ingredients of the Loop Invariant

- Found:** If you trace a path from s to a node, then we will say that the node has been *found*.
- Handled:** At some point in time after node u has been found, you will want to follow all the edges from u and find all the nodes v that have edges from u . When you have done that for node u , we say that it has been *handled*.
- Data Structure:** You must maintain (1) the set of nodes *foundHandled* that have been found and handled, and (2) the set of nodes *foundNotHandled* that have been found but not handled. See [Figure 20.1](#).

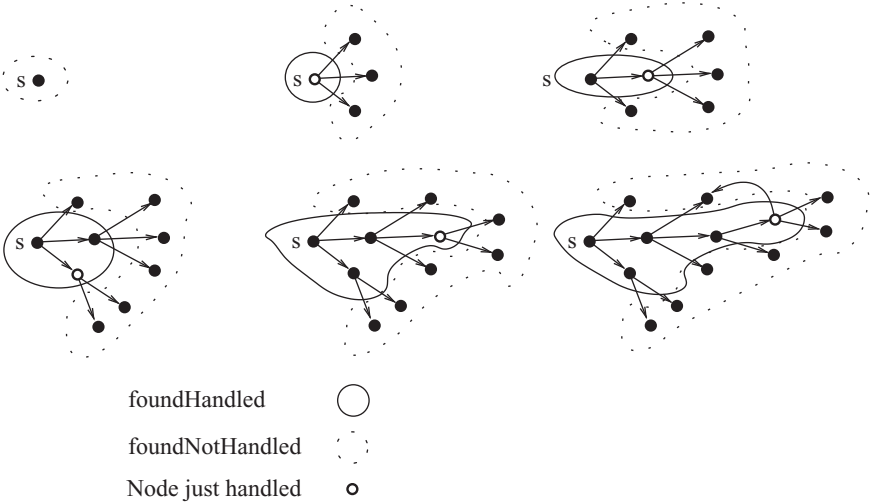


Fig. 20.1 The generic search algorithm *handles* one found node at a time by *finding* its neighbors.

The Loop Invariant

- LI1:** For each found node v , we know that v is reachable from s , because we have traced out a path $s \rightarrow v$ from s to it.
- LI2:** If a node has been handled, then all of its neighbors have been found.

These loop invariants are simple enough that establishing and maintaining them should be easy. But do they suffice to prove the postcondition? We will see.

Body of the Loop: A reasonable step would be:

- Choose some node u from *foundNotHandled*, and handle it. This involves following all the edges from u .
- Newly found nodes are now added to the set *foundNotHandled* (if they have not been found already).
- u is moved from *foundNotHandled* to *foundHandled*.

Code

algorithm *GenericSearch*(G, s)

⟨**pre-cond**⟩: G is a (directed or undirected) graph, and s is one of its nodes.

⟨**post-cond**⟩: The output consists of all the nodes u that are reachable by a path in G from s .

begin

foundHandled = $\emptyset$

foundNotHandled = $\{s\}$

loop

⟨**loop-invariant**⟩: See LI1, LI2.

exit when *foundNotHandled* = $\emptyset$

let u be some node from *foundNotHandled*

for each v connected to u

if v has not previously been found then

add v to *foundNotHandled*

end if

end for

move u from *foundNotHandled* to *foundHandled*

end loop

return *foundHandled*

end algorithm

Maintaining the Loop Invariant (⟨**LI'**⟩ & not ⟨**exit**⟩ & $\text{code}_{\text{loop}} \rightarrow \langle \text{LI}'' \rangle$): Suppose that **LI'** (the statement of the loop invariant before the iteration) is true, the exit condition ⟨**exit**⟩ is not, and we have executed another iteration of the algorithm.

Maintaining LI1: After the iteration, the node v is considered found. Hence, in order to maintain the loop invariant, we must be sure that v is reachable from s . Because u was in *foundNotHandled*, the loop invariant assures us that we have traced out a path $s \rightarrow u$ to it. Now that we have traced the edge $u \rightarrow v$, we have traced a path $s \rightarrow u \rightarrow v$ to v .

Maintaining LI2: Node u is designated handled only after ensuring that all its neighbors have been found.

The Measure of Progress: The measure of progress requires the following three properties:

Progress: We must guarantee that our measure of progress increases by at least one every time around the loop. Otherwise, we may loop forever, making no progress.

Bounded: There must be an upper bound on the progress required before the loop exits. Otherwise, we may loop forever, increasing the measure of progress to infinity.

Conclusion: When sufficient progress has been made to exit, we must be able to conclude that the problem is solved.

An obvious measure would be the number of found nodes. The problem is that when handling a node, you may only find nodes that have already been found. In such a case, no progress is actually made.

A better measure of progress is the number of nodes that have been handled. We can make progress simply by handling a node that has not yet been handled. We also know that if the graph G has only n nodes, then this measure cannot increase past n .

Exit Condition: Given our measure of progress, when are we finished? We can only handle nodes that have been found and not handled. Hence, when all the nodes that have been found have also been handled, we can make no more progress. At this point, we must stop.

Initial Code ($\langle pre-cond \rangle$ & $code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$): Initially, we know only that s is reachable from s . Hence, let's start by saying that s is found but not handled and that all other nodes have not yet been found.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): Our output will be the set of found nodes. The postcondition requires the following two claims to be true.

Claim: Found nodes are reachable from s .

This is clearly stated in the loop invariant.

Claim: Every reachable node has been found. A logically equivalent statement is that every node that has not been found is not reachable.

One Proof: Draw a circle around the nodes of the graph G that have been found. If there are no edges going from the inside of the circle to the outside of the circle, then there are no paths from s to the nodes outside the circle. Hence, we can claim we have found all the nodes reachable from s . How do we know that this circle has no edges leaving it? Consider a node u in the circle. Because v has been found and $foundNotHandled = \emptyset$, we know that u has also been handled. By the loop invariant LI2, if $\langle u, v \rangle$ is an edge, then u has been found and thus is in the circle as well. Hence, if u is in the circle and $\langle u, v \rangle$ is an edge, then v is in the circle as well (i.e., no edges leave the circle).

Closure Property: This is known as a *closure property*. See [Section 24.3.3](#) for more information on this property.

Another Proof: Proof by contradiction.

Suppose that w is reachable from s and that w has not been found. Consider a path from s to w . Because s has been found and w has not, the path starts in the set of found nodes and at some point leaves it. Let $\langle u, v \rangle$ be the first edge in the path for which u but not v has been found. Because u has been found and $\text{foundNotHandled} = \emptyset$, it follows that u has been handled. Because u has been handled, v must be found. This contradicts the definition of v .

Running Time

A Simple but False Argument: For every iteration of the loop, one node is handled, and no node is handled more than once. Hence, the measure of progress (the number of nodes handled) increases by one with every loop. G only has $|V| = n$ nodes. Hence, the algorithm loops at most n times. Thus, the running time is $O(n)$.

This argument is false, because while handling u we must consider v for every edge coming out of u .

Overestimation: Each node has at most n edges coming out of it. Hence, the running time is $O(n^2)$.

Correct Complexity: Each edge of G is looked at exactly twice, once from each direction. The algorithm's time is dominated by this fact. Hence, the running time is $O(|E|)$, where E is the set of edges in G .

The Order of Handling Nodes: This algorithm specifically did not indicate which node u to select from foundNotHandled . It did not need to, because the algorithm works no matter how this choice is made. We will now consider specific orders in which to handle the nodes and specific applications of these orders.

Queue (Breadth-First Search): One option is to handle nodes in the order they are found in. This treats foundNotHandled as a queue: “first in, first out.” The effect is that the search is *breadth first*, meaning that all nodes at distance 1 from s are handled first, then all those at distance 2, and so on. A byproduct of this is that we find for each node v a shortest path from s to v . See [Section 20.2](#).

Priority Queue (Shortest (Weighted) Paths): Another option calculates for each node v in foundNotHandled the minimum weighted distance from s to v along any path seen so far. It then handles the node that is closest to s according to this approximation. Because these approximations change throughout time, foundNotHandled is implemented using a priority queue: “highest current priority out first.” Like breadth-first search, the search handles nodes that are closest to s first, but now the length of a path is the sum of its edge weights. A byproduct

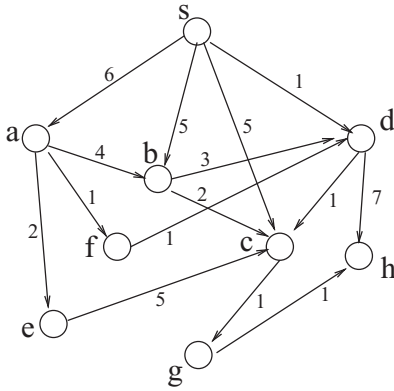


Fig. 20.2 An example graph used in [Exercise 20.1.1](#) to run breadth-first search, shortest weighted paths, and depth-first search.

of this method is that we find for each node v the shortest weighted path from s to v . See [Section 20.3](#).

Stack (Depth-First Search): Another option is to handle the node that was found most recently. This method treats *foundNotHandled* as a stack: “last in, first out.” The effect is that the search is *depth first*, meaning that a particular path is followed as deeply as possible into the graph until a dead end is reached, forcing the algorithm to backtrack. See [Section 20.4](#).

Exercise 20.1.1 Try searching the graph in [Figure 20.2](#) using queue (breadth-first search), priority queue (shortest (weighted) paths), and stack (depth-first search).

20.2 Breadth-First Search for Shortest Paths

We will now develop an algorithm for an optimization problem called the *shortest-path problem*. The algorithm uses a breadth-first search. This algorithm is a less generic version of the algorithm in [Section 20.1](#), because the order in which the nodes are handled is now specified more precisely. The loop invariants are strengthened in order to solve the shortest-path problem.

The Shortest-Path Problem (Multiple Sink): Generally, the shortest-path problem finds a shortest path between a source node s and a sink node t in a graph G . Here, however, we will consider the case where we simultaneously consider all nodes v to be the sink.

Precondition: $\langle G, s \rangle$ consists of a graph G and a source node s . The graph G can be directed or undirected.

Postconditions: The output consists of a $d(v)$ and a $\pi(v)$ for each node of G . It has the following properties:

1. For each node v , $d(v)$ gives the length $\delta(s, v)$ of the shortest path from s to v .

2. The *shortest-path* or *breadth-first search tree* is defined using a function π as follows: s is the root of the tree. $\pi(v)$ is the parent of v in the tree. For each node v , one of the shortest paths from s to v is given backward, with $v, \pi(v), \pi(\pi(v)), \pi(\pi(\pi(v))), \dots, s$. A recursive definition is that this shortest path from s to v is the given shortest path from s to $\pi(v)$, followed by the edge $\langle \pi(v), v \rangle$.

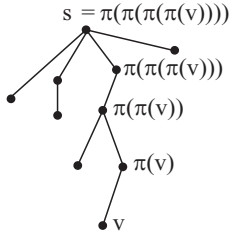


Fig. 20.3 As indicated by the second postcondition of the Shortest-Path Problem, $\pi(v)$ is the parent of v in the tree giving that a shortest path from s to v is given backward, with $v, \pi(v), \pi(\pi(v)), \pi(\pi(\pi(v))), \dots, s$.

An Optimization Problem: The single-source, single-sink version of the shortest-path problem can be viewed as an optimization problem. See [Chapter 19](#).

Instances: An instance $\langle G, s, t \rangle$ consists of a graph and two nodes s and t .

Solutions for Instance: A solution for the instance $\langle G, s, t \rangle$ is a path π from s to t .

Measure of Success: The length (or cost) of a path π is the number of edges in the path.

Goal: Given an instance $\langle G, s, t \rangle$, the goal is to find an optimal solution, i.e., a shortest path from s to t in G .

Brute Force Algorithm: As is often the case with optimization problems, the number of solutions for an instance may well be exponential. We do not want to check them all.

Prove Path Is Shortest: In order to claim that the shortest path from s to v is of some length $d(v)$, you must do two things:

Not Further: You must produce a suitable path of this length. We call this path a *witness* of the fact that the distance from s to v is at most $d(v)$. In finding a node, we trace out a path from s to it. If we have already traced out a shortest path from s to u with $d(u)$ edges in it and we trace an edge from u to v , then we have traced a path from s to v with $d(v) = d(u) + 1$ edges in it. In this path from s to v , the node preceding v is $\pi(v) = u$.

Not Closer: You must prove that there are no shorter paths. This is harder. Other than checking an exponential number of paths, how can you prove that there are no shorter paths? We will do it using the following trick: Suppose we can ensure

that the order in which we find the nodes is according to the length of the shortest path from s to them. Then, when we find v , we know that there isn't a shorter path to it, or else we would have found it already.

Definition of V_j : Let V_j denote the set of nodes at distance j from s .

The Loop Invariant: See Figure 20.4.

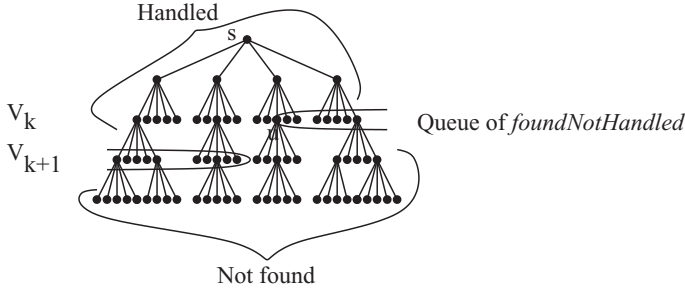


Fig. 20.4 Breadth-first search tree: We cannot assume that the graph is a tree. Here I have presented with only the tree edges given by π . The figure helps to explain the loop invariant, showing which nodes have been found, which found but not handled, and which handled.

LI1: For each found node v , the values of $d(v)$ and $\pi(v)$ are as required, that is, they give the shortest length and a shortest path from s to the node.

LI2: If a node has been handled, then all of its neighbors have been found.

LI3: So far, the order in which the nodes have been found is according to the length of the shortest path from s to it, that is, the nodes in V_j before those in V_{j+1} .

Order in Which to Handle Nodes: The only way in which we are changing the generic search algorithm of Section 20.1 is being more careful in our choice of which node from *foundNotHandled* to handle next. According to LI3, the nodes that were found earlier are closer to s than those that are found later. The closer a node is to s , the closer are its neighbors. Hence, in an attempt to find close nodes, the algorithm will next handle the earliest found node. This is accomplished by treating the set *foundNotHandled* as a queue, “first in, first out.”

Example: See Figure 20.5.

Body of the Loop: Remove the first node u from the *foundNotHandled* queue and handle it as follows. For every neighbor v of u that has not been found:

- add the node to the queue,
- let $d(v) = d(u) + 1$,
- let $\pi(v) = u$, and
- consider u to be handled and v to be in *foundNotHandled*.

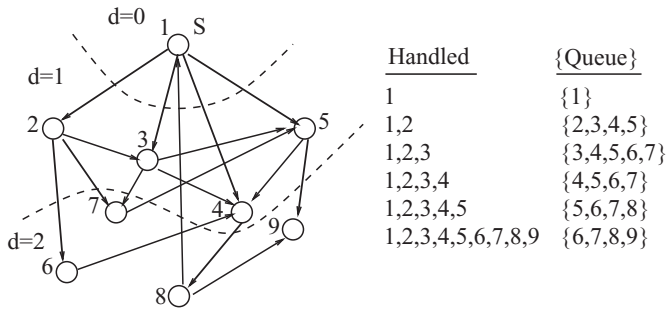


Fig. 20.5 Breadth-first search of a graph. The numbers show the order in which the nodes were found. The contents of the queue are given at each step. The tree edges are darkened.

Code

algorithm *ShortestPath*(G, s)

<pre-cond>: G is a (directed or undirected) graph, and s is one of its nodes.

<post-cond>: π specifies a shortest path from s to each node of G , and d specifies their lengths.

begin

$foundHandled = \emptyset$

$foundNotHandled = \{s\}$

$d(s) = 0, \pi(s) = \epsilon$

 loop

<loop-invariant>: See above.

 exit when $foundNotHandled = \emptyset$

 let u be the node in the front of the queue $foundNotHandled$

 for each v connected to u

 if v has not previously been found then

 add v to $foundNotHandled$

$d(v) = d(u) + 1$

$\pi(v) = u$

 end if

 end for

 move u from $foundNotHandled$ to $foundHandled$

 end loop

 (for unfound $v, d(v) = \infty$)

 return $\langle d, \pi \rangle$

end algorithm

Maintaining the Loop Invariant ($\langle LI' \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI'' \rangle$): Suppose that LI' (the statement of the loop invariant before the iteration) is true, the exit condition $\langle exit \rangle$ is not, and we have executed another iteration of the algorithm.

Closer Nodes Have Already Been Found: We will need the following claim twice.

Claim: If the first node in the queue *foundNotHandled*, that is, u , is in V_k , then

1. all the nodes in $V_0, V_1, V_2, \dots, V_{k-1}$ have already been found and handled, and
2. all the nodes in V_k have already been found.

Proof of Part 1 of Claim: Let u' denote any node in $V_0, V_1, V_2, \dots, V_{k-1}$. Because LI3' ensures that nodes have been found in the order of their distance and because u' is closer to s than u , u' must have been found earlier than u . Hence, u' cannot be in the queue *foundNotHandled*, or else it would be earlier in the queue than u , yet u is first. This proves that u' has been handled.

Proof of Part 2 of Claim: Consider any node v in V_k and any path of length k to it. Let u' be the previous node in this path. Because the subpath to u' is of length $k - 1$, u' is in V_{k-1} , and hence by claim 1 has already been handled. Therefore, by LI2', the neighbors of u' , of which v is one, must have been found.

Maintaining LI1: During this iteration, all the neighbors v of node u that had not been found are now considered found. Hence, their $d(v)$ and $\pi(v)$ must now give the shortest length and a shortest path from s . The code sets $d(v)$ to $d(u) + 1$ and $\pi(v)$ to u . Hence, we must prove that the neighbors v are in V_{k+1} .

Not Further: There is a path from s to v of length $k + 1$: follow the path of length k to u , and then take the edge to v . Hence, the shortest path to v can be no longer than this.

Not Closer: We know that there isn't a shorter path to v , or it would have been found already. More formally, the claim states that all the nodes in $V_0, V_1, V_2, \dots, V_k$ have already been found. Because v has not already been found, it cannot be one of these.

Maintaining LI2: Node u is designated handled only after ensuring that all its neighbors have been found.

Maintaining LI3: By the claim, all the nodes in $V_0, V_1, V_2, \dots, V_k$ have already been found and hence have already been added to the queue. We have also already proved that the node v being found is in V_{k+1} . It follows that the order in which the nodes are found continues to be according to their distance from s .

Initial Code ($\langle pre \rangle \rightarrow \langle LI \rangle$): The initial code puts the source s into *foundNotHandled* and sets $d(s) = 0$ and $\pi(s) = \epsilon$. This is correct, given that initially s has been found but

not handled. The other nodes have not been found, and hence their $d(v)$ and $\pi(v)$ are irrelevant. The loop invariants follow easily.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): The general-search postconditions from [Section 20.1](#) prove that all reachable nodes have been found. LI1 states that for these nodes the values of $d(v)$ and $\pi(v)$ are as required.

For the nodes that are unreachable from s , you can set $d(v) = \infty$ or you can leave them undefined. In some applications (such as the World Wide Web), you have no access to unreachable nodes. An advantage of this algorithm is that it never needs to know about a node unless it has been found.

Exercise 20.2.1 (See solution in Exercise Solutions.) Suppose u is being handled, $u \in V_k$, and v is a neighbor of u . For each of the following cases, explain which $V_{k'}v$ might be in:

- $\langle u, v \rangle$ is an undirected edge, and v has been found before.
- $\langle u, v \rangle$ is an undirected edge, and v has not been found before.
- $\langle u, v \rangle$ is a directed edge, and v has been found before.
- $\langle u, v \rangle$ is a directed edge, and v has not been found before.

Exercise 20.2.2 (See solution in Exercise Solutions.) Estimate the time required to find the shortest path between two given nodes s and t .

20.3 Dijkstra's Shortest-Weighted-Path Algorithm

We will now make the shortest-path problem more general by allowing each edge to have a different weight (length). The length of a path from s to v will be the sum of the weights on the edges of the path. This makes the problem harder, because the shortest path to a node may wind deep into the graph along many short edges instead of along a few long edges. Despite this, only small changes need to be made to the algorithm. The new algorithm is called *Dijkstra's algorithm*.

Specifications of the Shortest-Weighted-Path Problem

Preconditions: The input is a graph G (either directed or undirected) and a source node s . Each edge $\langle u, v \rangle$ is allocated a nonnegative weight $w_{\langle u, v \rangle}$.

Postconditions: The output consists of d and π , where for each node v of G , $d(v)$ gives the length $\delta(s, v)$ of the shortest weighted path from s to v , and π defines a *shortest-weighted-path tree*. (See [Section 20.2](#).)

Prove Path Is Shortest: As before, proving that the shortest path from s to v is of some length $d(v)$ involves producing a suitable path of this length and proving that there are no shorter paths.

Not Further: As before, a witness that there is such a path is produced by tracing it out. The only change is that when we find a path $s \rightarrow u \rightarrow v$, we compute its length to be $d(v) = d(u) + w_{\langle u, v \rangle}$ instead of only $d(v) = d(u) + 1$.

Not Closer: Unlike the breadth-first search shortest-path algorithm from [Section 20.2](#), the algorithm does not find the nodes in the order of length of the shortest path to them from s . It does, however, *handle* the nodes in this order. See [Figure 20.6](#). Because of this, when we handle a node, we know that there is no shorter path to it, because otherwise we would have handled it already.

The Next Node to Handle: The algorithm must choose which of the unhandled nodes to handle next. The difficulty is that initially we do not know the length of the shortest path. Instead, we choose the node closest to s according to our current approximation. In [Figure 20.6](#), after handling s our best approximation of the distance to v is 100 and to t is only 1. Hence, we handle t next.

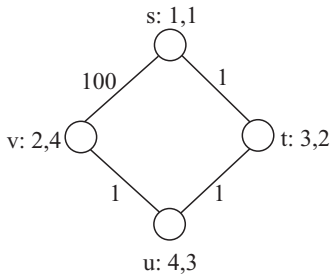


Fig. 20.6 The shortest-path algorithm handles the nodes in the order of length of the shortest path to them.

An Adaptive Greedy Criterion: This choice amounts to an *adaptive greedy criterion*. See [Chapter 22](#) for more on greedy algorithms.

Growing a Tree One Node at a Time: It turns out that the next node to be handled will always be only one edge from a previously handled node. Hence, the tree handled nodes expands out, one node at a time.

Approximation of Shortest Distances: For every node v , before getting its shortest overall distance, we will maintain $d(v)$ and $\pi(v)$ as the shortest length and path from s to v from among those paths that we have *handled* so far.

Updating: This information is continuously updated as we find shorter paths to v . For example, if we find v when handling u , then we update these values as follows:

```

foundPathLength =  $d(u) + w_{\langle u, v \rangle}$ 
if  $d(v) > \text{foundPathLength}$  then
     $d(v) = \text{foundPathLength}$ 
     $\pi(v) = u$ 
end if
  
```

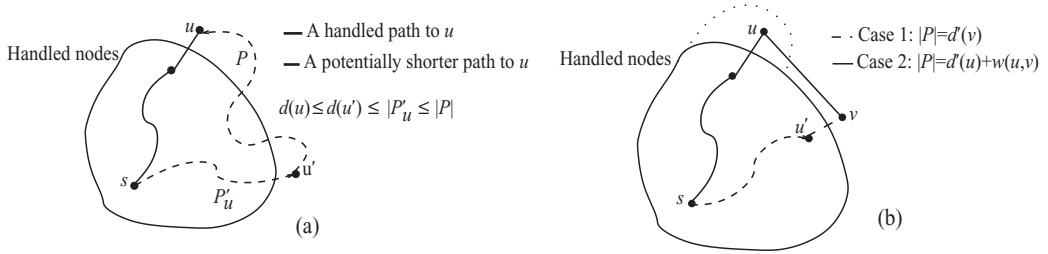


Fig. 20.7 (a) shows a handled path to node u and is used to maintain LI1. (b) is used to maintain LI2.

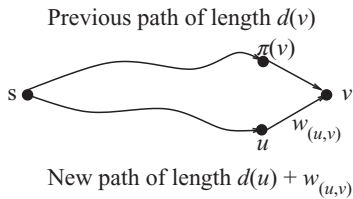


Fig. 20.8 Update the length of the shortest path from s to v , when handling u .

When handling s in our example, $d(v)$ is set to $d(s) + w_{(s,v)} = 0 + 100 = 100$. Later, when handling u , it is updated to $d(u) + w_{(u,v)} = 2 + 1 = 3$.

Definition of a Handled Path: We say that a path has been handled if it contains only handled edges. Such paths start at s , visit as any number of handled nodes, and then follow one last edge to a node that may or may not be handled. (See the solid path to u in [Figure 20.7.a.](#))

Priority Queue: The next node to be handled is the one with the smallest $d(u)$ value. Searching the set of unhandled nodes for this node during each iteration would be too time-consuming. Re-sorting the nodes each iteration as the $d(u)$ values change would also be too time-consuming. A more efficient implementation uses a priority queue to hold the unhandled nodes prioritized according to their current $d(u)$ value. This can be implemented using a heap. (See [Section 13.3.](#)) We will denote this priority queue by *notHandled*.

Consider All Nodes “Found”: No path has yet been handled to any node that has not yet been found, and hence $d(v) = \infty$. If we add these nodes to the queue, they will be selected last. Therefore, there is no harm in adding them. Hence, we will distinguish only between those nodes that have been handled and those that have not.

The Loop Invariant

LI1: For each handled node v , the values of $d(v)$ and $\pi(v)$ give the shortest length and a shortest path from s (and this path contains only handled nodes).

LI2: For each of the unhandled nodes v , the values of $d(v)$ and $\pi(v)$ give the shortest length and path from among those paths that have been handled.

Body of the Loop: Take the next node u from the priority queue *notHandled*, and handle it. This involves handling all edges $\langle u, v \rangle$ out of u . Handling the edge $\langle u, v \rangle$ involves updating the $d(v)$ and $\pi(v)$ values. The priorities of these nodes are changed in the priority queue as necessary.

Example: See [Figure 20.9](#).

Code

algorithm *DijkstraShortestWeightedPath*(G, s)

⟨pre-cond⟩: G is a weighted (directed or undirected) graph, and s is one of its nodes.

⟨post-cond⟩: π specifies a shortest weighted path from s to each node of G , and d specifies their lengths.

begin

$d(s) = 0, \pi(s) = \epsilon$

for other $v, d(v) = \infty$ and $\pi(v) = \text{nil}$

$\text{handled} = \emptyset$

$\text{notHandled} =$ priority queue containing all nodes. Priorities given by $d(v)$.

loop

⟨loop-invariant⟩: See above.

exit when $\text{notHandled} = \emptyset$

let u be a node from notHandled with smallest $d(u)$

for each v connected to u

$\text{foundPathLength} = d(u) + w_{\langle u, v \rangle}$

if $d(v) > \text{foundPathLength}$ then

$d(v) = \text{foundPathLength}$

$\pi(v) = u$

(update the notHandled priority queue)

end if

end for

move u from notHandled to handled

end loop

return $\langle d, \pi \rangle$

end algorithm

Maintaining LI1 (⟨LI1', LI2'⟩ & not ⟨exit⟩ & $\text{code}_{\text{loop}} \rightarrow \langle \text{LI1}'' \rangle$): The loop handles a node u with smallest $d(u)$ from notHandled . Hence to maintain LI1, we must ensure that its $d(u)$ and $\pi(u)$ values give an overall shortest path to u . Consider some other path P to u . We will see that it is no shorter. See [Figure 20.7\(a\)](#) Because the path P starts at the handled node s and ends at the previously unhandled node u , there has to be some node u' that is the first previously unhandled node along P . (It is possible that

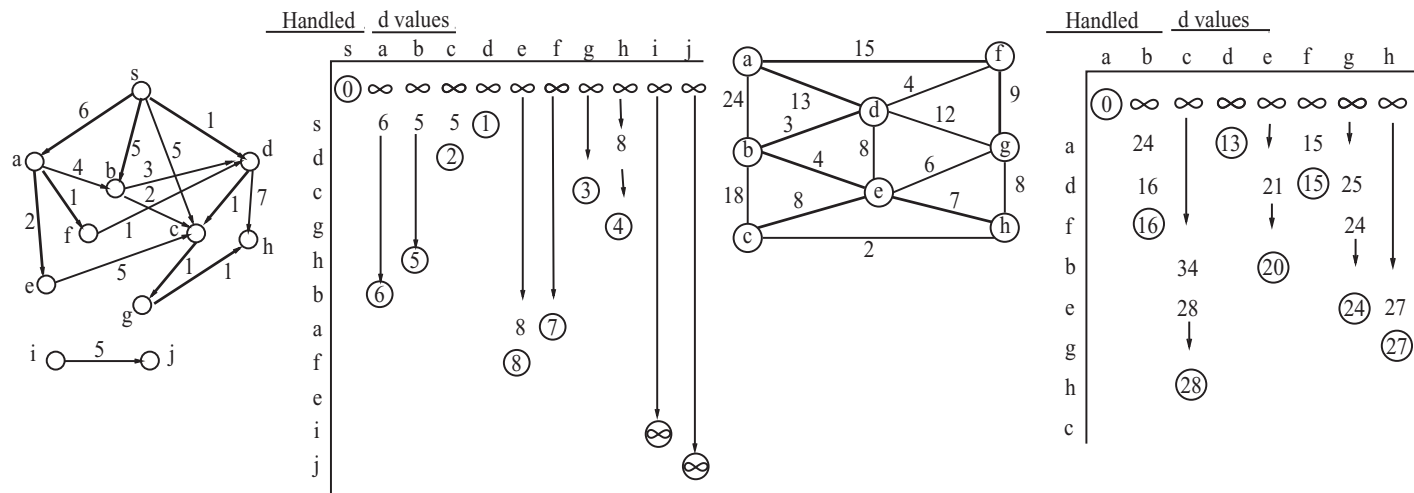


Fig. 20.9 Dijkstra's algorithm. The d value at each step is given for each node. The tree edges are darkened.

$u' = u$.) By the choice of u , u has the smallest $d(u)$ from *notHandled*. Hence, $d(u) \leq d(u')$. Let $P_{u'}$ be the part the path that goes from s to u' . This is a previously handled path. Hence, by LI2', $d(u') \leq |P_{u'}|$. Since $|P_{u'}|$ is a subpath and there are no negative weights, $|P_{u'}| \leq |P|$. Combining these gives $d(u) \leq |P|$. In conclusion, $d(u)$ is the length of the shortest path to u , and hence LI1 has been maintained.

Maintaining LI2 ($\langle LI1', LI2' \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI2'' \rangle$): Setting $d''(v)$ to $\min\{d'(v), d'(u) + w_{\langle u, v \rangle}\}$ ensures that there is a handled path with this length to v . To maintain LI2, we must prove that there does not exist a shorter one from among those paths that now are considered handled. Such paths can now include the newly handled node u . Let P be a shortest one. See Figure 20.7.b. Let u' be the second last node in P . Because P is a handled path, u' must be a handled node. There are two cases:

$u \neq u'$: If u' is a previously handled node, then by the second part of LI1', the shortest path to it does not need to contain the newly handled node u . It follows that this path P to v is a previously handled path. Hence, its length is at least the length of the shortest previously handled path to v , which by LI2', is $d'(v)$. This in turn is at least $\min\{d'(v), d'(u) + w_{\langle u, v \rangle}\} = d''(v)$.

$u = u'$: If the second last node in P is the newly handled node u , then its length is the length of the shortest path to u , which we now know is $d'(u)$, plus the weight of the edge $\langle u, v \rangle$. It follows that $|P| \geq \min\{d'(v), d'(u) + w_{\langle u, v \rangle}\} = d''(v)$.

Either way, the shortest path P to v that is now considered to be handled has length at least $d''(v)$. Hence, LI2 is maintained.

Initial Code ($\langle pre \rangle \rightarrow \langle LI \rangle$): The initial code is the same as that for the breadth-first search shortest-path algorithm from Section 20.2, that is, s is found but not handled with $d(s) = 0$, $\pi(s) = \epsilon$. Initially no paths to v have been handled, and hence the length of the shortest handled path to v is $d(v) = \infty$. This satisfies all three loop invariants.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): See the shortest-path algorithm.

Exercise 20.3.1 Estimate the running time of Dijkstra's shortest-weighted-path algorithm.

Exercise 20.3.2 (See solution in Exercise Solutions.) Given a graph where each edge weight is one, compare and contrast the computation of the breadth-first search shortest-path algorithm from Section 20.2 and that of Dijkstra's shortest-weighted-path algorithm. How do their choices of the next node to handle and their loop invariants compare?

Exercise 20.3.3 Dijkstra's algorithm:

1. Give the full loop invariant for Dijkstra's algorithm. Include the definition of any terms you use.
2. What is the exit condition for Dijkstra's algorithm?
3. Prove that the postcondition is obtained.

4. Consider a computation of Dijkstra's algorithm on Figure 20.10 when the circled nodes have been handled. The start node is a . On the left, give the current values of d .

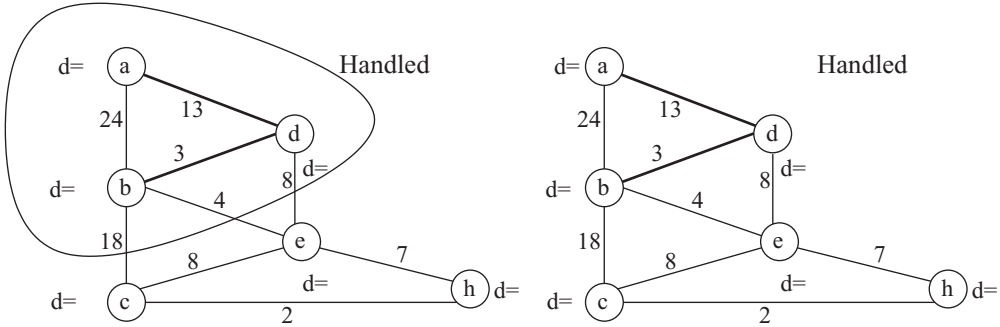


Fig. 20.10 Dijkstra's algorithm after the nodes circled have already been handled.

5. On the right, change the figure to take one step in Dijkstra's algorithm. Include as well any π 's that change.

Exercise 20.3.4 Give a simple graph with an edge with a negative weight, and show that Dijkstra's algorithm gives the wrong answer.

20.4 Depth-First Search

We have considered breadth-first search, which first visits nodes at distance 1 from s , then those at distance 2, and so on. We will now consider a *depth-first search*, which continues to follow some path as deeply as possible into the graph before it is forced to backtrack. I give an iterative algorithm in this section and a recursive one in Section 20.5.

Changes to the Generic Search Algorithm in Section 20.1: The next node u we handle is the one most recently found. The stack *foundNotHandled* will be implemented as a stack of tuples $\langle v, i_v \rangle$. At each iteration, we pop the most recently pushed tuple $\langle u, i_u \rangle$ and handle the $(i_u + 1)$ st edge from u . Try this out on a graph (or on a tree). The pattern in which nodes are found consists of a single path with single edges hanging off it. See Figure 20.11.

In order to prevent the single edges hanging off the path from being searched, we make a second change to the original searching algorithm: We no longer completely handle one node before we start handling edges from other nodes. From s , an edge is followed to one of its neighbors v_1 . Before visiting the other neighbors of s , the current path to v_1 is extended to $v_2, v_3, \dots$ (See Figure 20.11.b.) We keep track of what has been handled by storing an integer i_u for each node u . We maintain that for each u , the first i_u edges of u have already been handled.

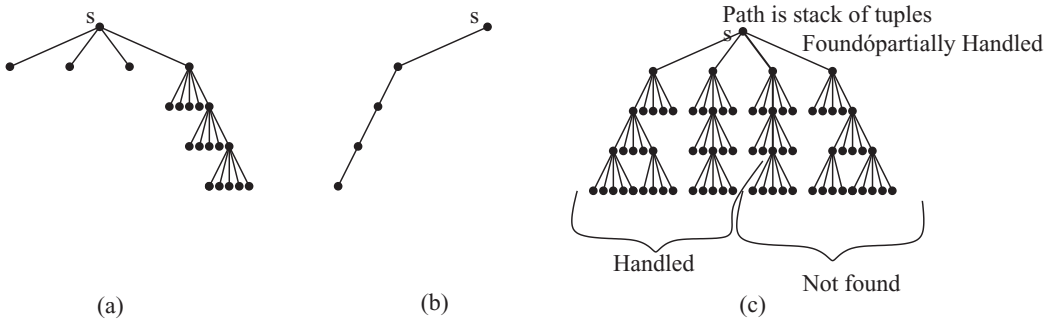


Fig. 20.11 If the next node in the stack was completely handled, then the initial order in which nodes are found is given in (a). If the next node is only partially handled, then this initial order is given in (b). And (c) presents more of the order in which the nodes are found. Though the input graph may not be a tree, the figure only shows the tree edges given by π .

Loop Invariants

LI1: The nodes in the stack *foundNotHandled* are ordered so that they define a path starting at s .

LI2: *foundNotHandled* is a stack of tuples $\langle v, i_v \rangle$ such that for each v , the first i_v edges of v have been handled. Each node v appears no more than once.

Code

algorithm *DepthFirstSearch*(G, s)

<pre-cond>: G is a (directed or undirected), graph, and s is one of its nodes.

<post-cond>: The output is a depth-first search tree of G rooted at s .

begin

foundHandled = $\emptyset$

foundNotHandled = $\{\langle s, 0 \rangle\}$

time = 0 % Used for *time stamping*. See following discussion.

loop

<loop-invariant>: See preceding list.

exit when *foundNotHandled* = $\emptyset$

pop $\langle u, i \rangle$ off the stack *foundNotHandled*

if u has an $(i + 1)$ st edge $\langle u, v \rangle$

push $\langle u, i + 1 \rangle$ onto *foundNotHandled*

if v has not previously been found then

$\pi(v) = u$

$\langle u, v \rangle$ is a tree edge

push $\langle v, 0 \rangle$ onto *foundNotHandled*

```

     $s(v) = \text{time}; \text{time} = \text{time} + 1$ 
  else if  $v$  has been found but not completely handled then
     $\langle u, v \rangle$  is a back edge
  else ( $v$  has been completely handled)
     $\langle u, v \rangle$  is a forward or cross edge
  end if
else
  move  $u$  to foundHandled
   $f(v) = \text{time}; \text{time} = \text{time} + 1$ 
end if
end loop
return foundHandled
end algorithm
```

Example: See Figure 20.12.

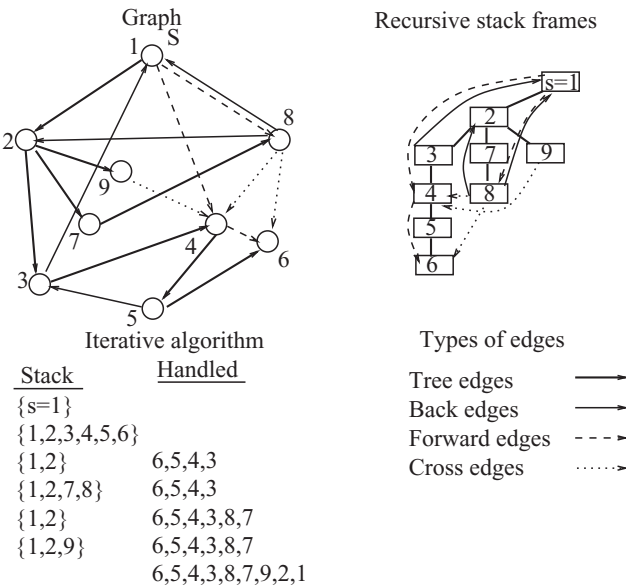


Fig. 20.12 Depth-first search of a graph. The numbers give the order in which the nodes are found. The contents of the stack are given at each step.

Establishing and Maintaining the Loop Invariant: It is easy to see that with $\text{foundNotHandled} = \{ \langle s, 0 \rangle \}$, the loop invariant is established. If the stack does contain a path from s to u and u has an unhandled edge to v , then u is kept on the stack and v is pushed on top. This extends the path from u onward to v . If u does not have an unhandled edge, then u is popped off the stack. This decreases the path from s by one.

Classification of Edges: The depth-first search algorithm can be used to classify edges:

Tree Edges: Tree edges are the edges $\langle u, v \rangle$ in the depth-first search tree. When such edges are handled, v has not yet been found.

Back Edges: Back edges are the edges $\langle u, v \rangle$ such that v is an ancestor of u in the depth-first search tree. When such edges are handled, v is in the stack, that is, found but not completely handled.

Cyclic: A graph is cyclic if and only if it has a back edge.

Proof ($\Leftarrow$): The loop invariant of the depth-first search algorithm ensures that the contents of the stack form a path from s through v and onward to u . Adding on the edge $\langle u, v \rangle$ creates a cycle back to v .

Proof ($\Rightarrow$): Later we prove that if the graph has no back edges, then there is a total ordering of the nodes respecting the edges and hence the graph has no cycles.

Bipartite: A graph is bipartite if and only if there is no back edge between any two nodes with the same level parity, that is, iff it has no odd-length cycles.

Forward Edges and Cross Edges: Forward edges are edges $\langle u, v \rangle$ such that v is a descendant of u in the depth-first search tree.

Cross edges $\langle u, v \rangle$ are such that u and v are in different branches of the depth-first search tree (that is, are neither ancestors nor descendants of each other) and v 's branch is traversed before (to the left of) u 's branch.

When forward edges and cross edges are handled, v has been completely handled. The depth-first search algorithm does not distinguish between forward edges and cross edges.

Time Stamping: Some implementations of depth-first search time-stamp each node u with a start time $s(u)$ and a finish time $f(u)$. Here *time* is measured by starting a counter at zero and incrementing it every time a node is found for the first time or a node is completely handled. $s(u)$ is the time at which node u is first found, and $f(u)$ is the time at which it is completely handled. The time stamps are useful in the following way:

- v is a descendant of u if and only if the time interval $[s(v), f(v)]$ is completely contained in $[s(u), f(u)]$.
- If u and v are neither ancestor or descendant of each other, then the time intervals $[s(u), f(u)]$ and $[s(v), f(v)]$ are completely disjoint.

Using the time stamps, this can be determined in constant time.

Exercise 20.4.1 Prove that when doing depth-first search on undirected graphs there are never any forward or cross edges.

20.5 Recursive Depth-First Search

I now present a recursive implementation of a depth-first search algorithm, which directly mirrors the iterative version. The only difference is that the iterative version uses a stack to keep track of the route back to the start node, while the recursive version uses the stack of recursive stack frames. The advantage of the recursive algorithm is that it is easier to code and easier to understand. The iterative algorithm might run slightly faster, but a good compiler will convert the recursive algorithm into an iterative one.

Code

algorithm *DepthFirstSearch*(*s*)

⟨pre-cond⟩: An input instance consists of a (directed or undirected) graph *G* with some of its nodes marked *found* and a source node *s*.

⟨post-cond⟩: The output is the same graph *G*, except all nodes *v* reachable from *s* without passing through a previously found node are now also marked as being found. The graph *G* is a global variable ADT, which is assumed to be both input and output to the routine.

```
begin
    if s is marked as found then
        do nothing
    else
        mark s as found
        for each v connected to s
            DepthFirstSearch(v)
        end for
    end if
end algorithm
```

Pruning Paths: Consider the instance graph in [Figure 20.13](#). There are two obvious paths from node *S* to node *v*. However, there are actually an infinite number of such paths. One path of interest is the one that starts at *S*, and traverses around past *u* up to *c* and then down to *v*. All of these equally valued paths will be pruned from consideration, except the one that goes from *S* through *b* and *u* directly to *v*.

Three Friends: Given this instance, we first mark our source node *S* with an *x*, and then we recurse three times, once from each of *a*, *b*, and *c*.

Friend a: Our first friend marks all nodes that are reachable from its source node *a* = *s* without passing through a previously marked node. This includes only the nodes in the leftmost branch, because when we marked our source *S*, we blocked his route to the rest of the graph.

Friend b: Our second friend does the same. They find, for example, the path that goes from *b* through *u* directly to *v*. They also find and mark the nodes back around to *c*.

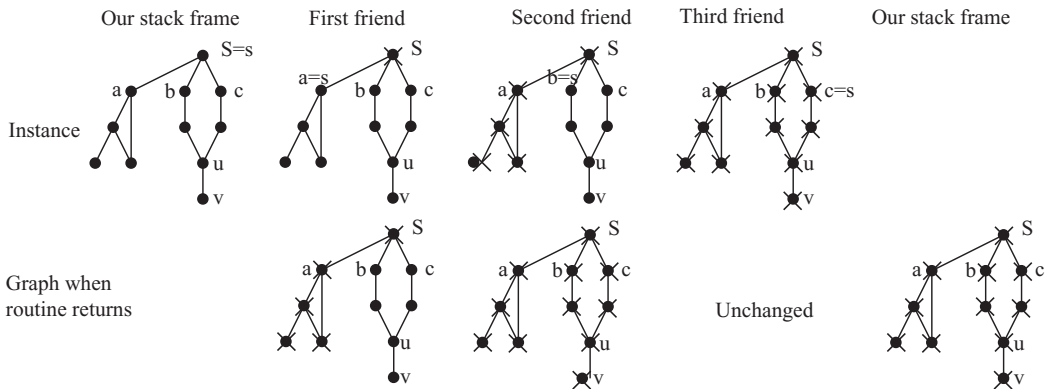


Fig. 20.13 An example instance graph.

Friend *c*: Our third friend is of particular interest. They find that his source node, *c*, has already been marked. Hence, they return without doing anything. This prunes off this entire branch of the recursion tree. The reason that they can do this is that for any path to a node that they would consider, another path to the same node has already been considered.

Achieving the Postcondition: Consider the component of the graph reachable from our source *s* without passing through a previously marked node. (Because our instance has no marked nodes, this includes all the nodes.) To mark the nodes within this component, we do the following. First, we mark our source *s*. This partitions our component of reachable nodes into subcomponents that are still reachable from each other. Each such subcomponent has at least one edge from *s* into it. When we traverse the first such edge, the friend marks all the nodes within this subcomponent.

Running Time: Marking a node before it is recursed from ensures that each node is recursed from at most once. Recursing from a node involves traversing each edge from it. Hence, each edge is traversed at most twice: once from each direction. Hence, the running time is linear in the number of edges.

Exercise 20.5.1 Trace out the iterative and the recursive algorithm on the same graph, and see how they compare. Do they have the same running time?

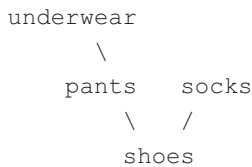
20.6 Linear Ordering of a Partial Order

Finding a linear order consistent with a given partial order is one of many applications of a depth-first search. (Hint: If a question ever mentions that a graph is directed acyclic, always start by running this algorithm.)

Definition of Total Order: A *total order* of a set of objects *V* specifies for each pair of objects *u, v* $\in V$ either (1) that *u* is before *v*, or (2) that *v* is before *u*. It must be *transitive*, in that if *u* is before *v* and *v* is before *w*, then *u* is before *w*.

Definition of Partial Order: A *partial order* of a set of objects V supplies only some of the information of a total order. For each pair of objects $u, v \in V$, it specifies either that u is before v , that v is before u , or that the order of u and v is undefined. It must also be transitive.

For example, you must put on your underwear before your pants, and you must put on your shoes after both your pants and your socks. According to transitivity, this means you must put your underwear on before your shoes. However, you do have the freedom to put your underwear and your socks on in either order. My son, Josh, when six, mistook this partial order for a total order and refused to put on his socks before his underwear. When he was eight, he explained to me that the reason that he could get dressed faster than I was that he had a “shortcut,” consisting in putting his socks on before his pants. I was thrilled that he had at least partially understood the idea of a partial order:



A partial order can be represented by a directed acyclic graph (DAG) G . The vertices consist of the objects V , and the directed edge $\langle u, v \rangle$ indicates that u is before v . It follows from transitivity that if there is a directed path in G from u to v , then we know that u is before v . A cycle in G from u to v and back to u presents a contradiction, because u cannot be both before and after v .

Specifications of the Topological Sort Problem

Preconditions: The input is a directed acyclic graph G representing a partial order.

Postconditions: The output is a total order consistent with the partial order given by G , that is, for all edges $\langle u, v \rangle \in G$, u appears before v in the total order.

An Easy but Slow Algorithm

The Algorithm: Start at any node v of G . If v has an outgoing edge, walk along it to one of its neighbors. Continue walking until you find a node t that has no outgoing edges. Such a node is called a *sink*. This process cannot continue forever, because the graph has no cycles.

The sink t can go after every node in G . Hence, you should put t last in the total order, delete t from G , and recursively repeat the process on $G - v$.

Running Time: It takes up to n time to find the first sink, $n - 1$ to find the second, and so on. The total time is $\Theta(n^2)$.

Algorithm Using a Depth-First Search: Start at any node s of G . Do a depth-first search starting at node s . After this search completes, nodes that are considered found will continue to be considered found, and so should not be considered again. Let s' be

any unfound node of G . Do a depth-first search starting at node s' . Repeat the process until all nodes have been found.

Use the time stamp $f(u)$ to keep track of the order in which nodes are *completely handled*, that is, removed from the stack. Output the nodes in reverse order.

If you ever find a back edge, then stop and report that the graph has a cycle.

Proof of Correctness

Lemma: For every edge $\langle u, v \rangle$ of G , node v is completely handled before u .

Proof of Lemma: Consider some edge $\langle u, v \rangle$ of G . Before u is completely handled, it must be put onto the stack *foundNotHandled*. At this point in time, there are three cases:

Tree Edge: v has not yet been found. Because u has an edge to v , v is put onto the top of the stack above u before u has been completely handled. No more progress will be made toward handling u until v has been completely handled and removed from the stack.

Back Edge: v has been found, but not completely handled, and hence is on the stack somewhere below u . Such an edge is a back edge. This contradicts the fact that G is acyclic.

Forward or Cross Edge: v has already been completely handled and removed from the stack. In this case, we are done: v was completely handled before u .

Topologically Sorted: Exercise 20.6.1 asks to show that this lemma is sufficient to prove that the reverse order in which the nodes were completely handled is a correct topological sort.

Example: See the example instance in Figure 20.14.

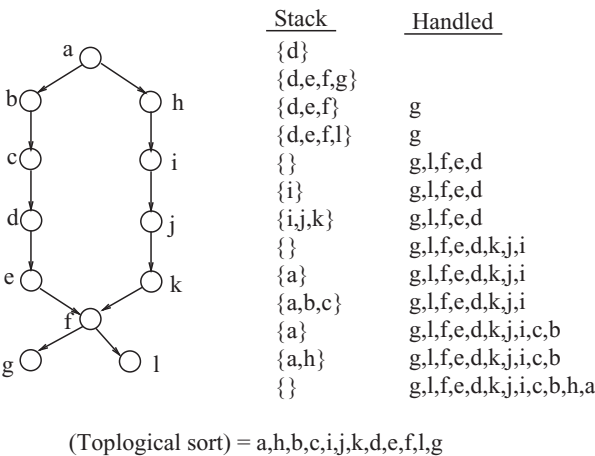


Fig. 20.14 A topological sort is found using a depth-first search.

Running Time: As with the depth-first search, no edge is followed more than once. Hence, the total time is $\Theta(|E|)$.

Shortest-Weighted Path on a DAG: Suppose you want to find the shortest-weighted path for a directed graph G that you know is acyclic. You could use Dijkstra's algorithm from [Section 20.3](#). However, as hinted above, whenever a question mentions that a graph is acyclic, it is always fastest to start by finding a linear order consistent with the edges of the graph. Once this has been completed, you can handle the nodes (as done in Dijkstra's algorithm) in this linear order. [Exercise 20.6.2](#) asks you to prove the correctness of this algorithm.

Exercise 20.6.1 Show that in order to prove that the reverse order in which the nodes were completely handled is a correct topological sort, it is sufficient to prove that for every edge $\langle u, v \rangle$ of G , node v is completely handled before u .

Exercise 20.6.2 (See solution in Exercise Solutions.) Prove the correctness and estimate the running time of this algorithm for shortest weighted paths for DAGs.

20.7 Exercise

Exercise 20.7.1 Trace breadth-first and depth-first searches on [Figure 20.15](#). For each, do the following:

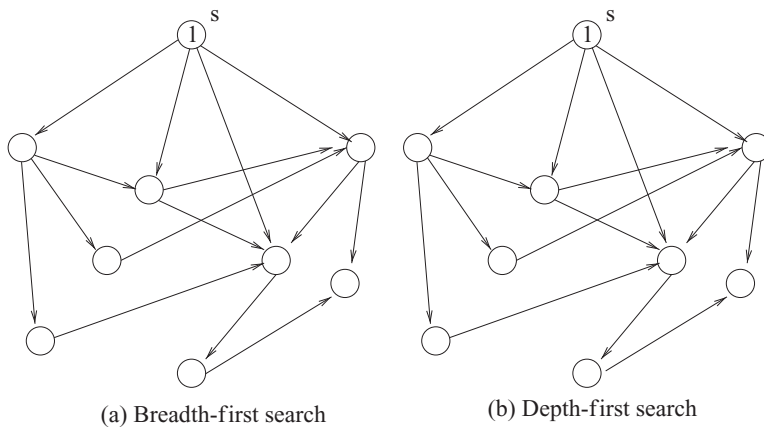


Fig. 20.15 Example graph for [Exercise 20.7.1](#).

1. Start at node s , and when there is a choice, follow edges from left to right. Number the nodes $1, 2, 3, \dots$ in the order that they are found, starting with node $s = 1$.
2. Darken the edges of the tree specified by the predecessor array π .
3. What is the data structure used by each search to store nodes that are found but not yet handled?
4. Circle the nodes that are in this data structure when node 8 is first found.

21 Network Flows and Linear Programming

Network flow is a classic computational problem with a surprisingly large number of applications, such as routing trucks and matching happy couples. Think of a given directed graph as a network of pipes starting at a source node s and ending at a sink node t . Through each pipe water can flow in one direction at some rate up to some maximum capacity. The goal is to find the maximum total rate at which water can flow from the source node s to the sink node t . If this were a physical system of pipes, you could determine the answer simply by pushing as much water through as you could. However, achieving this algorithmically is more difficult than you might at first think, because the exponentially many paths from s to t overlap, winding forward and backward in complicated ways.

An Optimization Problem: Network flow is another example of an optimization problem, which involves searching for a best solution from some large set of solutions. The formal specifications are described in [Chapter 19](#).

Network Flow Specification: Given an instance $\langle G, s, t \rangle$, the goal is to find a maximum rate of flow through graph G from node s to node t .

Precondition: We are given one of the following instances.

Instances: An *instance* $\langle G, s, t \rangle$ consists of a directed graph G and specific nodes s and t . Each edge $\langle u, v \rangle$ is associated with a positive capacity $c_{\langle u, v \rangle}$. For example, see [Figure 21.1\(a\)](#).

Postcondition: The output is a solution with maximum value and the value of that solution.

Solutions for Instance: A solution for the instance is a *flow* F , which specifies the flow $F_{\langle u, v \rangle}$ through each edge of the graph. The requirements of a flow are as follows. For example, see [Figure 21.1\(b\)](#).

Unidirectional Flow: For any pair of nodes, it is easiest to assume that flow does not go in both directions between them. Hence, we will require that at least one of $F_{\langle u, v \rangle}$ and $F_{\langle v, u \rangle}$ be zero and that neither be negative.

Edge Capacity: The flow through any edge cannot exceed the capacity of the edge, namely, $F_{\langle u, v \rangle} \leq c_{\langle u, v \rangle}$.

No Leaks: No water can be added at any node other than the source s , and no water can be drained at any node other than the sink t .

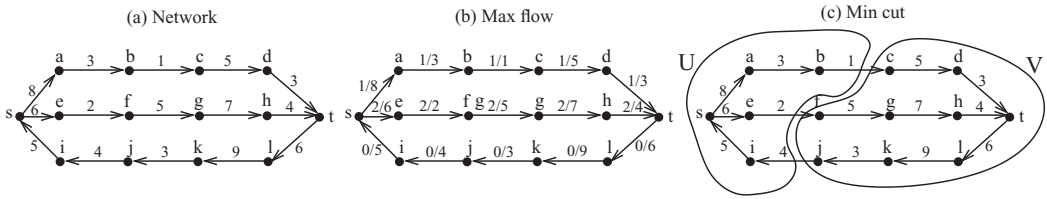


Fig. 21.1 (a) A network with its edge capacities labeled. (b) A maximum flow in this network. The first value associated with each edge is its flow, and the second is its capacity. The total rate of the flow is $3 = 1 + 2 - 0$. Note that no more flow can be pushed along the top path, because the edge $\langle b, c \rangle$ is at capacity. Similarly for the edge $\langle e, f \rangle$. Note also that no flow is pushed along the bottom path, because this would decrease the total from s to t . (c) A minimum cut in this network. The capacity of this min cut is $3 = 1 + 2$. Note that the capacity of the edge $\langle j, i \rangle$ is not included in the capacity of the cut, because it is going in the wrong direction. (b) vs. (c): The rate of the maximum flow is the same as the capacity of the min cut. The edges crossing forward across the cut are at capacity in the flow, while those crossing backward have zero flow. These things are not coincidences.

At each other node the total flow into the node equals the total flow out, i.e., for all nodes $u \notin \{s, t\}$, $\sum_v F_{\langle v, u \rangle} = \sum_v F_{\langle u, v \rangle}$.

Measure of Success: The value of a flow F , denoted $rate(F)$, is the total rate of flow from the source s to the sink t . We will define this to be the total that leaves s without coming back, $rate(F) = \sum_v [F_{\langle s, v \rangle} - F_{\langle v, s \rangle}]$. Agreeing with our intuition, we will later prove that because no flow leaks or is created in between s and t , this flow equals that flowing into t without leaving it, namely $\sum_v [F_{\langle v, t \rangle} - F_{\langle t, v \rangle}]$.

Min Cut Specification: Another interesting and perhaps surprisingly related optimization problem is *min cut*. Given an instance $\langle G, s, t \rangle$, the goal is to find a *cut* between s and t that has the possible minimum capacity crossing it from the s side to the t side.

Precondition: We are given one of the following instances.

Instances: An *instance* $\langle G, s, t \rangle$ consists of a directed graph G and specific nodes s and t . Each edge $\langle u, v \rangle$ is associated with a positive capacity $c_{\langle u, v \rangle}$. Note that the network flow and min cut problems have the same instances. For example, see Figure 21.1(a).

Postcondition: The output is a solution with minimum value and the value of that solution.

Solutions for the Instance: A solution for the instance is a *cut* $C = \langle U, V \rangle$, which is a partitioning of the nodes of the graph into two sets U and V such that the source s is in U and the sink t is in V . For example, see Figure 21.1(c).

For example, if G gives the roads, s is Toronto, and t is Berkeley, a cut could be the Canadian–US border. Because the nodes in a graph do not have a location as cities do, there is no reason for the partition of the

nodes to be geographically contiguous. Anyone of the exponential number of partitions will do.

Measure of Success: The capacity of a cut is the sum of the capacities of all edges from U to V , namely, $\text{cap}(C) = \sum_{u \in U} \sum_{v \in V} c_{\langle u, v \rangle}$. Note that this does not include the capacities $c_{\langle v, u \rangle}$ of the edges going back from V to U .

In [Section 21.1](#), we will design an algorithm for the network flow problem. We will see that this algorithm is an example of a *hill-climbing algorithm* and that it does not necessarily work, because it may get stuck in a small local maximum. In [Section 21.2](#), we will modify the algorithm and use the *primal–dual* method, which uses a min cut to guarantee that it has found a global maximum. This algorithm, however, may have exponential running time. In [Section 21.3](#), we prove that the steepest-ascent version of this hill climbing algorithm runs in polynomial time. Finally, [Section 21.4](#) relates these ideas to another, more general problem called *linear programming*.

Exercise 21.0.1 Suppose we ensured that flow goes in only one direction between any two nodes, not by requiring that the flow in one direction $F_{\langle v, u \rangle}$ be zero, but by requiring that $F_{\langle v, u \rangle} = -F_{\langle u, v \rangle}$. This is less consistent with intuition and obscures some subtleties. The change does, however have the advantage of simplifying many of the equations. For example, the no-leak requirement simplifies to $\sum_v F_{\langle u, v \rangle} = 0$. How does this change all of the other equations in this section?

21.1 A Hill-Climbing Algorithm with a Small Local Maximum

Hiking at the age of seven, my son Josh stated that the way to find the top of the hill is simply to keep walking in a direction that takes you up, and you know you are there when you cannot go up any more. Little did he know that this is also a common technique for finding the best solution for many optimization problems. The algorithm maintains one solution for the problem and repeatedly makes one of a small set of prescribed changes to this solution in a way that makes it a better solution. It stops when none of these changes seems able to make a better solution.

There are two problems with this technique. First, it is not necessarily clear how long it will take until the algorithm stops. Second, sometimes it finds a small local maximum, i.e., the top of a small hill, instead of the overall global maximum. Many hill-climbing algorithms, however, are used extensively even though they are not guaranteed to work, because in practice they seem to work well. In this section, we describe a hill-climbing algorithm that is guaranteed to quickly solve its problem.

Basic Ideas: I will start by giving some ideas that do not work.

Push from Source: The first obvious thing to try is to simply start pushing water out of s . If the capacities of the edges near s are large, then they can take lots of flow. Further down the network, however, the capacities may be smaller, in which case the flow that we started will get stuck. To avoid causing capacity violation

or leaks, we will have to back off the flow that we started. Even further down the network, an edge may fork into edges with larger capacities, in which case we will need to decide in which direction to route the flow. Keeping track of this could be a headache.

Plan Path for a Drop of Water: A solution to both the problem of flow getting stuck and the problem of routing flow along the way is to first find an entire path from s to t through which flow can take place. In the example in [Figure 21.2](#), water can flow along the path $\langle s, b, c, t \rangle$: see the top middle path. We then can push as much as possible through this path. It is easy to see that the bottleneck is the edge $\langle b, c \rangle$ with capacity 30. Hence, we add a flow of 30 to each edge along this path. That working well, we can try adding more water through another path. Let us try the path $\langle s, a, c, t \rangle$. The first interesting thing to note is that the edge $\langle c, t \rangle$ in this path already has flow 30 through it. Because this edge has a capacity of 75, the maximum flow that can be added to it is $75 - 30 = 45$. This, however, turns out not to be the bottleneck, because the edge $\langle s, a \rangle$ has capacity 20. Adding a flow of 20 to each edge along this path gives the flow shown on the right in [Figure 21.2](#). For each edge, the left value gives its flow and the right gives its capacity. There being no more paths forward from s to t , we are now stuck. Is this the maximum flow?

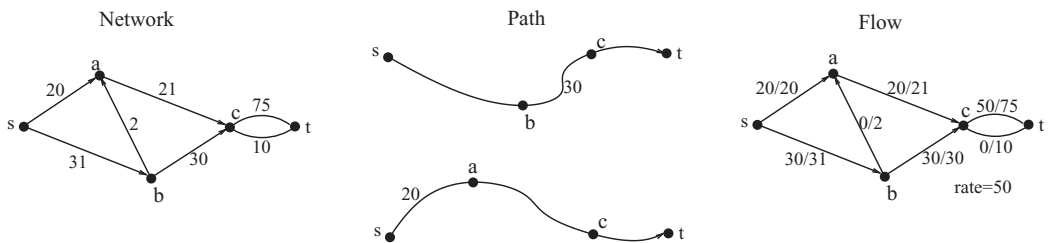


Fig. 21.2 A network, with its edge capacities labeled, is given on the left. In the middle are two paths through which flow can be pushed, with the resulting flow on the right. The first value associated with each edge is its flow, and the second is its capacity. The total rate of the flow is 50.

A Winding Path: Water has a funny way of seeping from one place to another. It does not need to only go forward. Though the path $\langle s, b, a, c, t \rangle$ winds backward, more flow can be pushed through it. Another way to see that the concept of “forward” is not relevant to this problem. The bottleneck in adding flow through this path is the edge $\langle a, c \rangle$. Already having a flow 20, its flow can only increase by 1. Adding a flow of 1 along this path gives the flow shown on the bottom left in [Figure 21.3](#). Though this example reminds us that we need to consider all possible paths from s to t , we know that finding paths through a graph is easy using either breadth-first or depth-first search ([Sections 20.2](#) and [20.4](#)). However, in addition, we want to make sure that the path we find is such that

we can add a nonzero amount of flow through it. For this, we introduce the idea of an *augmentation graph*.

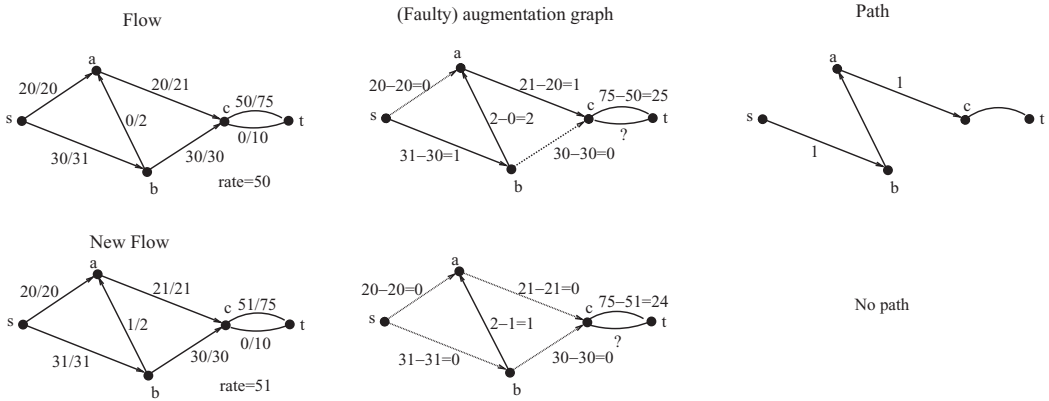


Fig. 21.3 The top left is the same flow given in Figure 21.2, the first value associated with each edge being its flow, and the second being its capacity. The top middle is a first attempt at an augmentation graph for this flow. Each edge is labeled with the amount of more flow that it can handle, namely $c_{\langle u,v \rangle} - F_{\langle u,v \rangle}$. The top right is the path in this augmentation graph through which flow is augmented. The bottom left is the resulting flow. The bottom middle is its (faulty) augmentation graph. No more flow can be added through it.

The (Faulty) Augmentation Graph: Before we can find a path through which more flow can be added, we need to compute for each edge the amount of flow that can be added through it. To keep track of this information, we construct from the current flow F a graph denoted by G_F and called an *augmentation graph*. (Augment means to add on. The augmentation is the amount you add on.) This graph initially will have the same directed edges as our network, G . Each of these edges is labeled with the amount by which its flow can be increase. We will call this the edge's *augmentation capacity*. Assuming that the current flow through the edge is $F_{\langle u,v \rangle}$ and its capacity is $c_{\langle u,v \rangle}$, this augmentation capacity is given by $c_{\langle u,v \rangle} - F_{\langle u,v \rangle}$. Any edge for which this capacity is zero is deleted from the augmentation graph. Because of this, nonzero flow can be added along any path found from s to t within this augmentation graph. The path chosen will be called the *augmentation path*. The minimum augmentation capacity of any of its edges is the amount by which the flow in each of its edges is augmented. For an example, see Figure 21.3. In this case, the only path happens to be $\langle s, b, a, c, t \rangle$, which is the path that we used. Its path is augmented by a flow of 1.

The (Faulty) Algorithm: We have now defined the basic steps and can easily fill in the remaining detail of the algorithm.

The Loop Invariant: The most obvious loop invariant is that at the top of the main loop we have a legal flow. It is possible that some more complex invariant will be needed, but for the time being this seems to be enough.

The Measure of Progress: The obvious measure of progress is how much flow the algorithm has managed to get between s and t , that is, the rate $rate(F)$ of the current flow.

The Main Steps: Given some current legal flow F through the network G , the algorithm improves the flow as follows: It constructs the augmentation graph G_F for the flow; finds an augmentation path from s to t through this graph using breadth-first or depth-first search; finds the edge in the path whose augmentation capacity is the smallest; and increases the flow by this amount through each edge in the path.

Maintaining the Loop Invariant: We must prove that the newly created flow is a legal flow in order to prove that $\langle loop\text{-}invariant' \rangle \& \text{not}(\text{exit-cond}) \& code_{loop} \Rightarrow \langle loop\text{-}invariant'' \rangle$.

Edge Capacity: We are careful never to increase the flow of any edge by more than the amount $c_{\langle u,v \rangle} - F_{\langle u,v \rangle}$. Hence, its flow never increases beyond its capacity $c_{\langle u,v \rangle}$.

No Leaks: We are careful to add the same amount to every edge along a path from s to t . Hence, for any node u along the path, there is one edge $\langle v,u \rangle$ into the node whose flow changes and one edge $\langle u,v' \rangle$ out of the node whose flow changes. Because these change by the same amount, the flow into the node remains equal to that out, so that for all nodes $u \notin \{s,t\}$, we have $\sum_v F_{\langle v,u \rangle} = \sum_v F_{\langle u,v \rangle}$. In this way, we maintain the fact that the current flow has no leaks.

Making Progress: Because the edges whose flows could not change were deleted from the augmenting graph, we know that the flow through the path that was found can be increase by a positive amount. This increases the total flow. Because the capacities of the edges are integers, we can prove inductively that the flows are always integers and hence the flow increases by at least one. (Having fractions as capacities is fine, but having irrationals as capacities can cause the algorithm to run forever.)

Initial Code: We can start with a flow of zero through each edge. This establishes the loop invariant, because it is a legal flow.

Exit Condition: At the moment, it is hard to imagine how we will know whether or not we have found the maximum flow. However, it is easy to see what will cause our algorithm to get stuck. If the augmenting graph for our current flow is such that there is no path in it from s to t , then unless we can think of something better to do, we must exit.

Termination: As usual, we prove that this iterative algorithm eventually terminates because at every iteration the rate of flow increases by at least one and because the total flow certainly cannot exceed the sum of the capacities of all the edges.

This completely defines an algorithm.

Getting Stuck at a Local Maximum: Hill-climbing algorithms move up until they cannot go up anymore. The reason that they are not allowed to go down is the same reason that iterative algorithms need to make progress every iteration, namely, to ensure that the algorithm eventually stops.

A major problem with this is that sometimes they find a small *local maximum*, that is, the top of a small hill, instead of the overall global maximum. Because a hill-climbing algorithm is not allowed to move down, it gets stuck at such a local maximum.

This is similar to the class of algorithms known as *greedy algorithms*, described in [Chapter 22](#). In these, no decision that is made is revoked. Our network flow algorithm could be considered to be greedy in that once the algorithm decides to put flow through an edge, it may later add more, but it never removes flow. Given that our goal is to get as much flow from s to t as possible and that it does not matter how that flow gets there, it makes sense that such a greedy approach would work. However, we will see that it does not work.

A Counterexample: Proving that a given algorithm works for every input instance can be a major challenge. However, in order to prove that it does not work, we only need to give one input instance in which it fails. [Figure 21.4](#) gives such an example. It traces out the algorithm on the same instance from [Figure 21.2](#) that we did before. However, this time the algorithm happens to choose different paths. First it puts a flow of 2 through the path $\langle s, b, a, c, t \rangle$, followed by a flow of 19 through $\langle s, a, c, t \rangle$, followed by a flow of 29 through $\langle s, b, c, t \rangle$. At this point, we are stuck because the augmenting graph does not contain a path from s to t . This is a problem because the current flow is only 50, whereas we have already seen that the flow for this network can be 51. In hill-climbing terminology, this flow is a small local maximum, because we cannot improve it using the steps that we have allowed; but it is not a global maximum, because there is a better solution.

Where We Went Wrong: From a hill-climbing perspective, we took a step in an arbitrary direction that takes us up, but with our first attempt we happened to head up the big hill and in the second we happened to head up the small hill. The flow of 51 that we obtained first turns out to be the unique maximum solution (often there are more than one possible maximum solutions). Hence, we can compare it with our present solution to see where we went wrong. In the first step, we put a flow of 2 through the edge $\langle b, a \rangle$; however, in the end it turns out that putting more than 1 through it is a mistake.

Fixing the Algorithm: The following are possible ways of fixing the kind of bugs we found.

Make Better Decisions: If we start by putting flow through the path $\langle s, b, c, t \rangle$, then the algorithm works, but if we start with the path $\langle s, b, a, c, t \rangle$, it does not. One way of fixing the bug is to find some way to choose which path to add flow to next so that we do not get stuck in this way. From the greedy algorithm's perspective, if we are going to commit to a choice, then we had better make a good one. I know of no way to fix the network flow algorithm in this way.

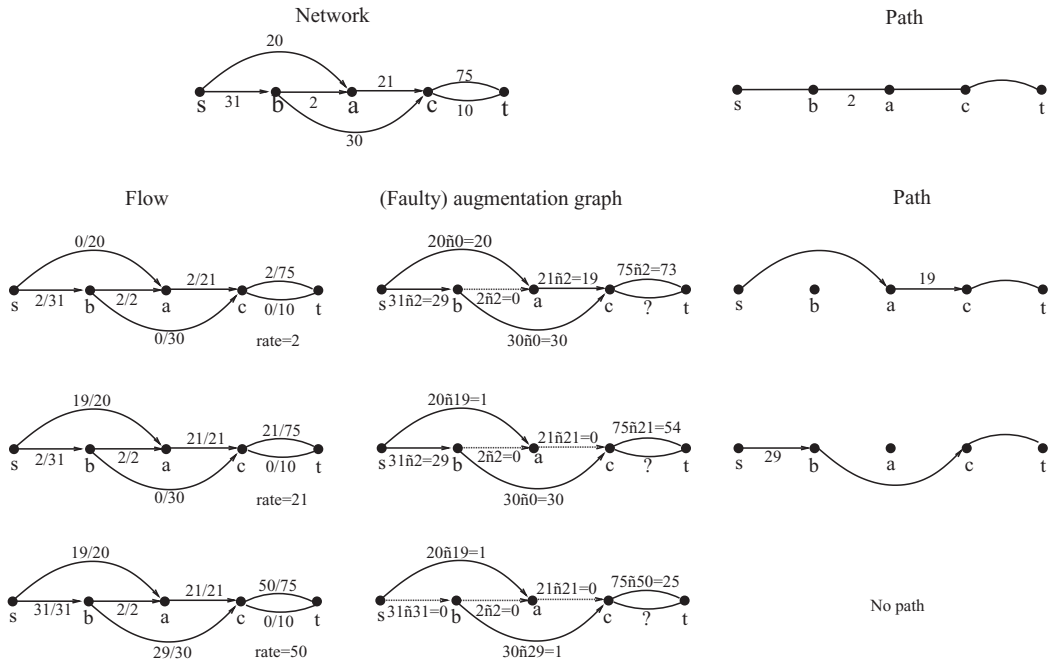


Fig. 21.4 The faulty algorithm is traced on the instance from Figure 21.2. The nodes in this graph are laid out differently to emphasize the first path chosen. The current flow is given on the left, the corresponding augmentation graph in the middle, the augmenting path on the right, and the resulting flow on the next line. The algorithm gets stuck at a suboptimal local maximum.

Backtrack: Chapter 23 describes another class of algorithms, known as recursive backtracking algorithms, that continually notice when they have made a mistake and that backtrack, trying other options, until a correct sequence of choices is made. In this example, we need to find a way of decreasing the flow through the edge (b, a) from 2 to 1. A general danger of backtracking algorithms in comparison with greedy algorithms is that the algorithm will have a much longer running time if it keeps changing its mind.

Take Bigger Steps: One way of avoiding getting stuck at the top of a small hill is to take a step that is big enough so that you step over the valley onto the slope of the bigger hill and a little higher up. Doing this requires you to redefine your definition of a step. This is the approach that we will take. We need to find a way of decreasing the flow through the edge (b, a) from 2 to 1 while maintaining the loop invariant that we have a legal flow and increasing the overall flow from s to t . The place in the algorithm at which we consider how the flow through an edge is allowed to change is where we define the augmenting graph. In the next section we reconsider this definition.

21.2 The Primal–Dual Hill-Climbing Method

We will now define larger step that the hill-climbing algorithm may take in hopes of avoiding a local maximum.

The (Correct) Algorithm

The Augmentation Graph: As before, the augmentation graph expresses how the flow in each edge is able to change.

Forward Edges: As before, when an edge $\langle u, v \rangle$ has flow $F_{\langle u, v \rangle}$ and capacity $c_{\langle u, v \rangle}$, we put the corresponding edge $\langle u, v \rangle$ in the augmentation graph with augmentation capacity $c_{\langle u, v \rangle} - F_{\langle u, v \rangle}$ to indicate that we are allowed to add this much flow from u to v .

Reverse Edges: Now we see that there is a possibility that we might want to decrease the flow from u to v . Given that its current flow is $F_{\langle u, v \rangle}$, this is the amount that it can be decreased by. Effectively, it is the same as increasing the flow from v to u by this same amount. Moreover, if the reverse edge $\langle v, u \rangle$ is also in the graph and has capacity $c_{\langle v, u \rangle}$, then we are able to increase the flow from v to u by this second amount $c_{\langle v, u \rangle}$ as well. Therefore, when the edge $\langle u, v \rangle$ has flow $F_{\langle u, v \rangle}$ and the reverse edge $\langle v, u \rangle$ has capacity $c_{\langle v, u \rangle}$, we also put the reverse edge $\langle v, u \rangle$ in the augmentation graph with augmentation capacity $F_{\langle u, v \rangle} + c_{\langle v, u \rangle}$. For more intuition, see [Figure 21.5](#), and for an example, see edge $\langle c, t \rangle$ in the second augmentation graph in [Figure 21.6](#).

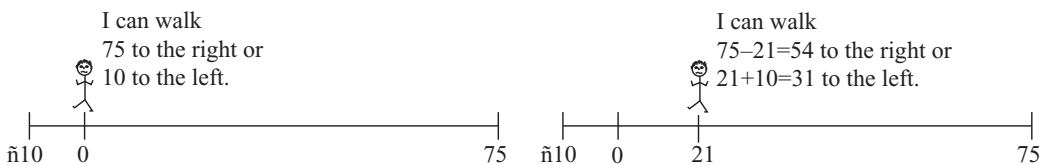


Fig. 21.5 Suppose that from my home, I can walk 75 km to the right or 10 to the left. If I am already 21 km to the right, then I can walk $75 - 21 = 54$ km to the right or $21 + 10 = 31$ to the left. Moreover, walking 31 to the left and -31 to the right are the same. Similarly, suppose that my bank account can only hold \$75 or go into overdraft of up to \$10. If I already have \$21 in the account, then I am able to add $75 - 21 = \$54$ or remove $21 + 10 = \$31$. Removing \$31 and adding $-\$31$ are also the same.

The Main Steps: Little else changes in the algorithm. Given some current legal flow F through the network G , the algorithm improves the flow as follows: It constructs the augmentation graph G_F for the flow; finds an augmentation path from s to t through this graph using breadth-first or depth-first search; finds the edge in the path whose augmentation capacity is the smallest; and increases the flow by this amount through each edge in the path. If the edge in the augmenting graph is in the opposite direction to that in the flow graph, then this involves

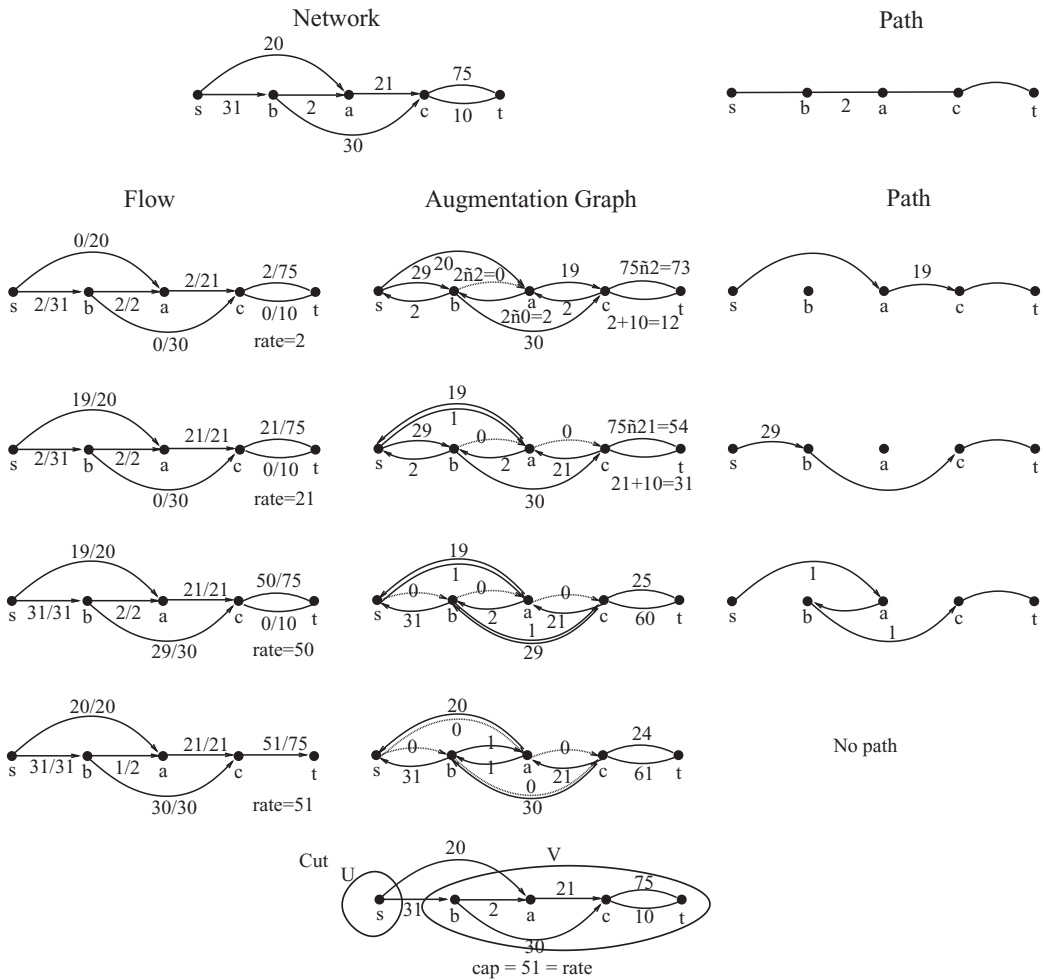


Fig. 21.6 A trace of the correct algorithm on the instance from Figure 21.4. The current flow is given on the left, the corresponding augmentation graph in the middle, the augmentation path on the right, and the resulting flow on the next line. The optimal flow is obtained. The bottom diagram shows a minimum cut $C = \langle U, V \rangle$.

decreasing its flow by this amount. This is because increasing flow from v to u is effectively the same as decreasing it from u to v .

Does It Work on the Counterexample?: Figure 21.6 traces this new algorithm on the same example as in Figure 21.4. The new augmenting graphs include edges in the reverse direction. Each step is the same as that in Figure 21.4, until the last step, in which these reverse edges provide the path $\langle s, a, b, c, t \rangle$ from s to t . The bottleneck in this path is 1. Hence, we increase the flow by 1 in each edge in the path. The effect is that the flow through the edge $\langle b, a \rangle$ decreases from 2 to 1, giving the optimal flow that we had obtained before.

Bigger Step: The reverse edges that have been added to the augmentation graph may well not be needed. They do, after all, undo flow that has already been added through an edge. On the other hand, having more edges in the augmentation graph can only increase the possibility of there being a path from s to t through it.

Maintaining the Loop Invariant and Making Progress: See [Exercise 21.2.1](#).

Exit Condition: As before, the algorithm exits when it gets stuck because the augmenting graph for our current flow is such that there is no path in it from s to t . However, with more edges in our augmenting graph, this may not occur as soon.

Code

algorithm *NetworkFlow*(G, s, t)

(pre-cond): G is a network given by a directed graph with capacities on the edges. s is the source node. t is the sink.

(post-cond): F specifies a maximum flow through G , and C specifies a minimum cut.

begin

F = the zero flow

loop

(loop-invariant): F is legal flow.

G_F = the augmentation graph for F , where

edge $\langle u, v \rangle$ has augmentation capacity $c_{\langle u, v \rangle} - F_{\langle u, v \rangle}$ and

edge $\langle v, u \rangle$ has augmentation capacity $c_{\langle v, u \rangle} + F_{\langle u, v \rangle}$.

exit when s is not connected to t in G_F

P = a path from s to t in G_F

w = the minimum augmentation capacity in P

Add w to the flow F in every edge in P

end loop

U = nodes reachable from s in G_F

V = nodes not reachable from s in G_F

$C = \langle U, V \rangle$

return(F, C)

end algorithm

Ending: The next step is to prove that this improved algorithm always finds a global maximum without getting stuck at a small local maximum. Using the notation of iterative algorithms, we must prove that *(loop-invariant)* & *(exit-cond)* & *code_{post-loop}* $\Rightarrow$ *(post-cond)*. From the loop invariant we know that the algorithm has a legal flow. Because we have exited, we know that the augmenting graph does not contain a path from s to t and hence we are at a local maxima. We must prove that there are no small local maxima and hence we must be at a global maximum and hence have an optimal flow. The method used is called the *primal–dual method*.

Primal–Dual Hill Climbing: Suppose that over the hills on which we are climbing there are an exponential number of roofs, one on top of the other. As before, our problem is to find a place to stand on the hills that has maximum height. We call this the *primal* optimization problem. An equally challenging problem is to find the lowest roof. We call this the *dual* optimization problem. The situation is such that each roof is above each place to stand. It follows trivially that the lowest and hence optimal roof is above the highest and hence optimal place to stand, but offhand we do not know how far above it is.

We say that a hill-climbing algorithm gets stuck when it is unable to step in a way that moves it to a higher place to stand. A primal–dual hill-climbing algorithm is able to prove that the only reason for getting stuck is that the place it is standing is pressed up against a roof. This is proved by proving that from any location, it can either step to a higher location or specify a roof to which this location is adjacent. We will now see how these conditions are sufficient for proving what we want.

Lemma: Finds Optimal. A primal–dual hill-climbing algorithm is guaranteed to find an optimal solution to both the primal and the dual optimization problems.

Proof: By the design of the algorithm, it only stops when it has a location L and a roof R with matching heights $\text{height}(L) = \text{height}(R)$. This location must be optimal, because every other location L' must be below this roof and hence cannot be higher than this location, that is, $\forall L', \text{height}(L') \leq \text{height}(R) = \text{height}(L)$. We say that this dual solution R *witnesses* the fact that the primal solution L is optimal. Similarly, L witnesses the fact that R is optimal, that is, $\forall R', \text{height}(R') \geq \text{height}(L) = \text{height}(R)$. This is called the *duality principle*.

Cuts as Upper Bounds: In order to apply these ideas to the network flow problem, we must find some upper bounds on the flow between s and t . Through a single path, the capacity of each edge acts as an upper bound, because the flow through the path cannot exceed the capacity of any of its edges. The edge with the smallest capacity, being the lowest upper bound, is the bottleneck. In a general network (see [Figure 21.1](#)), a single edge cannot act as a bottleneck, because the flow might be able to go around this edge via other edges. A similar approach, however, works. Suppose that we wanted to bound the traffic between Toronto and Berkeley. We know that any such flow must cross the Canadian–US border. Hence, there is no need to worry about what the flow might do within Canada or within the USA. We can safely say that the flow from Toronto to Berkeley is bounded above by the sum of the capacities of all the border crossings. Of course, this does not mean that that flow can be achieved. Other upper bounds can be obtained by summing the border crossing for other regions. For example, you could bound the traffic leaving Toronto, leaving

Ontario, entering California, or entering Berkeley. This brings us to the following definition.

Cut of a Graph: A cut $C = \langle U, V \rangle$ of a graph is a partitioning of the nodes of the graph into two sets U and V such that the source s is in U and the sink t is in V . The capacity of a cut is the sum of the capacities of all edges from U to V , namely, $cap(C) = \sum_{u \in U} \sum_{v \in V} c_{\langle u, v \rangle}$.

Because the nodes in a graph do not have a location as cities do, there is no reason for the partition of the nodes to be geographically contiguous. Any one of the exponential number of partitions will do.

Flow across a Cut: To be able to compare the rate of flow from s to t with the capacity of a cut, we will first need to define the flow across a cut.

rate(F, C): Define $rate(F, C)$ to be the current flow F across the cut C , which is the total of all the flow in edges that cross from U to V minus the total of all the flow that comes back, i.e., $rate(F, C) = \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle} - F_{\langle v, u \rangle}]$.

rate(F) = rate($F, \langle \{s\}, G - \{s\} \rangle$): We defined the flow from s to t to be the total flow that leaves s without coming back, namely, $rate(F) = \sum_v [F_{\langle s, v \rangle} - F_{\langle v, s \rangle}]$. This is precisely the equation for the flow across the cut that puts s all by itself, namely, $rate(F) = rate(F, \langle \{s\}, G - \{s\} \rangle)$.

Lemma: rate(F, C) = rate(F): Intuitively this makes sense. Because no water leaks or is created between the source s and the sink t , the flow out of s equals the flow across any cut between s and t , which in turn equals the flow into t . It is because these are the same that we simply call this the flow from s to t . Since the flow into a node is the same as that out of the node, if you move the node from one side of the cut to the other this does not change the total flow across the cut. Hence we can change the cut one node at a time from being the one containing only s to being the cut that we are interested in.

More formally this is done by induction on the size of U . For the base case, $rate(F) = rate(F, \langle \{s\}, G - \{s\} \rangle)$ gives us that our hypothesis $rate(F, C) = rate(F)$ is true for every cut that has only one node in U . Now suppose that, by way of induction, we assume that it is true for every cut that has i nodes in U . We will now prove it for those cuts that have $i + 1$ nodes in it. Let $C = \langle U, V \rangle$ be any such cut. Choose one node x (other than s) from U and move it across the border. This gives us a new cut $C' = \langle U - \{x\}, V \cup \{x\} \rangle$, where the side $U - \{x\}$ contains only i nodes. Our assumption then gives us that the flow across this cut is equal to the flow of F : $rate(F, C') = rate(F)$. Hence, in order to prove that $rate(F, C) = rate(F)$, we only need to prove that $rate(F, C) = rate(F, C')$. We will do this by proving that the difference between these is zero. By definition,

$$\begin{aligned}
& \text{rate}(F, C) - \text{rate}(F, C') \\
&= \left[\sum_{u \in U} \sum_{v \in V} F_{\langle u, v \rangle} - F_{\langle v, u \rangle} \right] - \left[\sum_{u \in U - \{x\}} \sum_{v \in V \cup \{x\}} F_{\langle u, v \rangle} - F_{\langle v, u \rangle} \right] \\
&\quad \text{(Figure 21.7 shows the terms that do not cancel.)} \\
&= \left[\sum_{v \in V} F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] - \left[\sum_{u \in U} F_{\langle u, x \rangle} - F_{\langle x, u \rangle} \right] \\
&= \left[\sum_{v \in V} F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] + \left[\sum_{v \in U} F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] \\
&= \left[\sum_v F_{\langle x, v \rangle} - F_{\langle v, x \rangle} \right] = 0.
\end{aligned}$$

This is the total flow out of the node x , minus the total flow into the node, which is zero by the requirement that no node leaks. This proves that $\text{rate}(F, C) = \text{rate}(F)$ for every cut that has $i + 1$ nodes in U . By induction, it then is true for all cuts for every size of U . This formally proves that the rate of any given flow F is the same across any cut C .

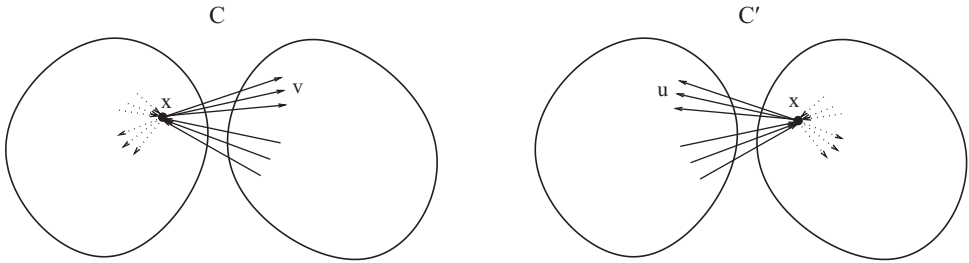


Fig. 21.7 The edges across the cut that do not cancel in $\text{rate}(F, C) - \text{rate}(F, C')$.

Lemma: $\text{rate}(F) \leq \text{cap}(C)$: It is now easy to prove that $\text{rate}(F)$ of any flow F is at most the capacity of any cut C . In the primal–dual analogy, this proves that each roof is above each place to stand. Given $\text{rate}(F) = \text{rate}(F, C)$, it is sufficient to prove that the flow across a cut is at most the capacity of the cut. This follows easily from the definition $\text{rate}(F, C) = \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle} - F_{\langle v, u \rangle}] \leq \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle}]$, because having positive flow backward across the cut from V to U only decreases the flow. Then this sum is at most $\sum_{u \in U} \sum_{v \in V} [c_{\langle u, v \rangle}]$, because no edge can have flow exceeding its capacity. This is the definition of the capacity $\text{cap}(C)$ of the cut. This proves the required $\text{rate}(F) \leq \text{cap}(C)$.

Take a Step or Find a Cut: The primal–dual method requires that from any location, you can either step to a higher location or specify a roof to which this location is adjacent. In the network flow problem, this translates to: given any legal flow F , either find a better flow or find a cut whose capacity is equal to the rate of the current flow. The augmentation graph G_F includes those edges through which the flow rate can be increased. Hence, the nodes reachable from s in this graph are the nodes to which more flow could be pushed. Let U denote this set of nodes. In contrast, the remaining set of nodes, which we will denote by V , are those to which more flow cannot be pushed. See the cut at the bottom of Figure 21.6. No flow can be pushed across the border between U and V , because all the edges crossing over are at capacity. If t is in U , then there is a path from s to t through which the flow can be increased. On the other hand, if t is in V , then $C = \langle U, V \rangle$ is a cut separating s and t . What remains is to formalize the proof that the capacity of this cut is equal to rate of the current flow. (For another example, see the cut in Figure 21.1.c. Although $\text{cap}(C) = \text{rate}(F)$, this cut was not formed as described here because the node i is not reachable from s in the augmentation graph.)

Since we know $\text{rate}(F, C) = \text{rate}(F)$, it remains only to prove $\text{rate}(F, C) = \text{cap}(C)$, that is, that the current flow across the cut C is equal to the capacity of the cut.

Lemma: $\text{rate}(F, C) = \text{cap}(C)$: To prove this, it is sufficient to prove that every edge $\langle u, v \rangle$ crossing from U to V has flow in F at capacity ($F_{\langle u, v \rangle} = c_{\langle u, v \rangle}$) and every edge $\langle v, u \rangle$ crossing back from V to U has zero flow in F . These give that $\text{rate}(F, C) = \sum_{u \in U} \sum_{v \in V} [F_{\langle u, v \rangle} - F_{\langle v, u \rangle}] = \sum_{u \in U} \sum_{v \in V} [c_{\langle u, v \rangle} - 0] = \text{cap}(C)$.

$F_{\langle u, v \rangle} = c_{\langle u, v \rangle}$: Consider any edge $\langle u, v \rangle$ crossing from U to V . If $F_{\langle u, v \rangle} < c_{\langle u, v \rangle}$, then the edge $\langle u, v \rangle$ with augmentation capacity $c_{\langle u, v \rangle} - F_{\langle u, v \rangle}$ would be added to the augmentation graph. However, having such an edge in the augmentation graph contradicts the fact that u is reachable from s in the augmentation graph and v is not.

$F_{\langle v, u \rangle} = 0$: If $F_{\langle v, u \rangle} > 0$, then the edge $\langle u, v \rangle$ with augmentation capacity $c_{\langle u, v \rangle} + F_{\langle v, u \rangle}$ would be added to the augmentation graph. Again, having such an edge is a contradiction.

This proves that $\text{rate}(F, C) = \text{cap}(C)$.

Lemma: $\text{cap}(C) = \text{rate}(F)$: $\text{cap}(C) = \text{rate}(F, C) = \text{rate}(F)$, that is, the flow we have found equals the capacity of the cut, as required.

Ending: This improved network flow algorithm always finds a global maximum without getting stuck at a small local maximum. In each iteration it either finds a path in the augmenting graph through which it can improve the current flow or finds a cut that witnesses the fact that there are no better flows.

Max-Flow–Min-Cut Duality Principle: The max-flow and min-cut problems were defined at the beginning of this chapter, both being interesting problems in their own right. Now we see that cuts can be used as the ceilings on the flows.

Max Flow = Min Cut: We have proved that, given any network as input, the network flow algorithm finds a maximum flow and—almost as an accident—finds a minimum cut as well. This proves that the maximum flow through any network equals its minimum cut.

Dual Problems: We say that the *dual* of the max-flow problem is the min-cut problem, and conversely that the *dual* of the min-cut problem is the max-flow problem.

Credits: This algorithm was developed by Ford and Fulkerson in 1962.

Running Time Exponential? Suppose that the network graph has m edges, each with a capacity that is represented by an $O(\ell)$ bit number. Each capacity could be as large as $O(2^\ell)$, and the total maximum flow could be as large as $O(m \cdot 2^\ell)$. Starting out as zero and increasing by about one each iteration, the algorithm would need $O(m \cdot 2^\ell)$ iterations until the maximum flow is found. This running time is polynomial in the number of edges, m . However, the size of the input instance, which in this case is the number of bits (or digits) needed to represent all of the values, is $O(m \cdot \ell)$. If ℓ is large, then the number of iterations, $O(m \cdot 2^\ell)$, is exponential in this size. This is a common problem with hill-climbing algorithms.

Exercise 21.2.1 Prove that the network flow algorithm presented in this section maintains the loop invariant that it always holds a legal flow. Do this by proving that the changes to the flow do not violate any edge capacities or create leaks at nodes. Also prove that progress is made because the total flow increases. You need to be careful with your plus and minus signs.

Exercise 21.2.2 Starting with the flow in Figure 21.8, complete the network flow algorithm.

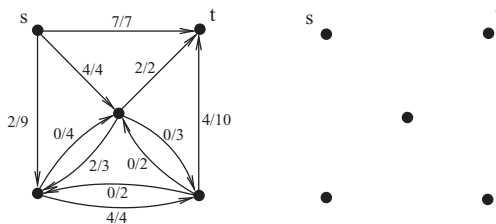


Fig. 21.8

Exercise 21.2.3 In hill-climbing algorithms there are steps that make lots of progress and steps that make very little progress. For example, the first iteration on the input given in Figure 21.2 might find a path through the augmentation graph through which

a flow of 30 can be added. It might, however, find the path through which only a flow of 2 can be added. How bad might the running time be when the computation is unlucky enough to always take the worst legal step allowed by the algorithm? Start by taking the step that increases the flow by 2 for the input given in [Figure 21.2](#). Then continue to take the worst possible step. You could draw out each and every step, but it is better to use this opportunity to use loop invariants. What does the flow look like after i iterations? Repeat this process on the same graph except that the four edges forming the square now have capacities 1,000,000,000,000,000 and the crossover edge has capacity 1. (Also move t to c give that last edge a large capacity.)

1. What is the worst case number of iterations of this network flow algorithm as a function of the number of edges m in the input network?
2. What is the official “size” of a network?
3. What is the worst case number of iterations of this network flow algorithm as a function of the size of the input network?

Exercise 21.2.4 If all the capacities in the given network are integers, prove that the algorithm always returns a solution in which the flow through each edge is an integer. For some applications, this fact is crucial.

Exercise 21.2.5 (See solution in Exercise Solutions.) Give an algorithm for solving the *min cut problem*. Given a network $\langle G, s, t \rangle$ with capacities on the edges, find a minimum cut $C = \langle U, V \rangle$ where $s \in U$ and $t \in V$. The cost of the cut is its capacity $\text{cap}(C) = \sum_{u \in U} \sum_{v \in V} c_{\langle u, v \rangle}$. (Hint: you have already been told how to do this.)

21.3 The Steepest-Ascent Hill-Climbing Algorithm

We have all experienced that climbing a hill can take a long time if you wind back and forth, barely increasing your height at all. In contrast, you get there much faster if you head energetically straight up the hill. This method, which is called the method of *steepest ascent*, is to always take the step that increases your height the most. If you already know that the hill-climbing algorithm in which you take any step up the hill works, then this new, more specific algorithm also works. However, if you are lucky, it finds the optimal solution faster.

In our network flow algorithm, the choice of what step to take next involves choosing which path in the augmentation graph to take. The amount the flow increases is the smallest augmentation capacity of any edge in this path. It follows that the choice that would give us the biggest improvement is the path whose smallest edge is the largest for any path from s to t . Our steepest-ascent network flow algorithm will augment such a best path each iteration. What remains to be done is to give an algorithm that finds such a path and to prove that this finds a maximum flow within a polynomial number of iterations.

Finding the Augmentation Path with the Biggest Smallest Edge: The input consists of a directed graph with positive edge weights and with special nodes s and t .

The output consists of a path from s to t through this graph whose smallest weighted edge is as big as possible.

Easier Problem: Before attempting to develop an algorithm for this, let us consider an easier but related problem. In addition to the directed graph, the input to the easier problem provides a weight, denoted w_{min} . It either outputs a path from s to t whose smallest weighted edge is at least as big as w_{min} , or states that no such path exists.

Using the Easier Problem: Assuming that we can solve this easier problem, we solve the original problem by running the first algorithm with w_{min} being every edge weight in the graph, until we find the weight for which there is a path with such a smallest weight, but there is no path with a bigger smallest weight. This is our answer. (See [Exercise 21.3.1.](#))

Solving the Easier Problem: A path whose smallest weighted edge is at least as big as w_{min} will obviously not contain any edge whose weight is smaller than w_{min} . Hence, the answer to this easier problem will not change if we delete from the graph all edges whose weight is smaller. Any path from s to t in the remaining graph will meet our needs. If there is no such path, then we also know there is no such path in our original graph. This solves the problem.

Implementation Details: In order to find a path from s to t in a graph, the algorithm branches out from s using breadth-first or depth-first search, marking every node reachable from s with the predecessor of the node in the path to it from s . If in the process t is marked, then we have our path. (See [Section 20.1.](#)) It seems a waste of time to have to redo this work for each w_{min} , so let's use an iterative algorithm. The loop invariant will be that the work for the previous w_{min} has been done and is stored in a useful way. The main loop will then complete the work for the current w_{min} , reusing as much of the previous work as possible. This can be implemented as follows. Sort the edges from biggest to smallest (breaking ties arbitrarily). Consider them one at a time. When considering w_i , we must construct the graph formed by deleting all the edges with weights smaller than w_i . Denote this by G_{w_i} . We must mark every node reachable from s in this graph. Suppose that we have already done these things in the graph $G_{w_{i-1}}$. We form G_{w_i} from $G_{w_{i-1}}$ by adding the single edge with weight w_i . Let $\langle u, v \rangle$ denote this edge. Nodes are reachable from s in G_{w_i} that were not reachable in $G_{w_{i-1}}$ only if u was reachable and v was not. This new edge then allows v to be reachable. Unmarked nodes now reachable from s via v can all be marked reachable by starting a depth-first search from v . The algorithm will stop at the first edge that allows t to be reached. The edge with the smallest weight in this path to t will be the edge with weight w_i added during this iteration. There is no path from s to t in the input graph with a larger smallest weighted edge, because t was not reachable when only the larger edges were added. Hence, this path is a path to t in the graph whose smallest weighted edge is the largest. This is the required output of this subroutine.

Running Time: Even though the algorithm for finding the path with the largest smallest edge runs depth-first search for each weight w_i , because the work done before is reused, no node in the process is marked reached more than once, and hence no edge is traversed more than once. It follows that this process requires only $O(m)$ time, where m is the number of edges. This time, however, is dominated by the time $O(m \log m)$ to sort the edges.

Code

algorithm *LargestShortestWeight*(G, s, t)

(pre-cond): G is a weighted directed (augmenting) graph. s is the source node. t is the sink.

(post-cond): P specifies a path from s to t whose smallest edge weight is as large as possible. $\langle u, v \rangle$ is its smallest weighted edge.

begin

Sort the edges by weight from largest to smallest

$G' =$ graph with no edges

mark s reachable

loop

(loop-invariant): Every node reachable from s in G' is marked reachable.

exit when t is reachable

$\langle u, v \rangle =$ the next largest weighted edge in G

Add $\langle u, v \rangle$ to G'

if (u is marked reachable and v is not) then

Do a depth-first search from v , marking all reachable nodes
not marked before.

end if

end loop

$P =$ path from s to t in G'

return $(P, \langle u, v \rangle)$

end algorithm

Running Time of Steepest Ascent: How many times must the network flow algorithm augment the flow in a path when the path chosen is that whose augmentation capacity is the largest possible?

Decreasing the Remaining Distance by a Constant Factor: The flow starts out as zero and may need to increase to be as large as $O(m \cdot 2^\ell)$ when there are m edges with ℓ bit capacities. We would like the number of steps to be not exponential but linear in ℓ . One way to achieve this is to ensure that the current flow doubles each iteration. This, however, is likely not to happen. Another possibility is to turn the measure of progress around. After the i th iteration, let R_i denote the remaining amount that the flow must increase. More formally, suppose

that the maximum flow is $rate_{max}$ and that the rate of the current flow is $rate(F)$. The remaining distance is then $R_i = rate_{max} - rate(F)$. We will show that the amount w_{min} by which the flow increases is at least some constant fraction of R_i .

Bounding the Remaining Distance: The strange thing about this measure of progress is that the algorithm does not know what the maximum flow $rate_{max}$ is. It is only needed as part of the analysis. We must bound how big the remaining distance, $R_i = rate_{max} - rate(F)$, is. Recall that the augmentation graph for the current flow is constructed so that the augmentation capacity of each edge gives the amount that the flow through this edge can be increased by. Hence, just as the sum of the capacities of the edges across any cut $C = \langle U, V \rangle$ in the network acts as an upper bound to the total flow possible, the sum of the augmentation capacities of the edges across any cut $C = \langle U, V \rangle$ in the augmentation graph acts as an upper bound to the total amount that the current flow can be increased.

Choosing a Cut: We need to choose which cut we will use. (This is not part of the algorithm.) As before, the natural cut to use comes out of the algorithm that finds the path from s to t . Let $w_{min} = w_i$ denote the smallest augmentation capacity in the path whose smallest augmentation capacity is largest. Let $G_{w_{i-1}}$ be the graph created from the augmenting graph by deleting all edges whose augmentation capacities are smaller than or equal to w_{min} . This is the last graph that the algorithm that finds the augmenting path considers before adding the edge with weight w_{min} that connects s and t . We know that there is not a path from s to t in $G_{w_{i-1}}$, or else there would be an path in the augmenting graph whose smallest augmenting capacity was larger then w_{min} . Form the cut $C = \langle U, V \rangle$ by letting U be the set of all the nodes reachable from s in $G_{w_{i-1}}$ and letting V be those that are not. Now consider any edge in the augmenting graph that crosses this cut. This edge cannot be in the graph $G_{w_{i-1}}$, or else it would be crossing from a node in U that is reachable from s to a node that is not reachable from s , which is a contradiction. Because this edge has been deleted in $G_{w_{i-1}}$, we know that its augmentation capacity is at most w_{min} . The number of edges across this cut is at most the number of edges in the network, which has been denoted by m . It follows that the sum of the augmentation capacities of the edges across this cut $C = \langle U, V \rangle$ is at most $m \cdot w_{min}$.

Bounding the Increase, $w_{min} \geq \frac{1}{m} R_i$: We have determined that the remaining amount that the flow needs to be increased, $R_i = rate_{max} - rate(F)$, is at most the sum of the augmentation capacities across the cut C , which is at most $m \cdot w_{min}$, that is, $R_i \leq m \cdot w_{min}$. Rearranging this gives that $w_{min} \geq \frac{1}{m} R_i$.

The Number of Iterations: If the flow increases each iteration by at least $\frac{1}{m}$ times the remaining amount R_i , it decreases the remaining amount, giving that $R_{i+1} \leq R_i - \frac{1}{m} R_i$. You might think that it follows that the maximum flow is obtained in only m iterations. This would be true if $R_{i+1} \leq R_i - \frac{1}{m} R_0$. However, it is not, because the smaller R_i gets, the smaller the amount it decreases by. One way to bound the number of iterations needed is to note that $R_i \leq (1 - \frac{1}{m})^i R_0$ and

then either to bound logarithms to base $1 - \frac{1}{m}$ or to note that $\lim_{m \rightarrow \infty} (1 - \frac{1}{m})^m = \frac{1}{e} \approx \frac{1}{2.718}$. However, I prefer the following method. As long as R_i is big, we know that it decreases by a lot. After some i th iteration, say that R_i is still big when it is still at least $\frac{1}{2}R_I$. As long as this is the case, R_i decreases by at least $\frac{1}{m}R_i \geq \frac{1}{2m}R_I$. After m such iterations, R_i would decrease from R_I to $\frac{1}{2}R_I$. The only reason that it would not continue to decrease this fast would be that it already had decreased that much. Either way, we know that every m iterations, R_i decreases by a factor of two. This process may make you think of Zeno's paradoxes. If you cut the remaining distance in half and then in half again and so on, then though you get very close very fast, you never actually get there. However, if all the capacities are integers, then all values will be integers, and hence when R_i decreases to less than one, it must in fact be zero, giving us the maximum flow.

Initially, the remaining amount $R_i = \text{rate}_{\max} - \text{rate}(F)$ is at most $O(m \cdot 2^\ell)$. Hence, if it decreases by at least a factor of 2 each m iterations, then after mj iterations, this amount is at most $O(m \cdot 2^\ell / 2^j)$. This reaches one when $j = O(\log_2(m \cdot 2^\ell)) = O(\ell + \log m)$, or in $O(m\ell + m \log m)$ iterations. If your capacities are real numbers, then you will be able to approximate the maximum flow to within ℓ' bits of accuracy in another $m\ell'$ iterations.

Bounding the Running Time: We have determined that each iteration takes $m \log m$ time and that only $O(m\ell + m \log m)$ iterations are required. It follows that this steepest-ascent network flow algorithm runs in time $O(\ell m^2 \log m + m^2 \log^2 m)$.

Fully Polynomial Time: A lot of work has been done finding an algorithm that is what is known as *fully polynomial*. This requires that the number of iterations be polynomial in the number of values and not depend at all on the values themselves. Hence, if you charge only one time step for addition and subtraction, even if the capacities are strange things like $\sqrt{2}$, then the algorithm gives the exact answer (at least symbolically) in polynomial time. My father, Jack Edmonds, and a colleague, Richard Karp, developed such an algorithm in 1972. It is a version of the original Ford–Fulkerson algorithm. In it, however, in each iteration, the path from s to t in the augmentation graph with the smallest number of edges is augmented. This algorithm iterates at most $O(nm)$ times, where n is the number of nodes and m the number of edges. In practice, this is slower than the $O(m\ell)$ -time steepest-ascent algorithm.

Exercise 21.3.1 Could we use binary search on the weights $w_{\min}$ to find the critical weight (see Section 1.4), and if so, would that be faster? Why?

21.4 Linear Programming

When I was an undergraduate, I had a summer job with a food company. Our goal was to make cheap hot dogs. Every morning we got the prices of thousands of ingredients: pig

hearts, sawdust, etc. Each ingredient had an associated variable indicating how much of it to add to the hot dogs. There are thousands of linear constraints on these variables: so much meat, so much moisture, and so on. Together these constraints specify which combinations of ingredients constitute a hot dog. The cost of the hot dog is a linear function of the quantities you put into it and their prices. The goal is to determine what to put into the hot dogs that day to minimize the cost. This is an example of a general class of problems referred to as *linear programs*.

Formal Specification: A linear program is an optimization problem whose constraints and objective functions are linear functions. The goal is to find a setting of variables that optimizes the objective function, while respecting all of the constraints.

Precondition: We are given one of the following instances.

Instances: An input instance consists of (1) a set of linear constraints on a set of variables, and (2) a linear objective function.

Postcondition: The output is a solution with minimum cost and the cost of that solution.

Solutions for Instance: A solution for the instance is a setting of all the variables that satisfies the constraints.

Measure of Success: The cost or value of a solutions is given by the objective function.

Example 21.4.1 (*Instance Example*)

$$\begin{aligned}
 &\text{maximize} \\
 &\quad 7x_1 - 6x_2 + 5x_3 + 7x_4 \\
 &\text{subject to} \\
 &\quad 3x_1 + 7x_2 + 2x_3 + 9x_4 \leq 258 \\
 &\quad 6x_1 + 3x_2 + 9x_3 - 6x_4 \leq 721 \\
 &\quad 2x_1 + 1x_2 + 5x_3 + 5x_4 \leq 524 \\
 &\quad 3x_1 + 6x_2 + 2x_3 + 3x_4 \leq 411 \\
 &\quad 4x_1 - 8x_2 - 4x_3 + 4x_4 \leq 685.
 \end{aligned}$$

Matrix Representation: A linear program can be expressed very compactly using matrix algebra. Let n denote the number of variables and m the number of constraints. Let a denote the row of n coefficients in the objective function, let M denote the matrix with m rows and n columns of coefficients on the left-hand side of the constraints, let b denote the column of m coefficients on the right-hand side of the constraints, and finally let x denote the column of n variables. Then the goal of the linear program is to maximize $a \cdot x$ subject to $M \cdot x \leq b$.

Network Flows: The network flows problem can be expressed as instances of linear programming. See [Exercise 21.4.1](#).

The Euclidean Space Interpretation: Each possible solution, giving values to the variables $x_1, \dots, x_n$ can be viewed as a point in n -dimensional space. This space is easiest to view when there are only two or three dimensions, but the same ideas hold for any number of variables.

Constraints: Each constraint specifies a boundary in space, on one side of which a valid solution must lie. When $n = 2$, this constraint is a one-dimensional line. See Figure 21.9. When $n = 3$, it is a two-dimensional plane, like the side of a box. In general, it is an $(n - 1)$ -dimensional space. The space bounded by all of the constraints is called a *polyhedral*.

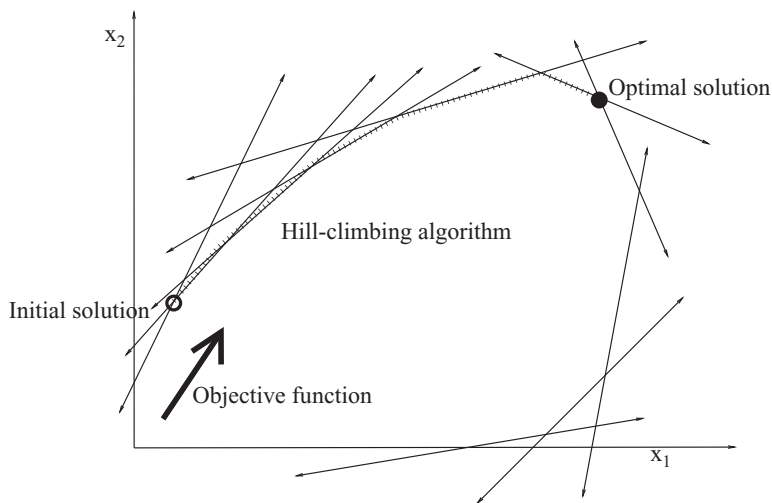


Fig. 21.9 The Euclidean space representation of a linear program with $n = 2$.

Vertices: The boundary of the polyhedral is defined by *vertices* where a number of constraints intersect. When $n = 2$, pairs of line constraints intersect at a vertex. See Figure 21.9. For $n = 3$, three sides of a box define a vertex (corner). In general, it requires n constraints to intersect to define a single vertex. This is because saying that the solution is on the constraint is saying that the linear equation holds with equality and not with “less than or equal to.” Then recall that n linear equations with n unknowns are sufficient to specify a unique solution.

The Objective Function: The objective function gives a direction in Euclidean space. The goal is to find a point in the bounded polyhedral that is the furthest in this direction. The best way to visualize this is to rotate the Euclidean space so that the objective function points straight up. The goal is to find a point in the bounded polyhedral that is as high as possible.

A Vertex Is an Optimal Solution: As you can imagine from looking at Figure 21.9, if there is a unique solution, it will be at a vertex where n constraints meet. If

there is a whole region of equivalently optimal solutions, then at least one of them will be a vertex. Our search for an optimal solution will focus on these vertices.

The Hill-Climbing Algorithm: The obvious algorithm simply climbs the hill formed by the outside of the bounded polyhedral until the top is reached. In defining a hill-climbing algorithm for linear programming we just need to devise a way to find an initial valid solution and to define what constitutes a step to a better solution.

A Step: Suppose, by the loop invariant, we have a solution that, in addition to being valid, is also a vertex of the bounding polyhedral. More formally, the solution satisfies all of the constraints and meets n of the constraints with equality. A step will involve climbing along the edge (one-dimensional line) between two adjacent vertices. This involves *relaxing* one of the constraints that is met with equality, so that it no longer is met with equality, and *tightening* one of the constraints that was not met with equality so that it now is met with equality. This is called *pivoting* out one equation and in another. The new solution will be the unique solution that satisfies with equality the n currently selected equations. Of course, at each iteration such a step can be taken only if it continues to satisfy all of the constraints and improves the objective function. There are fast ways of finding a good step to take. However, even if you do not know these, there are only $n \cdot m$ choices of steps to try, when there are n variables and m equations.

Finding an Initial Valid Solution: If we are lucky, the origin will be a valid solution. However, in general finding some valid solution is itself a challenging problem. Our algorithm to do so will be an iterative algorithm that includes the constraints one at a time. Suppose we have a vertex solution that satisfies all of the constraints in [Example 21.4.1](#) except the last one. We will then treat the negative of this next constraint as the objective function, namely, $-4x_1 + 8x_2 + 4x_3 - 4x_4$. We will run our hill-climbing algorithm, starting with the vertex we have, until we have a vertex solution that maximizes this new objective function subject to the first i equations. This is equivalent to minimizing the objective $4x_1 - 8x_2 - 4x_3 + 4x_4$. If its minimum is less than 685, then we have found a vertex solution that satisfies the first $i + 1$ equations. If not, then we have determined that no such solution exists.

No Small Local Maximum: To prove that the algorithm eventually finds a global maximum, we must prove that it will not get stuck in a small local maximum.

Convex: Because the bounded polyhedral is the intersection of straight cuts, it is what we call *convex*. More formally, this means that the line between any two points in the polyhedral is also in the polyhedral. This means that there cannot be two local maximum points, because between these two hills there would need to be a valley, and a line between two points across this valley would be outside the polyhedral.

The Primal–Dual Method: The primal–dual method formally proves that a global maximum will be found. Given any linear program, defined by an optimization function and a set of constraints, there is a way of forming its *dual* minimization linear program. Each solution to this dual acts as a roof or upper bound on how high the primal solution can be. Then each iteration either finds a better solution for the primal or provides a solution for the dual linear program with a matching value. This dual solution witnesses the fact that no primal solution is bigger.

Forming the Dual: If the primal linear program is to maximize $a \cdot x$ subject to $Mx \leq b$, then the dual is to minimize $b^T \cdot y$ subject to $M^T \cdot y \geq a^T$, where b^T , M^T , and a^T are the transposes formed by interchanging rows and columns. The dual of [Example 21.4.1](#) is

$$\begin{aligned} &\text{minimize} \\ &258 + 721y_2 + 524y_3 + 411y_4 + 685y_5 \\ &\text{subject to} \\ &3y_1 + 6y_2 + 2y_3 + 3y_4 + 4y_5 \geq 7 \\ &7y_1 + 3y_2 + 1y_3 + 6y_4 - 8y_5 \geq -6 \\ &2y_1 + 9y_2 + 5y_3 + 2y_4 - 4y_5 \geq 5 \\ &9y_1 - 6y_2 + 5y_3 + 3y_4 + 4y_5 \geq 7. \end{aligned}$$

The dual will have a variable for each constraint in the primal and a constraint for each of its variables. The coefficients of the objective function become the numbers on the right-hand sides of the inequalities, and the numbers on the right-hand sides of the inequalities become the coefficients of the objective function. Finally, “maximize” becomes “minimize.” The dual is the same as the original primal.

Upper Bound: We prove that the value of any solution to the primal linear program is at most the value of any solution to the dual linear program. The value of the primal solution x is $a \cdot x$. The constraints $M^T \cdot y \geq a^T$ can be turned around to give $a \leq y^T M$. This gives that $a \cdot x \leq y^T \cdot M \cdot x$. Using the constraints $Mx \leq b$, this is at most $y^T \cdot b$. This can be turned around to give $b^T \cdot y$, which is the value of the dual solution y .

Running Time: The primal–dual hill-climbing algorithm is guaranteed to find the optimal solution. In practice, it works quickly (though for my summer job, the computers would crank for hours). However, there is no known hill-climbing algorithm that is guaranteed to run in polynomial time.

There is another algorithm that solves this problem, called the *ellipsoid method*. Practically, it is not as fast, but theoretically it probably runs in polynomial time.

Exercise 21.4.1 Express the network flow instance in [Figure 21.2](#) as a linear program.

21.5 Exercises

Exercise 21.5.1 (See solution in Exercise Solutions.) Let $G = (L \cup R, E)$ be a bipartite graph with nodes L on the left and R on the right. A matching is a subset of the edges such that each node appears at most once. For any $A \subseteq L$, let $N(A)$ be the neighborhood set of A , namely, $N(A) = \{v \in R \mid \exists u \in A \text{ such that } (u, v) \in E\}$. Prove Hall's theorem, which states that there exists a matching in which every node in L is matched if and only if $\forall A \subseteq L, |A| \leq |N(A)|$.

1. For each of the two bipartite graphs in Figure 21.10, give a short witness either to the fact that it has a perfect matching or to the fact that it does not. Use Hall's theorem in your explanation why a graph does not have a matching. No need to mention flows or cuts.

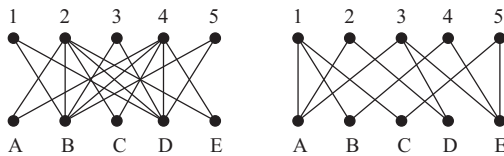


Fig. 21.10 Bipartite graphs for Exercise 21.5.1.

2. $\Rightarrow$: Suppose there exists a matching in which every node in L is matched. For $u \in L$, let $M(u) \in R$ specify one such matching. Prove that $\forall A \subseteq L, |A| \leq |N(A)|$.
3. Section 28.4 describes a network with nodes $\{s\} \cup L \cup R \cup \{t\}$ with a directed edge from s to each node in L , the edges E from L to R in the bipartite graph directed from L to R , and a directed edge from each node in R to t . The nodes give each edge capacity 1. However, the edges $\langle u, u \rangle$ across the bipartite graph could just as well be given capacity ∞ . Consider some cut (U, V) in this network. Note that U contains s , some nodes of L , and some nodes of R , whereas V contains the remaining nodes of L , the remaining nodes of R , and t . Assume that $\forall A \subseteq L, |A| \leq |N(A)|$. Prove that the capacity of this cut, i.e., $\text{cap}(U, V) = \sum_{u \in U} \sum_{v \in V} c_{\langle u, v \rangle}$, is at least $|L|$.
4. $\Leftarrow$: Assume that $\forall A \subseteq L, |A| \leq |N(A)|$ is true. Prove that there exists a matching in which every node in L is matched. (Hint: Use everything you know about network flows.)
5. Suppose that there is some integer $k \geq 1$ such that every node in L has degree at least k and every node in R has degree at most k . Prove that there exists a matching in which every node in L is matched.

22 Greedy Algorithms

“Every two-year-old knows the greedy algorithm. In order to get what you want, just start grabbing what looks best.”

22.1 Abstractions, Techniques, and Theory

Specifications: A very select number of optimization problems can be solved using a greedy algorithm. Most of these have the following form.

Precondition: We are given one of the following instances.

Instances: An instance consists of a set of objects and a relationship between them. Think of the objects as being prizes that you must choose among.

Postcondition: Given an instance, the goal is to find one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) measure of success. (The solution to be outputted need not be unique.)

Solution for Instance: A solution S requires the algorithm to make a choice about each of the objects in the instance. Sometimes, this choice is complex, but usually it is simply whether or not to keep it. In this case, a solution is the subset of the objects that you have kept.

Valid Solution: The problem specifies some criteria according to which a solution S is or is not valid. For example, some subsets might not be allowed because the objects in it “conflict” somehow with each other.

Measure of Success: The problem statement assigns each solution S a value, cost, or measure of success $Val(S)$. For example, this value $Val(S) = \sum_{obj \in S} Val(obj)$ might simply be the sum of the values of its the objects taken.

Goal: Given an instance I , the goal is to find one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) measure of success. The solution to be outputted need not be unique.

The Brute Force Algorithm (Exponential Time): The brute force algorithm for an optimization problem considers each possible solution for the given instance, computes

its cost, and outputs the cheapest. Because each instance has an exponential number of solutions, this algorithm takes exponential time.

The Greedy Choice: The greedy step is the first that would come to mind when designing an algorithm for a problem. Given the set of objects specified in the input instance, the greedy step chooses and makes an irrevocable decision about one of these objects because, according to some simple criterion, it seems to be the best. When proving that the algorithm works, we must be able to prove that this locally greedy choice does not have negative global consequences.

Example 22.1.1 (The Game Show)

Suppose the instance specifies a set of prizes and an integer m and allows you to choose m of the prizes. The criterion, according to which some of these prizes appear to be better than others, may be its dollar price, the amount of joy it would bring you, how practical it is, or how much it would impress the neighbors. At first it seems obvious that you should choose your first prize using the greedy approach. However, some of these prizes conflict with each other, and (as is often the case in life) compromises need to be made. For example, if you take the pool, then your yard is too full to be able to take many of the other prizes. A lion might impress your neighbors, but it might eat your dog. As is also true in life, it is sometimes hard to look into the future and predict the ramifications of the choices made today.

Example 22.1.2 (Making Change)

The goal of this optimization problem is to find the minimum number of quarters, dimes, nickels, and pennies that total to a given amount. An instance consists of a set of objects and a relationship between them. Here, the set is a huge pile of coins, and the relationship is that the chosen coins must total to the given amount. The cost of a solution, which is to be minimized, is the number of coins in the solution.

The Greedy Choice: The coin that appears to be best to take is a quarter, because it makes the most progress toward making our required amount while only incurring a cost of one.

A Valid Choice: Before committing to a quarter, we must make sure that it does not conflict with the possibility of arriving at a valid solution. If the sum to be obtained happens to be less than \$0.25, then this quarter should be rejected, even though it appears at first to be best. On the other hand, if the amount is at least \$0.25, then we can commit to the quarter without invalidating the solution we are building.

Leading to an Optimal Solution: A much more difficult and subtle question is whether or not committing to a quarter leads us toward obtaining an optimal solution. In this case, it happens that it does, though this not at all obvious.

Going Wrong: Suppose that the problem is generalized to include as part of the input the set of coin denominations available. Then greedy algorithm does not work. For example, suppose we have 4-, 3-, and 1-cent coins. If the given amount is 6, then the optimal solution contains two 3-cent coins. We go wrong by greedily committing to a 4-cent coin.

Proof of Correctness Using Loop Invariants: Committing to the pool, to the lion, or to the 4-cent coin, though they locally appear to be the best objects, does not lead to an optimal solution. However, for some problems and for some definitions of “best,” the greedy algorithm does work. When it does not work, we will prove it using one simple counterexample as we did above. When it does work, we will prove it using the same method with which we prove all iterative algorithms correct: loop invariants.

Main Steps: Each iteration the algorithm chooses the best object from among those not considered so far, and either commits to it or rejects it.

Make Progress: One more object has been considered.

Exit Condition: All objects have been considered.

Loop Invariant: Recall that a loop invariant makes a statement each time the algorithm is at the top of the loop about what it has accomplished. So far the algorithm has made one iteration, during which it has committed to one of the objects. What statement do we want to make about this action?

Types of Loop Invariants: In order to find a good loop invariant, let us start by considering the types of loop invariants considered in [Section 1.4](#). It turns out that our chosen loop invariant will be of the narrowing-the-search-space type.

More of the Output: Recall that the more-of-the-output loop invariant states that the first i items of the output have been produced and the output constructed so far is correct. In the case of greedy algorithms, the output is a decision about each object, and the algorithm makes another decision each iteration. Hence, the first part of the loop invariant is easily maintained. However, what does it even mean for the output constructed so far to be correct?

No Conflict Yet: If our goal is to obtain a solution that has no conflicts, a good loop invariant might be that within the commitments made so far, there are no conflicts. The problem with this is that the commitments made so far may have backed the algorithm into a corner so that future conflicts are inevitable. See [Exercise 22.1.3](#).

The Postcondition as a Loop Invariant: The postcondition is that the algorithm has committed to an optimal solution. However, at this point we cannot talk about the algorithm’s “solution,” because before the algorithm completes it may have made some commitments already, but this partial work does not yet constitute a valid solution.

Optimal So Far: The loop invariant might be “What the algorithm has done so far is optimal.” However, being optimal is a property of full solutions, and we don’t have one yet.

More of the Input: Recall that the more-of-the-input loop invariant states that if we pretend that the prefix of the input read so far by the algorithm is the entire input, then the algorithm has an optimal solution. Though this is tempting, it does

not work. That what you have is optimal within this part of the instance is only a *local* property. It won't necessarily guarantee global optimality. For example, if the input of the game show problem consisted of only a lion, then the optimal solution would be to take the lion. However, when there are also more objects, the lion is not part of the solution. Similarly, in the make change problem with 4-, 3-, and 1-cent coins, if the input is to make the amount 4, then one should take a 4; but not if the input is to make the amount 6.

Narrowing the Search Space: Recall the narrowing-the-search-space type of loop invariant. While searching for something, the algorithm narrows the search space. The loop invariant is "If the thing being searched for is anywhere, then it is in this narrowed search space." Greedy algorithms are in fact searching for an optimal solution from the search space of all possible solutions. Every time the algorithm commits to something, the set of possible solutions that it might still output narrows to those that are consistent with the decisions made so far. A natural loop invariant would then be "If the optimal solution being searched for is anywhere, then it is consistent with the decisions made so far."

"The" Optimal Solution Contains the "Best" Object: Before committing to the seemingly best object or making the first decision, we need to prove that we do not go wrong by doing so. As a first attempt, we might try to prove that for every set of objects that might be given as an instance, the "best" of these objects is definitely in its optimal solution. The problem with this is that there may be more than one optimal solution. It might not be the case that all of them contain the chosen object.

At Least One Optimal Solution Remaining: Instead of requiring all optimal solutions to contain the "best" object, what we need to prove is that at least one does. The effect of this is that though committing to the "best" object may eliminate the possibility of some of the optimal solutions, it does not eliminate all of them. There is the saying, "Do not burn your bridges behind you." The message here is slightly different. It is OK to burn a few of your bridges as long as you do not burn all of them.

If There Is an Optimal Solution: For most problems, it is clear that an instance has at least one solution (maybe a trivial solution) and hence there is a best solution. Hence, this first conditional part of the loop invariant is usually dropped. Be aware, however, that it is sometimes needed. See [Exercise 22.1.3](#).

The Loop Invariant: To recap, the loop invariant is that we have not gone wrong. There is at least one valid optimal solution S_t consistent with the choices A_t made by the algorithm during the first t time steps.

Consistent: We say that solution S_t is *consistent* with A_t (or *extends*) every choice made by A_t is made in the same way by S_t . Being a full solution, S_t makes choices about remaining objects as well. We say the algorithm has not gone

wrong because though it has made these irrevocable decisions, it still has the possibility of producing this solution.

The Second Step: After the “best” object has been chosen and committed to, the algorithm must choose the next objects in the input and make an irrevocable decision about it. You can think about this process within either the iterative or the recursive paradigm. Though the resulting algorithm is (usually) the same, having the different paradigms at your disposal can be helpful.

Iterative: In the iterative version, there is a main loop. At each iteration, you choose the best from among the objects that have not yet been considered. The algorithm then commits to some choice about this object. Usually, this involves deciding whether to commit to putting this chosen object in the solution or to commit to rejecting it.

Greedy Criteria: The algorithm specifies a *criteria* with which it chooses which object to make a decision about next. In order to keep the algorithm fast, it is not allowed to choose based on some complicated process that depends on the entire input. Instead, the algorithm defines a function assigning a real number to every object that might ever exist in the universe indicating how much the algorithm “likes” the object. The algorithm simply sorts the unconsidered objects in the input according to this measure and considers the best (or the worst). For example, the making change algorithm assigns value 25 to each quarter, 10 to each dime, 5 to each nickel, and 1 to each penny and takes the unconsidered object with the largest value.

Decision-Making Process: After using the greedy criteria to choose which object to consider next, the algorithm must make an irrevocable decision about it. If the decision to be made is whether or not to keep the object, usually the answer is yes, unless this new object “conflicts” with the objects committed to previously or because it fills no requirements that are not already filled by the objects already committed to.

Cannot Predict the Future: At each step, the choice that is made can depend on the choices that were made in the past, but it cannot depend on the objects in the input that have not been considered yet.

Irrevocable Decision: Making this decision can be a little scary because once made, it can never be changed. In contrast, the process of an algorithm making and later going back and changing a decision is called *backtracking*. Backtracking algorithms tend to require exponential time, while greedy algorithms require linear time.

Making Change Example: The greedy algorithm for finding the minimum number of coins summing to a given amount is as follows. Commit to quarters until the next quarter increases your current sum above the required amount. Then reject the remaining quarters. Then do the same with the dimes, the nickels, and the pennies.

Recursive: A recursive greedy algorithm makes a greedy first choice and then recurses once or twice in order to solve the remaining subinstance.

Making Change Example: After committing to a quarter, we could subtract \$0.25 from the required amount and ask a friend to find the minimum number of coins to make this new amount. Our solution will be his solution plus our original quarter.

Binary Search Tree Example: The recursive version of a greedy algorithm is more useful when you need to recurse more than once. For example, suppose you want to construct a binary search tree for a set of keys that minimizes the total height of the tree, i.e., a balanced tree. The greedy algorithm will commit to the middle key being at the root. Then it will recurse once for the left subtree and once for the right.

To learn more about how to recurse after the greedy choice has been made, see the recursive backtracking algorithms in [Chapter 23](#).

Proof of Correctness: Greedy algorithms themselves are very easy to understand and to code. If your intuition is that they should not work, then your intuition is correct. For most optimization search problems, no greedy algorithms that are tried work. By some miracle, however, for some problems there is a greedy algorithm that works. The proof that they work, however, is very subtle and difficult. As with all iterative algorithms, we prove that they work using loop invariants.

A Formal Proof: [Chapter 1](#) proves that an iterative algorithm works if the loop invariant can be established and maintained and from it the postcondition can be proved. The loop invariant here is that the algorithm has not gone wrong: there is at least one optimal solution consistent with the choices made so far. This is established ($\langle pre \rangle \Rightarrow \langle LI \rangle$) by noting that initially no choices have been made and hence all optimal solutions are consistent with these choices. The loop invariant is maintained $\langle LI_{t-1} \rangle \ \&not\ \langle exit \rangle \ \&code_{loop} \implies \langle LI_t \rangle$ as follows. If it is true when at the top of the loop, then let S_{t-1} denote one such solution. $code_{loop}$ during the next iteration either commits to or rejects the next best object. The proof describes a method for modifying S_{t-1} into S_t and proves that this is a valid solution, is consistent both with the choices made previously by the algorithm and with this new choice, and is optimal. The existence of such a S_t proves that the loop invariant has been maintained. The last step is to prove that in the end, the algorithm has a concrete optimal solution ($\langle LI \rangle \ \& \ \langle exit \rangle \Rightarrow \langle post \rangle$). Progress is made at each step by committing to or rejecting another object. When each object has been considered, the algorithm exits. These choices specify a solution. The loop invariant states there is an optimal solution consistent with these choices. Hence, the solution obtained must be optimal.

We will now redo the proof in a more intuitive, fun, and detailed way.

The Loop Invariant: The loop invariant maintained is that we have not gone wrong. There is at least one optimal solution consistent with the choices made

so far, that is, containing the objects committed to so far and not containing the objects rejected so far.

Three Players: To help understand this proof, we will tell a story involving three characters: the algorithm, the prover, and a fairy godmother.

The Algorithm: At each iteration, the algorithm chooses the best object from amongst those not considered so far and either commits to it or rejects it. Let A_t denote every decision the algorithm has made up until time t .

The Prover: The prover's task is to prove that the loop invariant is maintained. Having a separate prover emphasizes that fact that his actions are not a part of the algorithm and hence do not need to be coded or executed.

The Fairy Godmother: The prover does not need to know the optimal solution S_t . He can pretend that a fairy godmother holds it witnessing the fact that such a solution exists.

Initially ($\langle pre \rangle \Rightarrow \langle LI \rangle$): Assuming that there is at least one valid solution, there is an optimal one S_0 . It is consistent with A_0 because at the beginning the algorithm has done nothing. If there are more than one equally good optimal solutions, then the fairy godmother chooses one arbitrarily.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle$ & not $\langle exit \rangle$ & $code_{loop} \Rightarrow \langle LI_t \rangle$): Now consider an arbitrary iteration.

What We Know: From the loop invariant $\langle LI_{t-1} \rangle$, the prover knows that there is at least one valid optimal solution S_{t-1} held by the fairy godmother consistent with the choices A_{t-1} made by the algorithm during the first $t-1$ time steps.

Taking a Step: At time t , the algorithm chooses the best object from amongst those not considered so far and either commits to it or rejects it. Let A_t extend A_{t-1} with this new choice.

Instructions for Modifying: If the prover does not know the solution S_{t-1} , but does know that the fairy godmother is holding it, he can give her detailed instructions on how to modify it into S_t so as to witness the fact that such a solution exists. Instead of S_{t-1} and S_t , the first version of the book used the notations $optS_{LI}$ and $optS_{ours}$ emphasizing that the first is stated to exist by the loop invariant and the second is the one created by the prover. However, people got confused, thinking that $optS_{ours}$ is the solution created by the algorithm.

Making Change Example: If the remaining amount required is at least \$0.25, then the algorithm commits to another quarter. Excluding the committed-to coins, the fairy godmother's optimal solution S_{t-1} must be making up the same remaining amount. This amount must contain either an additional quarter, three dimes, two dimes and a nickel, one dime and three nickels, five nickels, or combinations

with at least five pennies. The prover tells her to replace the three dimes with the newly committed-to quarter and nickel, and the other options with just the quarter. If the algorithm, on the other hand, rejects the next (and later all remaining) quarters because the remaining amount required is less than \$0.25, then the prover is confident that optimal solution held by his fairy godmother cannot contain additional quarters either.

Many Cases: The loop invariant states that the fairy godmother's solution S_{t-1} is optimal. However, the prover does need to consider every possible case of what S_{t-1} might be. To be polite to the fairy godmother, let's not judge whether or not her solution is optimal. Instead, let's just prove that our new solution S_t is at least as good as hers, i.e., $Val(S_t) \geq Val(S_{t-1})$.

Tree of Options: When there are many, one ensures that all the cases have been considered by forming a decision tree. Each node of the tree is labeled by a question about the current case. Each outgoing edge is labeled by an answer to this question. Each leaf node is labeled by a proof of how to handle all the cases that would follow the path from the root to this leaf. See Exercise III.6.

Head in the Right Direction & Return to Path: It is sometimes hard to know how to change S_{t-1} into S_t so that it is valid, consistent, and optimal. One option is to do this in two steps.

Make S_t Consistent: If the decision the fairy godmother in S_{t-1} has made about Obj_t is different from the algorithm made in A_t , then the first modification the prover must tell the fairy godmother is to change this decision, namely, $S'_t = S_{t-1} + (A_t$'s decision made about Obj_t). For example, if the algorithm committed to adding Obj_t and she did not, then get her to add it.

Fix Problems: We know that S_{t-1} is a valid optimal solution as it is. Changing the decision made about Obj_t most likely either makes it no longer valid or no longer optimal. In order to "return to the path," define S_t to be the new solution $S_t = S_{t-1} + (A_t$'s decision made about Obj_t) + (the prover's new decision about Obj'). Here Obj' is some object that you must find such that changing the decision about it made by the fairy godmother returns solution S_t is being valid and optimal again. Note that Obj' cannot be an object for which the algorithm A_{t-1} has already made a decision or else this will make S_t inconsistent with A_t . Also when describing Obj' do not talk about the "next" decision that S_{t-1} makes, because S_{t-1} does not make the decision outside of A_{t-1} in a particular order. Maybe the decision about more than one object needs to be changed.

Making Change Example: For example, if the algorithm's next step is adding a quarter, then Obj_t is that quarter. $S'_t = S_{t-1} + 25\text{¢}$ means that S'_t is no longer valid because to be valid, the taken coins must sum to the stated amount and now it sums to 25 cents too much. It is also no longer optimal because the measure of success is minimizing the number of coins taken and S'_t takes one more.

One of the many cases defined is that the fairy godmother's solution has taken two dimes and a nickel within her solution S_{t-1} and the prover tells her to replace them with the algorithm's new quarter, i.e., $S_t = S_{t-1} + 25\text{¢} - (2 \cdot 10\text{¢} + 5\text{¢})$. This is valid because it adds up to the correct amount again and is better than her solution because it went up by one coin and down by three.

Proving That She Has a Witness: It is the job of the prover to prove that the thing S_t the fairy godmother now holds is a valid, consistent, and optimal solution.

Proving a Valid Solution: The prover knows that S_{t-1} was valid. (Reread the definition of the problem to recall what that means.) They just have to prove that their modifications did not introduce invalid things, proving that S_t is still valid.

Proving Consistency: The prover know that S_{t-1} was consistent with A_{t-1} . They only need to prove that they modified it to be consistent with the new choices without messing up earlier ones, hence proving that S_t is consistent with A_t .

Proving Optimal: (Reread the definition of the value $val(S)$ of a solution to recall what that means). Though the prover knows from the loop invariant that S_{t-1} was optimal, let's not use this fact in case focusing on this makes us ignore possible cases that need to be covered. The prover only needs to prove that his modifications did not make it worse, i.e., $Val(S_t) \geq Val(S_{t-1})$.

This completes the prover's proof that the fairy godmother now has an optimal solution consistent both with the previous choices and with the latest choice. This witnesses the fact that such a solution exists.

This proves that the loop invariant has been maintained.

Continuing: This completes everybody's requirements for this iteration. The process is repeated over and over again. Each iteration, the algorithm commits to more about the solution, and the fairy godmother's solution is changed to be consistent with these commitments.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \Rightarrow \langle post \rangle$): After the algorithm has considered every object in the instance and each has either been committed to or rejected,

the algorithm exits. We still know that the loop invariant is true. Hence, the prover knows that there is an optimal schedule S_T consistent with all of these choices. Previously, this optimal solution was only imagined. However, now we concretely know that this imagined solution consists of those objects committed to. Hence, the algorithm can return this set as the solution.

Running Time: Greedy algorithms are very fast, because they take only a small amount of time per object in the instance.

Alternative Proof: In case it's useful, we will now provide an alternative proof that defines a *globally optimal* algorithmic step in contrast to a *locally optimal* one and will use first order logic to define the *greedy game*.

Definition of Correct: We need to prove that for every input I , our greedy algorithm finds an optimal solution S_A for this instances, namely, S_A is at least as good as every other valid solution S for input I .

***potentialVal*(A_t):** Though the decisions A_t made by the algorithm may or may not be good, they cannot be changed. Define *potentialVal*(A_t) to be the maximum value of a full solution S that can be made while remaining consistent to A_t , namely, $\text{potentialVal}(A_t) = \max_S$ is a full valid solution consistent with A_t *Val*(S).

Locally Optimal Step: An algorithm's action is said be *locally optimal* if it seems to increase as much as possible the value of the solution being produced.

Globally Optimal Step: An algorithm's action is said to be *globally optimal* if it maintains the potential value, namely, $\text{potentialVal}(A_t) \geq \text{potentialVal}(A_{t-1})$.

Does Not Go Down: Suppose when you were born, you had the potential to become a doctor. But the night before the medical exam, you partied with your friends and failed the test. Being fun, this was the locally optimal choice. If because of it you ended up a bum, then it was not globally optimal. On the other hand, going to the party would be globally optimal if doing so did not decrease your potential. Maybe you could rewrite the exam. Or maybe, even though you can no longer be a doctor, it did not burn the bridge to being a lawyer. Before the party, the fairy godmother is holding the solution S_{t-1} in which you are a doctor. After the party, the prover convinces her to switch to a solution S_t in which you are a lawyer. These solutions have the same value. Hence, $\text{potentialVal}(A_t) = \text{potentialVal}(A_{t-1})$.

Does Not Go Up: Suppose at a party you meet somebody (your partner, boss, co-author to be) who changes your life for the better. One might think that the decision to go to this party increased your *potentialVal*. However, it does not. Recall that $\text{potentialVal}(A_{t-1})$ is defined to be the best you could possibly do in your life given you have reached the moment in your life just before the party. Note that this maximization includes the possibility of you going to the party. Hence, it is impossible that $\text{potentialVal}(A_t) > \text{potentialVal}(A_{t-1})$.

Theorem: If each step taken by the algorithm is globally optimal, then the algorithm returns an optimal solution, i.e., $Val(S_A) = \max_S Val(S)$.

Proof: Before the algorithm has done any action, every valid solution is consistent with what the algorithm has done so far. Hence, $\max_S Val(S) = potentialVal(A_0)$. Because all steps are globally optimal, $potentialVal(A_0) = potentialVal(A_1) = potentialVal(A_2) = \dots = potentialVal(A_T)$. After the algorithm has produced a solution S_A , this is the only solution consistent. Hence, $potentialVal(A_T) = Val(S_A)$.

Game as Proof: The game we described between the algorithm, prover, and the fairy godmother proves that the algorithm works.

Greedy Game: The game begins by an adversary choosing input I , the algorithm choosing its first t actions A_t , and the fairy godmother choosing a valid solution S_{t-1} consistent with the first $t-1$ of these. Assume that these choices were adversarial, i.e., worst case. The prover then tells the fairy godmother to modify S_{t-1} into S_t . Then the prover proves that this is valid, extends A_t , and its value has not decreased.

First Order Logic: This game corresponds to the following statement:
 $\forall$ inputs I , $\forall$ choices A_t , $\forall$ solutions S_{t-1} extending A_{t-1} , $\exists$ a solution S_t extending A_t , such that $Val(S_t) \geq Val(S_{t-1})$.

Game: Section Logic teaches that the validity of a first order logic statement is proved by having a strategy for the corresponding game. See Proof by Adversarial Game in [Chapter 34](#).

Proving: We prove that a statement is true for every value of x by allowing some *adversary* to choose x 's value and then proving the statement for this value. We prove that there exists a value y for which a statement is true, by the *prover* constructing such a value for y , and then proving that it is true for that value.

Using: Conversely, when you know a statement is true, you know that an *oracle* can win the game for the prover. If the oracle assures them that it is true for all x , then the prover can specify their favorite value for x and the oracle will assure them that it is true for this value. To assure that it is true some y , the oracle must construct a value for y for which it is true.

Proof Globally Optimal: This game proves that each step made by the algorithm is globally optimal, namely, $potentialVal(A_t) \geq potentialVal(A_{t-1})$. In our game, we can assume that the fairy godmother is holding the best solution S_{t-1} possible so that $potentialVal(A_{t-1}) = Val(S_{t-1})$. The oracle provides a S_t of the same value. This witnesses that such a value is obtainable, namely, $Val(S_{t-1}) = Val(S_t) \leq \max_S Val(S)$ is a full valid solution consistent with A_t , $Val(S) = potentialVal(A_t)$.

Fixed vs. Adaptive Priority: Iterative greedy algorithms come in two flavors, *fixed priority* and *adaptive priority*.

Fixed Priority: A fixed-priority greedy algorithm begins by sorting the objects in the input instance from best to worst according to a fixed greedy criterion. For example, it might sort the objects based on the cost of the object or the arrival time of the object. The algorithm then considers the objects one at a time in this order.

Adaptive Priority: In an adaptive priority greedy algorithm, the greedy criterion is not fixed, but depends on which objects have been committed to so far. At each step, the next-best object is chosen according to the current greedy criterion. Blindly searching the remaining list of objects each iteration for the next-best object would be too time-consuming. So would re-sorting the objects each iteration according to the new greedy criterion. A more efficient implementation uses a priority queue to hold the remaining objects prioritized according to the current greedy criteria. This can be implemented using a heap. (See [Section 13.3](#).)

Code

algorithm *AdaptiveGreedy*(set of objects)

$\langle$ *pre-cond* $\rangle$: The input consists of a set of objects.

$\langle$ *post-cond* $\rangle$: The output consists of an optimal subset of them.

begin

 Put objects in a priority queue according to the initial greedy criterion

$Commit = \emptyset$ % set of objects previously committed to

 loop

$\langle$ *loop-invariant* $\rangle$: There is at least one optimal solution
consistent with the choices made so far.

 Exit when the priority queue is empty

 Remove “best” object from priority queue.

 If this object does not conflict with those in $Commit$ and is
needed, then

 Add object to $Commit$

 end if

 Update the priority queue to reflect the new greedy criterion

 This is done by changing the priorities of the objects effected.

 end loop

 return ($Commit$)

end algorithm

Example: Dijkstra’s shortest-weighted-path algorithm ([Section 20.3](#)) can be considered to be a greedy algorithm with an adaptive priority criteria. It chooses the next edge to include in the optimal shortest-weighted-path tree based on which node currently seems to be the closest to s . Those yet to be

chosen are organized in a priority queue. Even breadth-first and depth-first searches can be considered to be adaptive greedy algorithms. In fact, they very closely resemble Prim's minimum-spanning-tree algorithm (Section 22.2.3), in how a tree is grown from a source node. They are adaptive in that as the algorithm proceeds, the set from which the next edge is chosen changes.

Exercise 22.1.1 We proved that the greedy algorithm does not work for the making change problem when the denominations of the coins are 4, 3, and 1 cent, but it does work when the denominations are 25, 10, 5, and 1. Does it work when the denominations are 25, 10, and 1, with no nickels?

Exercise 22.1.2 Suppose the coin denominations are $c_1 > c_2 > \dots > c_r$ in the order taken by the greedy algorithm. An interesting problem is determining whether the greedy algorithm works. A complete answer to this question is too hard. However, what restrictions on the coin denominations are sufficient to ensure that the greedy algorithm works?

- Suppose for each i , each coin c_i is an integer multiple of the next smaller c_{i+1} , e.g., 120, 60, 12, 3, 1. If this is true, do we know that the greedy algorithm works? If it is not true, do we know that the greedy algorithm does not work?
- Suppose, for each i , each coin is more than twice the previous, that is, $c_i \geq 2c_{i+1}$. Do we know that the greedy algorithm works? If this is not true, do we know that the greedy algorithm does not work?
- Are there other interesting characteristics?

Exercise 22.1.3 The 2-coloring problem is as follows: Given any undirected n -node graph, color the nodes with two colors so that no edge connects two nodes of the same color (i.e., adjacent nodes always have different colors), or report that it is impossible.

1. A tempting loop invariant is "There are no adjacent nodes of the same color in my partial solution." In order for this to be a good loop invariant, one needs to be able to arrive from Mars, knowing nothing about what the algorithm has done so far except that the loop invariant is true. From here that algorithm must be able to continue until the postcondition is obtained. Prove that this is a bad loop invariant as follows:
 - (a) Provide a graph that has a valid 2-coloring.
 - (b) Provide a valid partial coloring that a (faulty) algorithm may provide for which the loop invariant holds.
 - (c) Prove that the algorithm has gone wrong because there are no valid colorings consistent with this partial solution.
2. This is an example in which an instance might not have any valid solution. One possible algorithmic technique is to check the validity of the solution after all the nodes have been colored. Another is to check as it proceeds, to make sure that the coloring being created has no mistakes. (Which technique is used in the binary search algorithm?) For each of these two techniques, give a loop invariant and then prove $\langle pre-cond \rangle \ \& \ code_{alg} \Rightarrow \langle post-cond \rangle$.

3. Briefly describe a greedy algorithm. The algorithm should be able to handle graphs that are not connected.
4. Is the greedy criterion used adaptive or nonadaptive? Does it need to be?
5. Prove that the loop invariant is maintained. ($\langle \text{loop-invariant}_{t-1} \rangle \ \& \ \text{not}(\text{exit-cond}) \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invariant}_t \rangle$). Be sure to handle boundary conditions, e.g., the first iteration, and graphs that are not connected.
6. What is the running time of your algorithm?

22.2 Examples of Greedy Algorithms

22.2.1 Example: The Job/Event Scheduling Problem

Suppose that many people want to use your conference room for events and you must schedule as many of these as possible. (The version in which some events are given a higher priority is considered in [Section 24.6](#).) We examine a fixed-priority greedy algorithm.

Specifications:

Precondition: We are given one of the following instances.

Instances: An instance is $\langle \langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \dots, \langle s_n, f_n \rangle \rangle$, where $0 \leq s_i \leq f_i$ are the starting and finishing times for the i th event.

Postcondition: The output is a solution with maximum number of events scheduled.

Solutions: A solution for an instance is a schedule S . This consists of a subset $S \subseteq [1 \dots n]$ of the events that don't conflict by overlapping in time.

Measure of Success: The success of a solution S is the number of events scheduled, that is, $|S|$.

Possible Criteria for Defining “Best”

The Shortest Event $f_i - s_i$: It seems that it would be best to schedule short events first, because they increase the number of events scheduled without booking the room for long period of time. This greedy approach does not work.

Counterexample: Suppose that the following lines indicate the starting and completing times of three events to schedule.



We would be going wrong to schedule the short event in the middle, because the only optimal schedule does not include it.

The Earliest Starting Time s_i or the Latest Finishing Time f_i : First come first served, which is a common scheduling algorithm, does not work either.



Counterexample

The long event is both the earliest and the latest. Committing to scheduling it would be a mistake.

Event Conflicting with the Fewest Other Events: Scheduling an event that conflicts with other events prevents you from scheduling these events. Hence a reasonable criterion would be to first schedule the event with the fewest conflicts.

Counterexample: In the following example, the middle event would be committed to first. This eliminates the possibility of scheduling four events.

Earliest Finishing Time f_i : This criterion may seem a little odd at first, but it makes sense. It says to schedule the event that will free up your room for someone else as soon as possible. We will see that this criterion works for every set of events.

Example: You can trace out this algorithm on the example shown in [Figure 22.1](#).

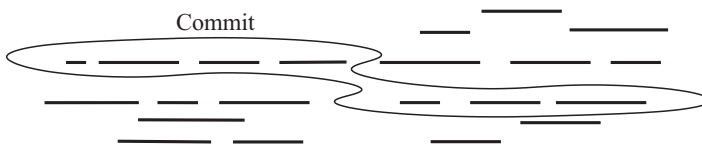


Fig. 22.1 A set of events and those committed to by the earliest-finishing-time-first Greedy algorithm.

Code: A greedy algorithm for the event scheduling problem.

algorithm *Scheduling*($\langle\langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \dots, \langle s_n, f_n \rangle\rangle$)

<pre-cond> The input consists of a set of events.

<post-cond> The output consists of a schedule that maximizes the number of events scheduled.

begin

Sort the events based on their finishing times f_i

$Commit = \emptyset$ % The set of events committed to be in the schedule

loop $i = 1 \dots n$ % Consider the events in sorted order.

 if(event i does not conflict with an event in $Commit$) then

$Commit = Commit \cup \{i\}$

 end loop

 return ($Commit$)

end algorithm

The Loop Invariant: The loop invariant is that we have not gone wrong: There is at least one optimal solution consistent with the choices made so far, that is, containing the objects committed to so far and not containing the objects rejected so far.

Initial Code ($\langle pre \rangle \Rightarrow \langle LI \rangle$): Initially, no choices have been made, and hence trivially all optimal solutions are consistent with these choices.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle \ \& \ \text{not} \ \langle exit \rangle \ \& \ code_{loop} \Rightarrow \langle LI_t \rangle$): We are at the top of the loop of the algorithm. Consider the example in Figure 22.2.

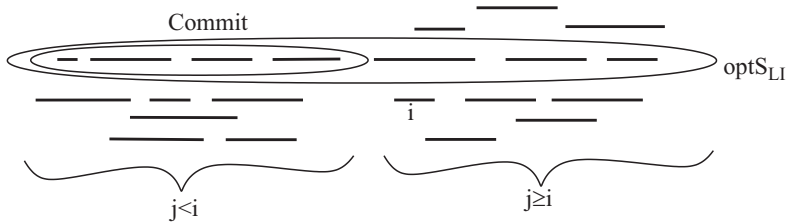


Fig. 22.2 A set of events, those committed to at the current point in time, those rejected, those in the optimal solution assumed to exist, and the next event to be considered.

Hypothetical Optimal Solution: Let S_{t-1} denote one of the hypothetical optimal schedules assumed to exist by the loop invariant.

Algorithm's Actions: If event i conflicts with an event in *Commit*, then the algorithm rejects it. S_{t-1} contains all the events in *Commit* and hence cannot contain event i either. Hence, the loop invariant is maintained. From here on, let us assume that i does not conflict with any event in *Commit* and hence will be added to it.

Modifying Optimal Solutions: If we are lucky, the schedule S_{t-1} already contains event i . In this case, we are done. Otherwise, we will modify the schedule S_{t-1} into another schedule S_t by adding i and removing any events that conflict with it. This modifying is not part of the algorithm, as we do not actually have an optimal schedule yet.

A Valid Solution: Our modified set S_t contains no conflicts, because S_{t-1} contained none and we were careful to introduce none.

Consistent with Choices Made: S_{t-1} was consistent with the previous choices. We added event i to make S_t consistent with these choices. We did not remove any events from *Commit*, because these do not conflict with event i .

Optimal: To prove that S_t has the optimal number of events in it, we need only to prove that it has at least as many as S_{t-1} . We added one event to the schedule. Hence, we must prove that we have not removed more than one. Let j denote a deleted event. Being in SLI and not in *Commit*, it must be an event not yet considered. Because the events are sorted based on their finishing time, $j > i$ implies that event j finishes after event i finishes, that is, $f_j \geq f_i$. If event j conflicts

with event i , it follows that it also starts before it finishes, i.e., $s_j \leq f_i$. (In Figure 22.2, there are three future events conflicting with i .) Combining $f_j \geq f_i$ and $s_j \leq f_i$ gives that such an event j is running at the finishing time f_i of event i . Hence, any two such events j conflict with each other. Therefore, they cannot both be in the schedule S_{t-1} , because it contains conflicts.

Loop Invariant Has Been Maintained: In conclusion, we have constructed a valid optimal schedule S_t that contains the events in $Commit \cup \{i\}$ and no rejected events. This proves that the loop invariant is maintained.

Exiting Loop ($\langle LI \rangle \ \& \ \langle exit \rangle \Rightarrow \langle post \rangle$): By LI, there is an optimal schedule S_{t-1} containing the events in $Commit$ and not containing the previous events not in $Commit$. Because all events are previous events, it follows that $Commit = S_{t-1}$ is in an optimal schedule for our instance.

Running Time: The loop is iterated once for each of the n events. The only work is determining whether event i conflicts with an event within $Commit$. Because of the ordering of the events, event i finishes after all the events in $Commit$. Hence, it conflicts with an event in $Commit$ if and only if it starts before the last finishing time of an event in it. It is easy to remember this last finishing time, because it is simply the finishing time of the last event to be added to $Commit$. Hence, the main loop runs in $\Theta(n)$ time. The total time of the algorithm then is dominated by the time to sort the events.

22.2.2 Example: The Interval Cover Problem

For this problem, we will develop an adaptive priority greedy algorithm.

Specifications: Given a set of points and intervals, the goal is to find an *optimal cover*, that is, a subset of the intervals that covers all the points and that contains the minimum number of intervals.

Precondition: We are given one of the following instances.

Instances: An instance consists a set P of points and a set I of intervals on the real line. An interval consists of a starting and a finishing time $\langle s_i, f_i \rangle$. See the example in Figure 22.3.



Fig. 22.3 An example instance of the interval cover problem. The intervals in one optimal solution are highlighted.

Postcondition: The output is a solution with minimum cost and the cost of that solution.

Solutions: A solution for an instance is a subset S of the intervals that covers all the points. It is fine if the intervals overlap.

Measure of Success: The cost of a solution S is $|S|$, the number of intervals required. Having longer or shorter intervals does not matter. Covering the points more than once does not matter. Covering parts of the line without points does not matter. Only the number of intervals matters.

The Adaptive Greedy Criterion: The algorithm sorts the points and covers them in order from left to right. If the intervals committed to so far cover all of the points in P , then the algorithm stops. Otherwise, let P_i denote the leftmost point in P that is not covered by *Commit*. The next interval committed to must cover this next uncovered point, P_i . Of the intervals that start to the left of the point, the algorithm greedily takes the one that extends as far to the right as possible. The hope in doing so is that the chosen interval, in addition to covering P_i , will cover as many other points as possible. Let I_j denote this interval. If there is no such interval I_j or it does not extend to the right far enough to cover the point, P_i , then no interval covers this point, and the algorithm reports that no subset of the intervals covers all of the points. Otherwise, the algorithm commits to this interval by adding it to *Commit*. This greedy criterion with which to select the next interval changes as the point P_i to be covered changes.

The Loop Invariant: The loop invariant is that we have not gone wrong: There is at least one optimal solution consistent with the choices made so far, that is, containing the objects committed to so far and not containing the objects rejected so far.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle$ & not $\langle exit \rangle$ & $code_{loop} \Rightarrow \langle LI_t \rangle$): Assume that we are at the top of the loop and that the loop invariant is true, so that there exists an optimal cover that contains all of the intervals in *Commit*. Let S_{t-1} denote such a cover that is assumed to exist. If we are lucky and S_{t-1} already contains the interval I_j being committed to in this iteration, then we automatically know that there exists an optimal cover that contains all of the intervals in $Commit \cup \{I_j\}$ and hence the loop invariant has been maintained. If, on the other hand, the interval I_j being committed to is not in S_{t-1} , then we must modify this optimal solution into another optimal solution that does contain it.

Modifying S_{t-1} into S_t : The optimal solution S_{t-1} must cover the point P_i . Let $I_{j'}$ denote one of the intervals in S_{t-1} that covers P_i . Our solution S_t is the same as S_{t-1} except that $I_{j'}$ is removed and I_j is added. We know that $I_{j'}$ is not in *Commit*, because the point P_i is not covered by *Commit*. Hence, as constructed, S_t contains all the intervals in $Commit \cup \{I_j\}$.

S_t Is an Optimal Solution: Because S_{t-1} is an optimal cover, we can prove that S_t is an optimal cover, proving that it covers all of the points covered by S_{t-1} and that it contains the same number of intervals.

The algorithm considered the point P_i because it was the leftmost uncovered point. It follows that the intervals in *Commit* cover all the points to the left of P_i .

The interval $I_{j'}$, because it covers P_i , must start to the left of P_i . Hence, the algorithm must have considered it when it chose I_j . Now I_j , being the interval that extends as far to the right as possible of those that start to the left of P_i , must extend at least as far to the right as $I_{j'}$, and hence I_j covers as many points to the right as $I_{j'}$ covers. It follows that S_t covers all of the points covered by S_{t-1} .

Because S_t is an optimal solution containing $\text{Commit} \cup \{I_j\}$, we have proved such a solution exists. Hence, the loop invariant has been maintained.

Maintaining the Greedy Criterion: As the point P_i to be covered changes, the greedy criterion according to which the next interval is chosen changes. Blindly searching for the interval that is best according to the current greedy criterion would be too time-consuming. The following data structures help to make the algorithm more efficient.

An Event Queue: The progress of the algorithm can be viewed as an *event marker* moving along the real line. An *event* in the algorithm occurs when this marker reaches either the start of an interval or a point to be covered. This is implemented not with an actual marker, but with an *event queue*. The queue is constructed initially by sorting the intervals according to their start time and the points according to their position, and merging these two lists together. The algorithm removes and processes these events one at a time.

Additional Loop Invariants: The following additional loop invariants relate the current position event marker with to current greedy criterion.

LI1, Points Covered: All the points to the left of the event marker have been covered by the intervals in *Commit*.

LI2, the Priority Queue: A priority queue contains all intervals (except possibly those in *Commit*) that start to the left of the event marker. The priority according to which they are organized is how far f_j to the right the interval extends. This priority queue can be implemented using a heap. (See Section 13.3.4.)

LI3, Last Place Covered: A variable *last* indicates the rightmost place covered by an interval in *Commit*.

Maintaining the Additional Loop Invariants:

A Start Interval Event: When the event marker passes the starting time s_j of an interval I_j , this interval from then on will start to the left of the event marker and hence is added to the priority queue, its priority being its finishing time f_j .

An End Interval Event: When the event marker passes the finishing time f_j of an interval I_j , we learn that this interval will not be able to cover future points P_i . Though the algorithm no longer wants to consider this interval, the algorithm will be lazy and leave it in the priority queue. Its priority will

be lower than those actually covering P_i . The algorithm does not consider these events, there being nothing useful to do with them.

A Point Event: When the event marker reaches a point P_i in P , the algorithm uses $last \geq P_i$ to check whether the point is already covered by an interval in *Commit*. If it is covered, then nothing needs to be done. If not, then LI1 assures us that this point is the leftmost uncovered point. LI2 ensures that the priority queue is organizing the intervals according to the current greedy criterion, namely, it contains all intervals that start to the left of the point P_i , sorted according to how far to the right the interval extends. Let I_j denote the highest-priority interval in the priority queue. Assuming that it covers P_i , the algorithm commits to it. As well, if it covers P_i , then it must extend further to the right than other intervals in *Commit*, and hence $last$ is updated to f_j . (The algorithm can either remove the interval I_j committed to from the priority queue or not. That interval will not extend far enough to the right to cover the next uncovered point, and hence its priority will be low in the queue.)

Code: Interval point cover.

algorithm *IntervalPointCover*(P, I)

<pre-cond>: P is a set of points, and I is a set of intervals on a line.

<post-cond>: The output consists of the smallest set of intervals that covers all of the points.

begin

Sort $P = \{P_1, \dots, P_n\}$ in ascending order of p_i s.

Sort $I = \{\langle s_1, f_1 \rangle, \dots, \langle s_m, f_m \rangle\}$ in ascending order of s_j s.

$Events = Merge(P, I)$ % sorted in ascending order

$consideredI = \emptyset$ % the priority queue of intervals being considered

$Commit = \emptyset$ % solution set: covering subset of intervals

$last = -\infty$ % rightmost point covered by intervals in *Commit*

for each event $e \in Events$, in ascending order do

if($e = \langle s_j, f_j \rangle$) then

Insert interval $\langle s_j, f_j \rangle$ into the priority queue *consideredI* with priority f_j

else ($e = P_i$)

if($P_i > last$) then % P_i is not covered by *Commit*

$\langle s_j, f_j \rangle = ExtractMax(consideredI)$ % f_j is max in *consideredI*

if(*consideredI* was empty or $P_i > f_j$) then

return (P_i cannot be covered)

else

$Commit = Commit \cup \{j\}$

$last = f_j$

end if


```

        end if
    end if
end for
return (Commit)
end algorithm

```

Running Time: The initial sorting takes $O((n+m) \log(n+m))$ time. The main loop iterates $n+m$ times, once per event. Since H contains a subset of I , the priority queue operations *Insert* and *ExtractMax* each take $O(\log m)$ time. The remaining operations of the loop take $O(1)$ time per iteration. Hence, the loop takes a total of $O((n+m) \log m)$ time. Therefore, the running time of the algorithm is $O((n+m) \log(n+m))$.

22.2.3 Example: The Minimum-Spanning-Tree Problem

Suppose that you are building a network of computers. You need to decide which pairs of computers to run a communication line between. You want all of the computers to be connected via the network. You want to do it in a way that minimizes your costs. This is an example of the minimum-spanning-tree problem.

Definitions: Consider a subset S of the edges of an undirected graph G .

A Tree: S is said to be *tree* if it contains no cycles and is connected.

Spanning Set: S is said to *span* the graph iff every pair of nodes is connected by a path through edges in S .

Spanning Tree: S is said to be a *spanning tree* of G iff it is a tree that spans the graph. Cycles will cause redundant paths between pairs of nodes.

Minimum Spanning Tree: S is said to be a *minimum spanning tree* of G iff it is a spanning tree with minimal total edge weight.

Spanning Forest: If the graph G is not connected, then it cannot have a spanning tree. S is said to be a *spanning forest* of G iff it is a collection of trees that spans each of the connected components of the graph. In other words, pairs of nodes that are connected by a path in G are still connected by a path if we consider only the edges in S .

Specification: The goal of the *minimum-spanning-tree (-forest) problem* is to find a minimum spanning tree (forest) for a given graph.

Precondition: We are given one of the following instances.

Instances: An instance consists of an undirected graph G . Each edge $\{u, v\}$ is labeled with a real-valued (possibly negative) weight $w_{\{u, v\}}$.

Postcondition: The output is a solution with minimum cost and the cost of that solution.

Solutions: A solution for an instance is a spanning tree (forest) S of the graph G .

Measure of Success: The cost of a solution S is the sum of its edge weights.

Possible Criteria for Defining “Best”

Cheapest Edge (Kruskal’s Algorithm): The obvious greedy algorithm simply commits to the cheapest edge that does not create a cycle with the edges committed to already. See the example in [Figure 22.4](#).

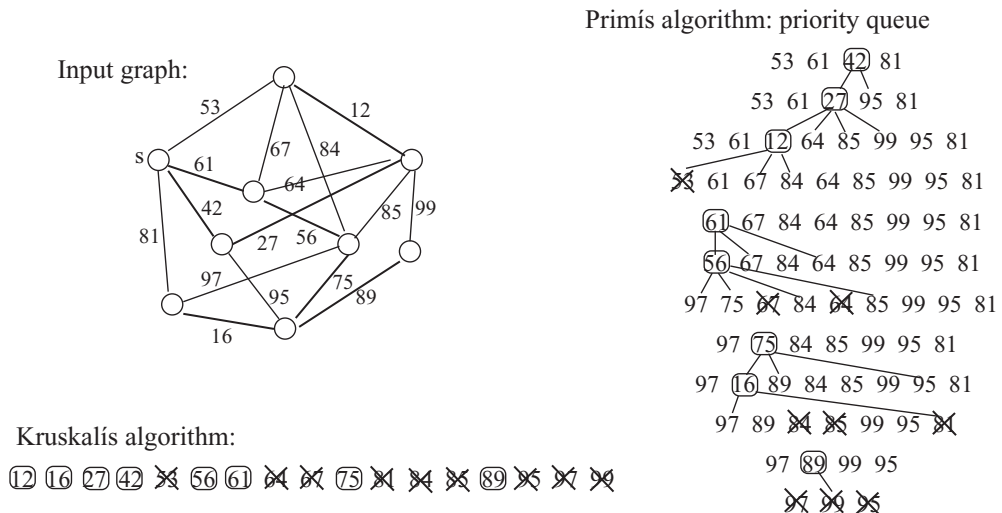


Fig. 22.4 Both Kruskal's and Prim's algorithms are run on the given graph. For Kruskal's algorithm, the sorted order of the edges is shown. For Prim's algorithm, the running content of the priority queue is shown (edges are in no particular order). Each iteration, the best edge is considered. If it does not create a cycle, it is added to the minimum spanning tree. This is shown by circling the edge weight and darkening the graph edge. For Prim's algorithm, the lines out of the circles indicate how the priority queue is updated. If the best edge does create a cycle, then the edge weight is crossed out.

Checking for a Cycle: One task that this algorithm must be able to do quickly is to determine whether the new edge i creates a cycle with the edges in *Commit*. As a task in itself, this would take a while. However, if we maintain an extra data structure, this task can be done very quickly.

Connected Components of Commit: We partition the nodes in the graph into sets so that between any two nodes in the same set, *Commit* provides a path between them, and between any two nodes in different sets, *Commit* does not connect them. These sets are referred to as *components* of the subgraph induced by the edges in *Commit*. The algorithm can determine whether the new edge i creates a cycle

with its edges in *Commit* by checking whether the end points of the edge $i = \{u, v\}$ are contained in the same component. The required operations on components are handled by the following *union–find* data structure.

Union–Find Set System: This data structure maintains a number of disjoint sets of elements and allows three operations: (1) *Makeset*(v), which creates a new set containing the specified element v ; (2) *Find*(v), which determines the *name* of the set containing a specified element; and (3) *Union*(u, v), which merges the sets containing the specified elements u and v . On average, for all practical purposes, each of these operations can be completed in a constant amount of time. See [Section 3.1](#).

Code: Kruskal’s algorithm with the union–find data structure incorporated is as follows.

algorithm *KruskalMST*(G)

$\langle$ *pre-cond* $\rangle$: G is an undirected graph.

$\langle$ *post-cond* $\rangle$: The output consists of a minimum spanning tree.

 begin

 Sort the edges based on their weights $w_{\{u, v\}}$.

$Commit = \emptyset$ % The set of edges committed to

 loop for each v

 % With no edges in *Commit*, each node is in a component by itself.

$MakeSet(v)$

 end for

 loop $i = 1 \dots m$ % Consider the edges in sorted order.

u and v are end points of edge i .

 if ($Find(u) \neq Find(v)$) then

 % The end points of edge i are in different components and hence do not create a cycle with edges in *Commit*.

$Commit = Commit \cup \{i\}$

$Union(u, v)$ % Edge i connects the two components: hence they are merged into one component.

 end if

 end loop

 return (*Commit*)

 end algorithm

Running Time: The initial sorting takes $O(m \log m)$ time when G has m edges. The main loop iterates m times, once per edge. Checking for a cycle takes time $\alpha(n) \leq 4$ on average. Therefore, the running time of the algorithm is $O(m \log m)$.

Running Time: The main loop iterates m times, once per edge. The priority queue operations *Insert* and *ExtractMax* each takes $O(\log m)$ time. Therefore, the running time of the algorithm is $O(m \log m)$.

A More General Algorithm: When designing an algorithm it is best to leave as many implementation details unspecified as possible. This gives more freedom to anyone who may want to implement or to modify your algorithm. Also, it provides better intuition as to why the algorithm works. The following is a greedy criterion that is quite general.

Cheapest Connected Edge of Some Component: Partition the nodes of the graph G into connected *components* of nodes that are reachable from each other only through the edges in *Commit*. Nodes with no adjacent edges will be in a component by themselves. Each iteration, the algorithm is free to choose one of these components for any reason it likes. Denote this component by C . (C can be the union of a number of different components.) Then the algorithm greedily commits to the cheapest edge of those that expand this component, that is, the cheapest from among those edges that are connected to the component and yet do not create a cycle with it. Whenever it likes, the algorithm also has the freedom to throw away uncommitted edges that create a cycle with the edges in *Commit*.

Generalizing: This greedy criterion is general enough to include both Kruskal's and Prim's algorithms. Therefore, if we prove that this greedy algorithm works no matter how it is implemented, then we automatically prove that both Kruskal's and Prim's algorithms work.

Cheapest Edge (Kruskal's Algorithm): This general algorithm may choose the component that is connected to the cheapest uncommitted edge that does not create a cycle. Then when it chooses the cheapest edge out of this component, it gets the overall cheapest edge.

Cheapest Connected Edge (Prim's Algorithm): This general algorithm may always choose the component that contains the source node s . This amounts to Prim's algorithm.

The Loop Invariant: The loop invariant is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle \wedge \text{not } \langle \text{exit} \rangle \wedge \text{code}_{\text{loop}} \implies \langle LI_t \rangle$): See the example in Figure 22.5. If we are unlucky and the optimal minimum spanning tree S_{t-1} that is assumed to exist by the loop invariant does not contain the edge i being committed to this iteration, then we must modify this optimal solution S_{t-1} into another one S_t that contains all of the edges in $\text{Commit} \cup \{i\}$. This proves that the loop invariant has been maintained.

Modifying S_{t-1} into S_t : To the optimal solution S_{t-1} we add the edge $i = \{u, v\}$ being committed to this iteration. Because S_{t-1} spans the graph G , there is some path P within it from node u to node v . This path along with the edge $i = \{u, v\}$ creates a cycle, which must be broken by removing one edge from P . Let C denote the component of *Commit* that the general greedy algorithm chooses the edge i

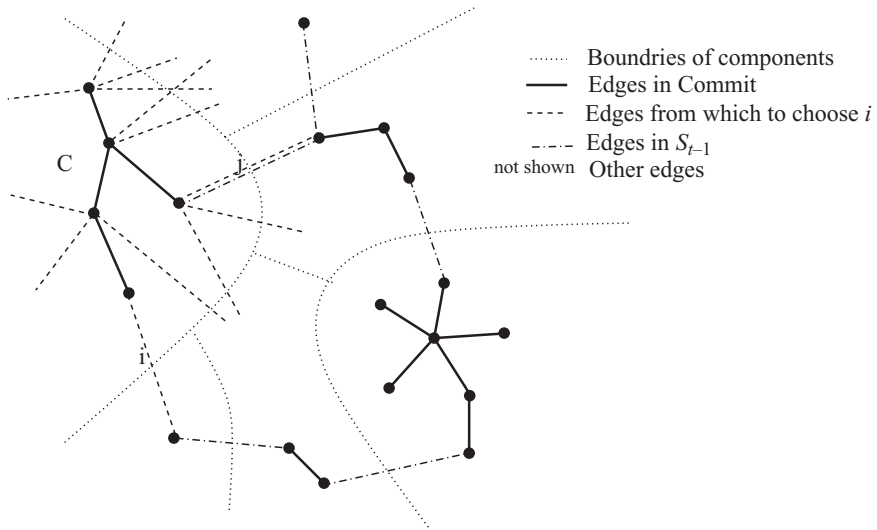


Fig. 22.5 C is one of the components of the graph induced by the edges in *Commit*. Edge i is the cheapest out of C . Edge j is an edge in S_{t-1} that is out of C . Finally, S_t is formed by removing j and adding i .

from. Because i expands the component without creating a cycle with it, one and only one of the edge i 's nodes u and v is within the component. Hence, this path P starts within C and ends outside of C . Hence, there must be some edge j in the path that leaves C . We will delete this edge from our optimal minimum spanning tree: $S_t = S_{t-1} \cup \{i\} - \{j\}$.

S_t is an Optimal Solution:

S_t Has No Cycles: Because S_{t-1} has no cycles, we create only one cycle by adding edge i , and we destroy this cycle by deleting edge j .

S_t Spans G : Because S_{t-1} spans G , S_t spans it as well. Any path between two nodes that goes through edge j in S_{t-1} will now follow the remaining edges of path P together with edge i in S_t .

S_t Has Minimum Weight: Because S_{t-1} has minimum weight, it is sufficient to prove that edge i is at least as cheap as edge j . Note that by construction edge j leaves component C and does not create a cycle with it. Because edge i was chosen because it was the cheapest such edges, (or at least one of the cheapest) it is true that edge i is at least as cheap as edge j .

Exercise 22.2.1 Show how the earliest-finishing-time criterion works on the three counterexamples in [Section 22.2.1](#) and on the example in [Figure 22.1](#).

Exercise 22.2.2 (See solution in Exercise Solutions.) For the example in [Section 22.2.2](#), consider the greedy criterion that selects the interval that covers the largest number of uncovered points. Does this work? If not, give a counterexample.

Exercise 22.2.3 In Figure 22.4, suppose we decide to change the weight 56 to some other real number from $-\infty$ to $+\infty$. What is the interval of values that it could be changed to for which the minimum spanning tree remains the same? Explain your answer. Similarly for the weight 85.

Exercise 22.2.4 Give a simple graph with edge weights for which the tree of shortest weighted paths from nodes is not a minimum spanning tree.

22.3 Exercises

Exercise 22.3.1 One aspect of the game called magic (or jugio) is as follows. You have n defense cards, the i th of which, denoted D_i , has worth w_i and defense ability d_i . Your opponent has n attack cards, the j th of which, denoted A_j , has attack ability a_j . You can see all of the cards. Your task is to define a one-to-one matching between the attacking cards and the defending cards, that is, each attack card is allocated to a unique defense card. If card D_i is defending against A_j and $d_i < a_j$, then d_i dies. Your goal is to maximize the sum of the worths w_i of your cards that live. Give your algorithm. Then prove that it works.

Exercise 22.3.2 (See solution in Exercise Solutions.) Review the job/event scheduling problem from Section 22.2.1. This problem is the same except you have r rooms/processors within which to schedule that the set of jobs/events in two rooms or on two processors. An instance is $\langle r, \langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \dots, \langle s_n, f_n \rangle \rangle$, where, as before, $0 \leq s_i \leq f_i$ are the starting and finishing times for the i th event. But now the input also specifies the number of rooms, r . A solution for an instance is a schedule $S = \langle S_1, \dots, S_r \rangle$ for each of the rooms. Each of these consists of a subset $S_j \subseteq [1 \dots n]$ of the events that don't conflict by overlapping in time. The success of a solution S is the number of events scheduled, that is, $|\cup_{j \in [r]} S_j|$. Consider the following four algorithms:

1. Find the greedy solution for the first room. Then find the greedy solution for the second room from the remaining events. Then for the third room, and so on.
2. The algorithm starts by sorting the events by their finishing times, just as in the one-room case. Then, it looks at each event in turn, scheduling it if possible. If it can be scheduled in more than one room, we assign it in the first room in which it fits, i.e., first try room 1, then room 2, and so on, until it fits. If it cannot be scheduled in any room, then it is not scheduled.
3. The same, except that the next event is scheduled in the room with the latest last-scheduled finishing time. For example, if the last event scheduled in room 5 finishes at time 10 and the last event scheduled in room 12 finishes at time 15 and the next event starts at time 17, then the next event could be scheduled into either of these rooms. This algorithm would schedule it, however, in room 12.
4. The same except that the next event is scheduled in the room with the earliest last-scheduled finishing time.

Prove that three of these algorithms do not lead to an optimal schedule, and the remaining one does.

23 Recursive Backtracking

The brute force algorithm for an optimization problem is to simply compute the cost or value of each of the exponential number of possible solutions and return the best. A key problem with this algorithm is that it takes exponential time. Another (not obviously trivial) problem is how to write code that enumerates over all possible solutions. Often the easiest way to do this is *recursive backtracking*. The idea is to design a recurrence relation that says how to find an optimal solution for one instance of the problem from optimal solutions for some number of smaller instances of the same problem. The optimal solutions for these smaller instances are found by recursing. After unwinding the recursion tree, one sees that recursive backtracking effectively enumerates all options. Though the technique may seem confusing at first, once you get the hang of recursion, it really is the simplest way of writing code to accomplish this task. Moreover, with a little insight one can significantly improve the running time by pruning off entire branches of the recursion tree. In practice, if the instance that one needs to solve is sufficiently small and has enough structure that a lot of pruning is possible, then an optimal solution can be found for the instance reasonably quickly. For some problems, the set of subinstances that get solved in the recursion tree is sufficiently small and predictable that the recursive backtracking algorithm can be mechanically converted into a quick dynamic programming algorithm. See [Chapter 24](#). In general, however, for most optimization problems, for large worst case instances, the running time is still exponential.

23.1 Recursive Backtracking Algorithms

An Algorithm as a Sequence of Decisions: An algorithm for finding an optimal solution for your instance must make a sequence of small decisions about the solution: “Do we include the first object in the solution or not?” “Do we include the second?” “The third?” . . . , or “At the first fork in the road, do we go left or right?” “At the second fork which direction do we go?” “At the third?” . . . As one stack frame in the recursive algorithm, our task is to deal only with the first of these decisions. A recursive friend will deal with the rest. We saw in [Chapter 22](#) that greedy algorithms make decisions simply by committing to the option that looks best at the moment. However, this usually does not work. Often, in fact, we have no inspirational technique to know how to make

each decision in a way that leads to an optimal (or even a sufficiently good) solution. The difficulty is that it is hard to see the global consequences of the local choices that we make. Sometimes a local initial sacrifice can globally lead to a better overall solution. Instead, we use perspiration. We try all options.

Example 23.1.1 (Searching a Maze)

When we come to a fork in the road, all possible directions need to be tried. For each, we get a friend to search exhaustively, backtrack to the fork, and report the highlights. Our task is to determine which of these answers is best overall. Our friends will have their own forks to deal with. However, it is best not to worry about this, since their path is their responsibility, not ours.

High-Level Code: The following is the basic structure that the code will take.

algorithm *Alg(I)*

<pre-cond>: *I* is an instance of the problem.

<post-cond>: *optSol* is one of the optimal solutions for the instance *I*, and *optCost* is its cost.

begin

 if(*I* is small) then

 return(brute force answer)

 else

 % Deal with the first decision by trying each of the *K* possibilities.
 for *k* = 1 to *K*.

 % Temporarily, commit to making the first decision in the *k*th way.

$\langle optSol_k, optCost_k \rangle =$ Recursively deal with all the remaining decisions, and in doing so find the best solution $optSol_k$ for our instance *I* from among those consistent with this *k*th way of making the first decision. $optCost_k$ is its cost.

 end for

 % Having the best, $optSol_k$, for each possibility *k*, we keep the best of these best.

$k_{min} =$ “a *k* that minimizes $optCost_k$ ”

$optSol = optSol_{k_{min}}$

$optCost = optCost_{k_{min}}$

 return $\langle optSol, optCost \rangle$

 end if

end algorithm

Example 23.1.2 (Searching for the Best Animal)

Suppose, instead of searching through a structured maze, we are searching through a large set of objects, say for the best animal at the zoo. See [Figure 23.1](#). Again, we break the search into smaller searches, each of which we delegate to a friend. We might ask one friend for the best vertebrate and another for the best invertebrate. We will take the better of these best as our answer. This algorithm is recursive. The friend with the vertebrate task asks a friend to find the best mammal, another for the best bird, and another for the best reptile.

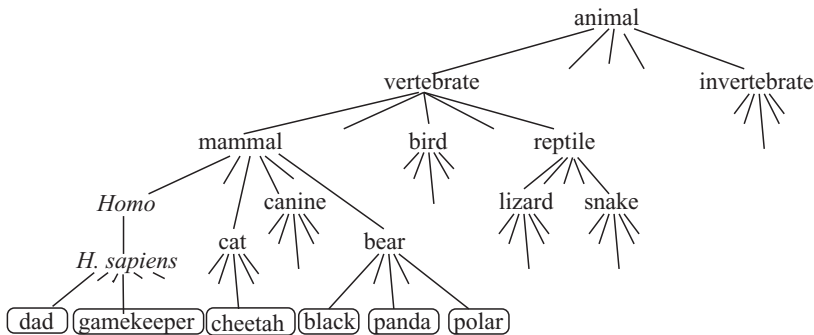


Fig. 23.1 Classification tree of animals.

A Classification Tree of Solutions: This algorithm unwinds into the tree of stack frames that directly mirrors the taxonomy tree that classifies animals. Each solution is identified with a leaf.

Iterating through the Solutions to Find the Optimal One: This algorithm amounts to using depth-first search ([Section 20.4](#)) to traverse this classification tree, iterating through all the solutions associated with the leaves. Though this algorithm may seem complex, it is often the easiest way to iterate through all solutions.

Speeding Up the Algorithm: This algorithm is not any faster than the brute force algorithm that simply compares each animal with every other. However, the structure that the recursive backtracking adds can possibly be exploited to speed up the algorithm. A branch of the tree can be pruned off when we know that this does not eliminate all optimal solutions. Greedy algorithms ([Chapter 22](#)) prune off all branches except one path down the tree. In [Section 24.2](#), we will see how dynamic programming reuses the optimal solution from one subtree within another subtree.

The Little Bird Abstraction: I like to use a *little bird* abstraction to help focus on two of the most difficult and creative parts of designing a recursive backtracking algorithm.

What Question to Ask: The key difference between searching a maze and searching for the best animal is that in the first the forks are fixed by the problem,

but in the second the algorithm designer is able to choose them. Instead of forking on vertebrates vs. invertebrates, we could fork on brown animals vs. green animals. This choice is a difficult and creative part of the algorithm design process. It dictates the entire structure of the algorithm, which in turns dictates how well the algorithm can be sped up. I like to view this process of forking as asking a *little bird* a question, “Is the best animal a vertebrate or an invertebrate?” or “Is the best vertebrate, a mammal, a bird, a reptile, or a fish?” The classification tree becomes a strategy for the game of twenty questions. Each sequence of possible answers (for example, vertebrate–mammal–cat–cheetah) uniquely specifies an animal. Worrying only about the top level of recursion, the algorithm designer must formulate one small question about the optimal solution that is being searched for. The question should be such that having a correct answer greatly reduces your search.

Constructing a Subinstance for a Friend: The second creative part of designing a recursive backtracking algorithm is how to express the problem “Find the best mammal” as a smaller instance of the same search problem. The little bird helps again. We pretend that they answered “mammal.” Trusting (at least temporarily) in her answer helps us focus on the fact that we are now only considering mammals, and this helps us to design a subinstance that asks for the best one. A solution to this subinstance needs to be translated before it is in the correct form to be a solution to our instance.

A Flock of Stupid Birds vs. Wise Little Bird: The following two ways of thinking about the algorithm are equivalent.

A Flock of Stupid Birds: Suppose that our question about whether the optimal solution is a mammal, a bird, or a reptile has K different answers. For each, we pretend that a bird gave us this answer. Giving the bird the benefit of doubt, we ask a friend to give us the optimal solution from among those that are consistent with this answer. At least one of these birds must have been telling us the truth. We find it by taking the best of the optimal solutions obtained in this way. See [Figure 23.2](#) for an illustration of these ideas.

A Wise Little Bird: If we had a little bird who would answer our questions correctly, designing an algorithm would be a lot easier: We ask the little bird, “Is the best animal a bird, a mammal, a reptile, or a fish?” They tell us a mammal. We ask our friend for the best mammal. Trusting the little bird and the friend, we give this as the best animal. Just as nondeterministic finite automata (NFAs) and nondeterministic Turing machines can be viewed as higher powers that provide help, our little bird can be viewed as a limited higher power. The bird is limited in that we can only ask them questions that do not have too many possible answers, because in reality we must try all these possible answers.

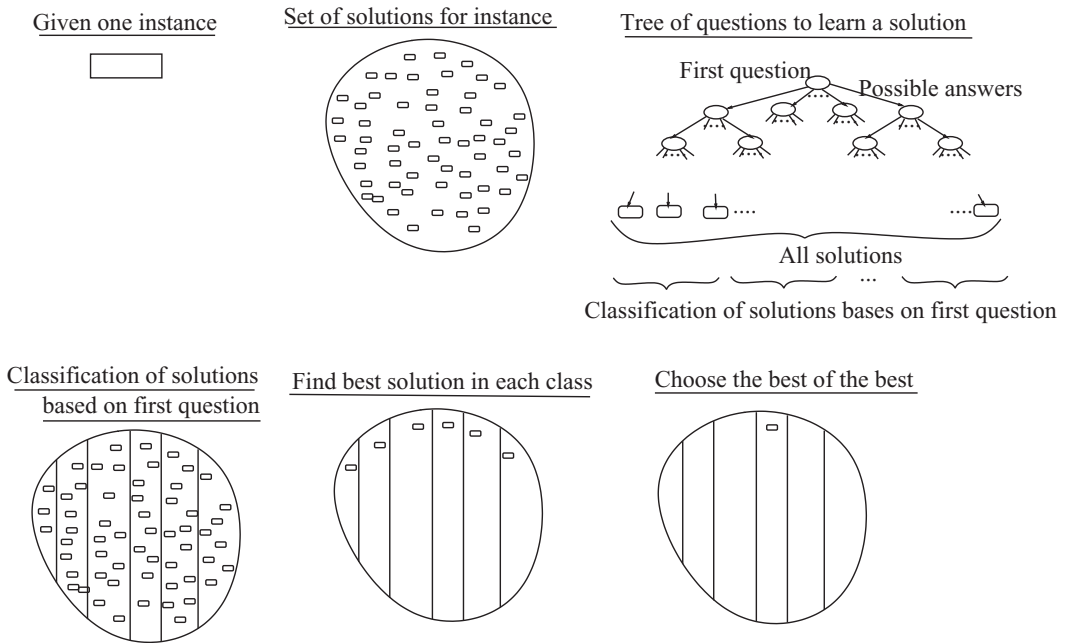


Fig. 23.2 Classifying solutions and taking the best of the best.

23.2 The Steps in Developing a Recursive Backtracking

This section presents the steps that I recommend using when developing a recursive backtracking algorithm. To demonstrate them, we will develop an algorithm for the *queens problem*.

Example 23.2.1 (The Queens Problem)

Physically get yourself (or make on paper) a chessboard and eight tokens to act as queens. More generally, you could consider an $n \times n$ board and n queens. A queen can move as far as she pleases, horizontally, vertically, or diagonally. The goal is to place all the queens on the board in a way such that no queen is able to capture any other.

Try It: Before reading on, try yourself to place the queens on an 8-by-8 board. How would you do it?

1) Specification: The first step is to be very clear about what problem needs to be solved. For an optimization problem, we need to be clear about what the set of instances is; for each instance, what its set of solutions is; and for each solution, what its cost or value is.

Queens: The set of possible solutions is the set of ways placing the queens on the board. We do not value one solution over another as long as it is valid, i.e.,

no queen is able to capture any other queen. Hence, the value of a solution can simply be one if it is a valid solution and zero if not. What is not clear is what an input instance for this problem is, beyond the dimension n . We will need to generalize the problem to include more instances in order to be able to recurse. However, we will get back to that later.

2) Design a Question and Its Answers for the Little Bird: Suppose the little bird knows one of the optimal solutions for our instance. You, the algorithm designer, must formulate a small question about this solution and the list of its possible answers.

Question about the Solution: The question should be such that having a correct answer greatly reduces the search. Generally, we ask for the first *part* of the solution.

Queens: We might ask the bird, “Where should I place the first queen?”

The Possible Bird Answers: Together with your question, you provide the little bird with a list $A_1, A_2, \dots, A_K$ of possible answers, and the bird simply returns the index $k \in [1 \dots K]$ of their answer. To be consistent, we will always use the letter k to index the bird answers. In order for the final algorithm to be efficient, it is important that the number K of different answers be small.

Queens: Given that there are n queens and n rows and that two queens cannot be placed in the same row, the first observation is that the first queen must be in the first row. Hence, there are $K = n$ different answers the bird might give.

3) Constructing Subinstances: Suppose that the bird gives us the k th of their answers. This gives us some of the solution; we want to ask a recursive friend for the rest of the solution. The friend must be given a smaller instance of the same search problem. You must formulate subinstance *subI* for the friend so that they return to us the information that we desire.

Queens: I start by (at least temporarily) trusting the bird, and I place a queen where the bird says in the first row. In order to give an instance to the friend, we need to go back to step 1 and generalize the problem. An input instance will specify the locations of queens in the first r rows. A solution is a valid way of putting the queens in the remaining rows. Given such an instance, the question for the bird is, “Where does the queen in the next row go?” Trusting the bird, we place a queen where they say. The instance, *subI*, we give the friend is the board with these $r + 1$ queens placed.

Trust the Friend: We proved in [Section 11.7](#) that we can trust the friend to provide an optimal solution to the subinstance *subI*, because they are really a smaller recursive version of ourselves.

4) Constructing a Solution for My Instance: Suppose that the friend gives you an optimal solution *optSubSol* for their instance *subI*. How do you produce an optimal

solution *optSol* for your instance *I* from the bird's answer *k* and the friend's solution *optSubSol*?

Queens: The bird tells you where on the $r+1^{\text{st}}$ row the queen should go and your friend tells you where on the rows $r+2$ to n the queens should go. Your solution combines these to tell where on the rows $r+1$ to n the queens should go.

We can trust the friend to give provide an optimal solution to the subinstance *subI*, because he is really a smaller recursive version of ourselves. Recall, in [Section 11.7](#), we used strong induction to prove that we can trust our recursive friends.

5) Costs of Solutions and Subsolutions: We must also return the cost *optCost* of our solution *optSol*.

Queens: The solutions in this case don't have costs.

6) Best of the Best: Try all the bird's answers, and take best of the best.

Queens: If we trust both the bird and the friend, we conclude that this process finds us a best placement of the queens. If, however, our little bird gave us the wrong placement of the queen in the $r+1^{\text{st}}$ row, then this might not be the best placement. However, our work was not wasted, because we did succeed in finding the best placement from among those consistent with this bird's answer. Not trusting the little bird, we repeat this process, finding a best placement starting with each of the possible bird answers. Because at least one of the bird answers must be correct, one of these placements must be an overall best. We return the best of these best as the overall best placement.

To find the best of these best placements, let *optSol_k* and *optCost_k* denote the optimal solution for our instance *I* and its cost that we formed when temporarily trusting the *k*th bird's answer. Search through this list of costs *optCost₁*, *optCost₂*, ..., *optCost_K*, finding the best one. Denote the index of the chosen one by *k_{max}*. The optimal solution that we will return is then *optSol* = *optSol_{k_{max}}* and its cost is *optCost* = *optCost_{k_{max}}*.

7) Base Cases: The base case instances are instances of your problem that are small enough that they cannot be solved using steps 2–6, but they can be solved easily in a brute force way. What are these base cases, and what are their solutions?

Queens: If all *n* queens have been placed, then there is nothing to be done.

8) Code: The following code might be made slightly simpler, but in order to be consistent we will always use this same basic structure.

algorithm *Queens*(*C*, *n*, *r*)

<pre-cond>: $C = \langle \langle 1, c_1 \rangle, \langle 2, c_2 \rangle, \dots, \langle r, c_r \rangle \rangle$ places the *j*th queen in the *j*th row and the *c_j*th column. The remaining rows have no queen.

<post-cond>: Returned if possible is a placement *optSol* of the *n* queens

consistent with this initial placement of the first r queens. A placement is legal if no two queens can capture each other. Whether this is possible is flagged with $optCost$ equal to one or zero.

```

begin
    % Base case: If all the queens have been already been placed, then the
    % problem is easy to solve.
    if ( $r = n$ ) then
        if ( $C$  is legal) then  $optSol = C$  &  $optCost = 1$  else  $optCost = 0$ 
        return  $\langle optSol, optCost \rangle$ 
    else
        % General case:
        % Try each possible bird answer.
        loop  $k = 1 \dots n$ 
            % The bird-and-friend algorithm: The bird tells us the column  $k$ 
            % in which to put the  $r + 1$ st queen. We ask the friend to place the
            % remaining queens. This will be our best solution  $optSol_k$  among
            % those consistent with this bird's answer.
             $C' = C \cup \langle r + 1, k \rangle$  % Place a queen at this location.
             $\langle optSol_k, optCost_k \rangle = Queens(C', n, r + 1)$ 
        end for
        % Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the
        % best of these best.
         $k_{max} = a\ k\ that\ maximizes\ optCost_k,$ 
        % i. e., if possible  $k$  for which  $optCost_k = 1$ 
         $optSol = optSol_{k_{max}}$ 
         $optCost = optCost_{k_{max}}$ 
        return  $\langle optSol, optCost \rangle$ 
    end if
end algorithm

```

9) Recurrence Relations: At the core of every recursive backtracking and dynamic programming algorithm is a *recurrence relation*. These define one element of a sequence as a function of previous elements in the same sequence. The following are examples.

The Fibonacci Sequence: If $Fib(i)$ is the i th element in the famous Fibonacci sequence, then $Fib(n) = Fib(n - 1) + Fib(n - 2)$. The base cases $Fib(0) = 0$ and $Fib(1) = 1$ are also needed.

Running Time: If $T(n)$ is the running time of merge sort on an input of n numbers, then we have $T(n) = 2T(\frac{n}{2}) + n$. The base case $T(1) = 1$ is also needed.

Optimal Solution: Suppose $Solution[I]$ is defined to be an optimal solution for instance I of a problem, and $Cost[I]$ is its cost. This is not a sequence of elements like the previous examples, because I is not in integer. However, in [Chapter 24](#)

the instances will be indexed by integers i so that $Solution[i]$ and $Cost[i]$ are sequences. But even when I is an arbitrary input instance, a *recurrence relation* can be developed as follows from the bird – friend algorithm.

$$\begin{aligned}
 &Solution[\text{my instance}] \\
 &= \text{Min}_{k \in [K]} [\text{An optimal solution to my instance from those that} \\
 &\quad \text{are consistent with the } k\text{th bird's answer}] \\
 &= \text{Min}_{k \in [K]} [\text{combine bird's answer and friend's answer}] \\
 &= \text{Min}_{k \in [K]} [\text{combine bird's answer and } Solution[\text{friend's instance}]] \\
 \\
 &Cost[\text{my instance}] \\
 &= \text{Min}_{k \in [K]} [\text{cost of optimal solution to my instance from those that} \\
 &\quad \text{are consistent with the } k\text{th bird's answer}] \\
 &= \text{Min}_{k \in [K]} [\text{combine bird's cost and friend's cost}] \\
 &= \text{Min}_{k \in [K]} [\text{combine bird's cost and } Cost[\text{friend's instance}]]
 \end{aligned}$$

Base cases “ $Solution[\text{small instance}] = \text{solution}$ ” are also needed.

Queens: For our queens example this becomes

$$Queens(C, n, r) = \text{Min}_{k \in [n]} Queens(C \cup \langle r + 1, k \rangle, n, r + 1).$$

This is bit of a silly example, because here all the work is done in the base cases.

Running Time: A recursive backtracking algorithm faithfully enumerates all solutions for your instance and hence requires exponential time. We will see that this time can be reduced by pruning off branches of the recursion tree.

23.3 Pruning Branches

The following are typical reasons why an entire branch of the solution classification tree can be pruned off.

Invalid Solutions: Recall that in a recursive backtracking algorithm, the little bird tells the algorithm something about the solution and then the algorithm recurses by asking a friend a question. Then this friend gets more information about the solution from their little bird, and so on. Hence, following a path down the recursive tree specifies more and more about a solution until a leaf of the tree fully specifies one particular solution. Sometimes it happens that partway down the tree, the algorithm has already received enough information about the solution to determine that it contains a conflict or defect, making any such solution invalid. The algorithm can stop recursing at this point and backtrack. This effectively prunes off the entire subtree of solutions rooted at this node in the tree.

Queens: Before we try placing a queen on the square $\langle r + 1, k \rangle$, we should check to make sure that this does not conflict with the locations of any of the queens on $\langle 1, c_1 \rangle, \langle 2, c_2 \rangle, \dots, \langle r, c_r \rangle$. If it does, we do not need to ask for help from this friend. [Exercise 23.5.4](#) bounds the resulting running time.

Time Saved: The time savings can be huge. Recall that for [Example 12.2.1](#) in [Section 12.2](#), reducing the number of recursive calls from two to one decreased the running time from $\Theta(N)$ to $\Theta(\log N)$, and how in [Example 12.2.2](#) reducing the number of recursive calls from four to three decreased the running time from $\Theta(n^2)$ to $\Theta(n^{1.58\dots})$.

No Highly Valued Solutions: Similarly, when the algorithm arrives at the root of a subtree, it might realize that no solutions within this subtree are rated sufficiently high to be optimal—perhaps because the algorithm has already found a solution provably better than all of these. Again, the algorithm can prune this entire subtree from its search.

Greedy Algorithms: Greedy algorithms are effectively recursive backtracking algorithms with extreme pruning. Whenever the algorithm has a choice as to which little bird's answer to take, i.e., which path down the recursive tree to take, instead of iterating through all of the options, it goes only for the one that looks best according to some greedy criterion. In this way the algorithm follows only one path down the recursive tree. Greedy algorithms are covered in [Chapter 22](#).

Modifying Solutions: Let us recall why greedy algorithms are able to prune, so that we can use the same reasoning with recursive backtracking algorithms. In each step in a greedy algorithm, the algorithm commits to some decision about the solution. This effectively burns some of its bridges, because it eliminates some solutions from consideration. However, this is fine as long as it does not burn all its bridges. The prover proves that there is an optimal solution consistent with the choices made by modifying any possible solution that is not consistent with the latest choice into one that has at least as good value and is consistent with this choice. Similarly, a recursive backtracking algorithm can prune branches in its tree when it knows that this does not eliminate all remaining optimal solutions.

Queens: By symmetry, any solution that has the queen in the second half of the first row can be modified into one that has the this queen in the first half, simply by flipping the solution left to right. Hence, when placing a queen in the first row, there is no need to try placing it in the second half of the row.

Depth-First Search: A recursive depth-first search ([Section 20.5](#)) is a recursive backtracking algorithm. A solution to the optimization problem of searching a maze for cheese is a path in the graph starting from s . The value of a solution is the weight of the node at the end of the path. The algorithm marks nodes that it has visited. Then, when the algorithm revisits a node, it knows that it can prune this subtree in this recursive search, because it knows that any node reachable from the current node has already been reached. In [Figure 20.13](#), the path $\langle s, c, u, v \rangle$ is pruned because it can be modified into the path $\langle s, b, u, v \rangle$, which is just as good.

23.4 Satisfiability

A famous optimization problem is called *satisfiability*, or SAT for short. It is one of the basic problems arising in many fields. The recursive backtracking algorithm given here is referred to as the *Davis–Putnam* algorithm. It is an example of an algorithm whose running time is exponential for worst case inputs, yet in many practical situations can work well. This algorithm is one of the basic algorithms underlying automated theorem proving and robot path planning, among other things.

The Satisfiability Problem:

Instances: An instance (input) consists of a set of constraints on the assignment to the binary variables $x_1, x_2, \dots, x_n$. A typical constraint might be $\langle x_1 \text{ or } \bar{x}_3 \text{ or } x_8 \rangle$, meaning $(x_1 = 1 \text{ or } x_3 = 0 \text{ or } x_8 = 1)$ or equivalently that either x_1 is true, x_3 is false, or x_8 is true. More generally, an instance could be a more general circuit built with *AND*, *OR*, and *NOT* gates, but we leave this until [Section 28.1](#).

Solutions: Each of the 2^n assignments is a possible solution. An assignment is valid for the given instance if it satisfies all of the constraints.

Measure of Success: An assignment is assigned the value one if it satisfies all of the constraints, and the value zero otherwise.

Goal: Given the constraints, the goal is to find a satisfying assignment.

Iterating through the Solutions: The brute force algorithm simply tries each of the 2^n assignments of the variables. Before reading on, think about how you would nonrecursively iterate through all of these solutions. Even this simplest of examples is surprisingly hard.

Nested Loops: The obvious algorithm is to have n nested loops, each going from 0 to 1. However, this requires knowing the value of n before compile time, which is not likely.

Incrementing Binary Numbers: Another option is to treat the assignment as an n -bit binary number and then loop through the 2^n assignments by incrementing this binary number each iteration.

Recursive Algorithm: The recursive backtracking technique is able to iterate through the solutions with much less effort in coding. First the algorithm commits to assigning $x_1 = 0$ and recursively iterates through the 2^{n-1} assignments of the remaining variables. Then the algorithm backtracks, repeating these steps with the choice $x_1 = 1$. Viewed another way, the first little bird question about the solutions is whether the first variable x_1 is set to zero or one, the second question asks about the second variable x_2 , and so on. The 2^n assignments of the variables $x_1, x_2, \dots, x_n$ are associated with the 2^n leaves of the complete binary tree with depth n . A given path from the root to a leaf commits each variable x_i to being

either zero or one by having the path turn to either the left or to the right when reaching the i th level.

Instances and Subinstances: Given an instance, the recursive algorithm must construct two subinstances for its friends to recurse with. There are two techniques for doing this.

Narrowing the Class of Solutions: Associated with each node of the classification tree is a subinstance defined as follows: The set of constraints remains unchanged except that the solutions considered must be consistent in the variables $x_1, x_2, \dots, x_r$ with the assignment given by the path to the node. Traversing a step further down the classification tree further narrows the set of solutions.

Reducing the Instance: Given an instance consisting of a number of constraints on n variables, we first try assigning $x_1 = 0$. The subinstance to be given to the first friend will be the constraints on remaining variables given that $x_1 = 0$. For example, if one of our original constraints is $\langle x_1 \text{ OR } \overline{x_3} \text{ OR } x_8 \rangle$, then after assigning $x_1 = 0$, the reduced constraint will be $\langle \overline{x_3} \text{ OR } x_8 \rangle$. This is because it is no longer possible for x_1 to be true, given that one of $\overline{x_3}$ or x_8 must be true. On the other hand, after assigning $x_1 = 1$, the original constraint is satisfied independently of the values of the other variables, and hence this constraint can be removed.

Pruning: This recursive backtracking algorithm for SAT can be sped up. This can either be viewed globally as a pruning off of entire branches of the classification tree or be viewed locally as seeing that some subinstances, after they have been sufficiently reduced, are trivial to solve.

Pruning Branches off the Tree: Consider the node of the classification tree arrived at down the subpath $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, \dots, x_8 = 0$. All of the assignment solutions consistent with this partial assignment fail to satisfy the constraint $\langle x_1 \text{ OR } \overline{x_3} \text{ OR } x_8 \rangle$. Hence, this entire subtree can be pruned off.

Trivial Subinstances: When the algorithm tries to assign $x_1 = 0$, the constraint $\langle x_1 \text{ OR } \overline{x_3} \text{ OR } x_8 \rangle$ is reduced to $\langle \overline{x_3} \text{ OR } x_8 \rangle$. Assigning $x_2 = 1$ does not change this particular constraint. Assigning $x_3 = 1$ reduces this constraint further to simply $\langle x_8 \rangle$, stating that x_8 must be true. Finally, when the algorithm is considering the value for x_8 , it sees from this constraint that x_8 is *forced* to be one. Hence, the $x_8 = 1$ friend is called, but the $x_8 = 0$ friend is not.

Stop When an Assignment Is Found: The problem specification only asks for one satisfying assignment. Hence, the algorithm can stop when one is found.

Davis–Putnam: The above algorithm branches on the values of each variable, $x_1, x_2, \dots, x_n$, in order. However, there is no particular reason that this order needs to be fixed. Each branch of the recursive algorithm can dynamically use some heuristic to decide which variable to branch on next. For example, if there is a variable, like x_8

in the preceding example, whose assignment is forced by some constraint, then clearly this assignment should be done immediately. Doing so removes this variable from all the other constraints, simplifying the instance. Moreover, if the algorithm branched on $x_4, \dots, x_7$ before the forcing of x_8 , then this same forcing would need to be repeated within all 2^4 of these branches.

If there are no variables to force, a common strategy is to branch on the variable that appears in the largest number of constraints. The thinking is that the removal of this variable may lead to the most simplification of the instance.

An example of how different branches may set the variables in a different order is the following. Suppose that $\langle x_1 \text{ OR } x_2 \rangle$ and $\langle \bar{x}_1 \text{ OR } x_3 \rangle$ are two of the constraints. Assigning $x_1 = 0$ will simplify the first constraint to $\langle x_2 \rangle$ and remove the second constraint. The next step would be to force $x_2 = 1$. On the other hand, assigning $x_1 = 1$ will simplify the second constraint to forcing $x_3 = 1$.

Code

algorithm *DavisPutnam(c)*

<pre-cond>: c is a set of constraints on the assignment to x .

<post-cond>: If possible, $optSol$ is a satisfying assignment and $optCost$ is also one. Otherwise $optCost$ is zero.

begin

 if (c has no constraints or no variables) then

 % c is trivially satisfiable.

 return $\langle \emptyset, 1 \rangle$

 else if (c has both a constraint forcing a variable x_i to 0

 and one forcing the same variable to 1) then

 % c is trivially not satisfiable.

 return $\langle \emptyset, 0 \rangle$

 else

 for any variable forced by a constraint to some value

 substitute this value into c .

 let x_i be the variable that appears the most often in c

 % Loop over the possible bird answers.

 for $k = 0$ to 1 (unless a satisfying solution has been found)

 % Get help from friend.

 let c' be the constraints c with k substituted in for x_i

$\langle optSubSol, optSubCost \rangle = DavisPutnam(c')$

$optSol_k = \langle \text{forced values, } x_i = k, optSubSol \rangle$

$optCost_k = optSubCost$

 end for

 % Take the best bird answer.

$k_{max} = \text{a } k \text{ that maximizes } optCost_k$

$optSol = optSol_{k_{max}}$

$optCost = optCost_{k_{max}}$

```

        return  $\langle optSol, optCost \rangle$ 
    end if
end algorithm

```

Running Time: If no pruning is done, then clearly the running time is $\Omega(2^n)$, as all 2^n assignments are tried. Considerable pruning needs to occur to make the algorithm polynomial time. Certainly in the worst case, the running time is $2^{\Theta(n)}$. In practice, however, the algorithm can be quite fast. For example, suppose that the instance is chosen randomly by choosing m constraints, each of which is the *OR* of three variables or their negations, e.g., $\langle x_1 \text{ OR } \bar{x}_3 \text{ OR } x_8 \rangle$. If few constraints are chosen (say m is less than about $3n$), then with very high probability there are many satisfying assignments and the algorithm quickly finds one of these assignments. If a lot of constraints are chosen, (say m is at least n^2), then with very high probability there are many conflicting constraints, preventing there from being any satisfying assignments, and the algorithm quickly finds one of these contradictions. On the other hand, if the number of constraints chosen is between these thresholds, then it has been proven that the Davis–Putnam algorithm takes exponential time.

23.5 Exercises

Exercise 23.5.1 (See solution in Exercise Solutions.) In one version of the game Scrabble, an input instance consists of a set of letters and a board, and the goal is to find a word that returns the most points. A student described the following recursive backtracking algorithm for it. The bird provides the best word out of the list of letters. The friend provides the best place on the board to put the word. Why are these bad questions?

Exercise 23.5.2 (See solution in Exercise Solutions.) Consider the following Scrabble problem. An instance consists of a set of letters and a dictionary. A solution consists of a permutation of a subset of the given letters. A solution is valid if it is in the dictionary. The value of a solution depends on its placement on the board. The goal is to find a highest-value word that is in the dictionary.

Exercise 23.5.3 (See solution in Exercise Solutions.) Trace the queens algorithm (Section 23.2.1) on the standard 8-by-8 board. What are the first dozen legal outputs for the algorithm? To save time, note that the first two or three queens do not move so fast. Hence, it might be worth it to draw a board with all squares conflicting with these crossed out.

Exercise 23.5.4 (See solution in Exercise Solutions.) What is the running time of the queens algorithm (Section 23.3.1) for the n -by- n board when there is no pruning? Give reasonable upper and lower bounds on the running time of this algorithm after all the pruning occurs.

Exercise 23.5.5 (See solution in Exercise Solutions.) An instance may have many optimal solutions with exactly the same cost. The postcondition of the problem allows any one of these to become output. In any recursive backtracking algorithm, which line of code chooses which of these optimal solutions will be selected?

Exercise 23.5.6 Suppose you are solving SAT from [Section 23.4](#). Suppose your instance is $x \text{ AND } y$, and the little bird tells you to set x to one. What is the instance that you give to your friend? Do the same for instances $\neg x \text{ AND } y$, $x \text{ OR } y$, and $\neg x \text{ OR } y$.

Exercise 23.5.7 Independent set: Given a graph, find a largest subset of the nodes for which there are no edges between any pair in the set. Give the bird-and-friend abstraction of a recursive backtracking algorithm for this problem. What do you ask the bird, and what do give your friend?

Exercise 23.5.8 Graph 3-coloring (3-COL): Given a graph, determine whether its nodes can be colored with three colors so that two nodes do not have the same color if they have an edge between them. What difficulty arises when attempting to design a recursive backtracking algorithm for it? Redefine the problem so that the input consists of a graph and a partial coloring of the nodes. The new goal is to determine whether there is a coloring of the graph consistent with the partial coloring given. Give the bird-and-friend abstraction of a recursive backtracking algorithm for this problem. What do you ask the bird, and what do give your friend?

24 Dynamic Programming Algorithms

Dynamic programming is another powerful tool for solving optimization problems. A table of subinstances is solved. Which are solved is not chosen “dynamically,” as the name implies, but are prescheduled and solved in the order smallest to largest. For each, an optimal solution is found by referring to the solutions for subsubinstances already solved.

In the [previous chapter](#), we had you design a recursive backtracking algorithm as follows. Given our instance to the problem, we ask a magical little bird to tell us something about the end of its optimal solution. Trusting this information, we are able to narrow our search problem down to that expressed by a smaller subinstance to our same problem. Our solution to our instance is a combination of our friend’s solution to their subinstance and the information given to us by the bird. Our friend is recursion. But because we don’t actually have a magical bird, we must try all possible bird answers. Solving our original instance then required recursively solving a number of subinstances which in turn calls on a number. We quickly have an exponentially large tree of subinstances. Sometimes, however, a solution is needed for the same subinstance many times. Dynamic programming reduces the total computation time using *memoization* to solve each subinstance only once. Which subproblems are solved is not chosen “dynamically,” as the name implies, but are prescheduled and solved in the order smallest to largest, storing an optimal solution of each in a table. Because this approach starts with our subinstance, it feels top-down.

Our second way of designing a dynamic program algorithm is to reduce our optimization problem to finding an optimal path through a leveled graph. Our third way considers walking through the computation of our problem. The states that you may reach become your subinstances. The actions that you might take become edges. This reduces your problem to finding an optimal path through this graph. Because the approach starts smallest subinstances, it feels more bottom-up.

24.1 Start by Developing a Recursive Backtracking Algorithm

This section reviews the recommended steps for developing a recursive backtracking algorithm.

Example 24.1.1 (Shortest Weighted Path within a Directed Layered Graph)

To demonstrate the steps, we will develop an algorithm for a version of the shortest-weighted-path problem from Chapter 20. We generalize the problem by allowing negative weights on the edges and simplify it by requiring the input graph to be leveled.

1) Specification: The first step is to be very clear about what problem needs to be solved. For an optimization problem, we need to be clear about what the set of instances is, for each instance what its set of solutions is, and for each solution what its cost is.

Precondition: We are given one of the following instances.

Instances: An instance consists of $\langle G, s, t \rangle$, where G is a weighted directed layered graph, s is a specified source node and t is a specified destination node. See Figure 24.1.a. The graph G has n nodes. Each node has maximum in and outdegree d . Each edge $\langle v_i, v_j \rangle$ is labeled with a real-valued (possibly negative) weight $w_{\langle v_i, v_j \rangle}$. The nodes are partitioned into levels so that each edge is directed from some node to a node in a lower level to prevent cycles. Such graphs are also called DAGs (directed acyclic graphs). It is easiest to assume that the nodes are ordered so that an edge can go from node v_i to node v_j only if $i < j$.

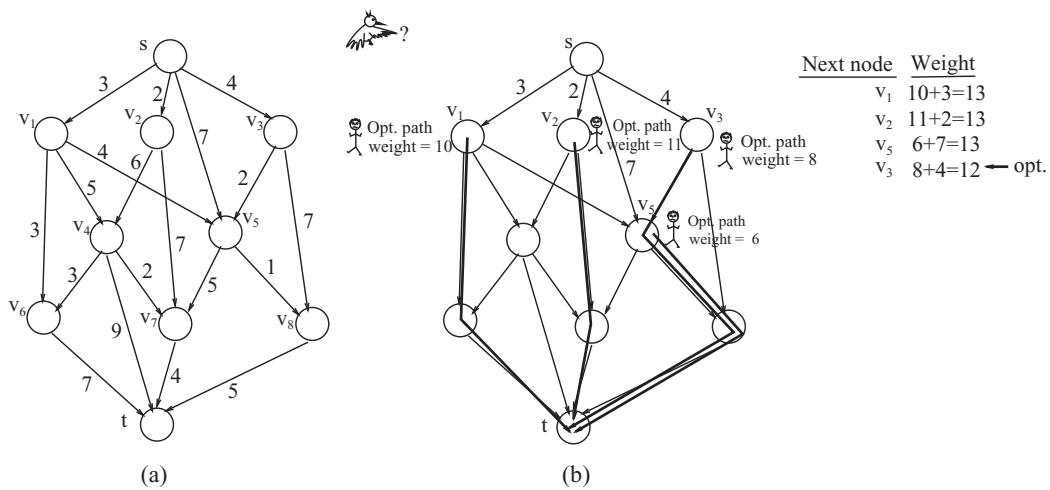


Fig. 24.1 (a) The directed layered weighted graph G for Example 24.1.1. (b) The recursive backtracking algorithm.

Postcondition: The output is a solution with minimum cost and the cost of that solution.

Solutions: A solution for an instance is a path from source node s to destination node t .

Cost of Solution: The cost of a solution is the sum of the weights of the edges within the path.

Brute Force Algorithm: The problem with simply trying all paths is that there may be an exponential number of them.

2) Design a Question and Its Answers for the Little Bird: Suppose the little bird knows one of the optimal solutions for our instance. You, the algorithm designer, must formulate a small question about this solution and the list of its possible answers. The question should be such that having a correct answer greatly reduces the search.

Question about the End of the Solution: When designing recursive backtracking algorithms, one generally asks about the *first part* of the solution. We will later see that if our ultimate goal is a dynamic programming algorithm, then it is best to turn this around and ask for the *last part* of the solution (see [Section 24.3.1](#)).

Leveled Graph: Not knowing yet why we ask about the last part of the solution, we will design this algorithm by asking instead about the first part. Given a graph and nodes s and t , I ask, “Which edge should we take first to form an optimal path to t ?” The little bird assures me that taking edge $\langle s, v_1 \rangle$ is good. The specification of the problem gives that there are at most d edges out of any node. Hence, this is a bound on the number K of different answers.

To be consistent, we will always use the letter k to index the bird answers. In order for the final algorithm to be efficient, it is important that the number K of different answers be small.

3) Constructing Subinstances: Suppose that the bird gives us the k th of their answers. This gives us some of the solution, and we want to ask a recursive friend for the rest of the solution. The friend must be given a smaller instance of the same search problem. You must formulate subinstance *subI* for the friend so that they return to us the information that we desire.

Leveled Graph: I start by (at least temporarily) trusting the bird and take a step along the edge $\langle s, v_1 \rangle$. Standing at v_1 , the natural question to ask my friend is, “Which is the best path from v_1 to t ?” Expressed as $\langle G, v_1, t \rangle$, this is a subinstance of the same computational problem.

4) Constructing a Solution for My Instance: Suppose that the friend gives us an optimal solution *optSubSol* for their instance *subI*. How do we produce an optimal solution *optSol* for your instance I from the bird’s answer k and the friend’s solution *optSubSol*?

Leveled Graph: My friend will faithfully give me the path $\text{optSubSol} = \langle v_1, v_6, t \rangle$, this being a best path from v_1 to t . The difficulty is that this is not a solution for my instance $\langle G, s, t \rangle$, because it is not, in itself, a path from s to t .

The path from s is formed by first taking the step from s to v_i and then following the best path from there to t , namely, $optSol = \langle \text{bird answer} \rangle + optSubSol = \langle s, v_1 \rangle + \langle v_1, v_6, t \rangle = \langle s, v_1, v_6, t \rangle$.

We proved in [Section 11.7](#) that we can trust the friend to give provide an optimal solution to the subinstance $subI$, because they are really a smaller recursive version of ourselves.

5) Costs of Solutions and Subsolutions: We must also return the cost $optCost$ of our solution $optSol$. How do we determine it from the bird's k and the cost $optSubCost$ of the friend's $optSubSol$?

Leveled Graph: The cost of the entire path from s to t is the cost of the edge $\langle s, v_1 \rangle$ plus the cost of the path from v_1 to t . Luckily, our friend gives the latter: $optCost_k = w_{\langle s, v_1 \rangle} + optSubCost = 3 + 10 = 13$.

6) Best of the Best: Try all the bird's answers, and take the best of the best.

Leveled Graph: If we trust both the bird and the friend, we conclude that this path from s to t is a best path. It turns out that because our little bird gave us the wrong first edge, this might not be the best path from s to t . However, our work was not wasted, because we did succeed in finding the best path from among those that start with the edge $\langle s, v_1 \rangle$. Not trusting the little bird, we repeat this process, finding a best path starting with each of $\langle s, v_2 \rangle$, $\langle s, v_5 \rangle$, and $\langle s, v_3 \rangle$. At least one of these four paths must be an overall best path. We give the best of these best as the overall best path.

7) Base Cases: The base case instances are instances of your problem that are small enough that they cannot be solved using steps 2–6, but they can be solved easily in a brute force way. What are these base cases, and what are their solutions?

Leveled Graph: The only base case is finding a best path from s to t when s and t are the very same node. In this case, the bird would be unable to give the first edge in the best path, because it contains no edges. The optimal solution is the empty path, and its cost is zero.

8) Code: From steps 1–7, the code can always be put together using the same basic structure.

algorithm *LeveledGraph*(G, s, t)

<pre-cond> G is a weighted directed layered graph, and s and t are nodes.

<post-cond> $optSol$ is a path with minimum total weight from s to t , and $optCost$ is its weight.

begin

% Base case: The only base case is for the best path from t to t . Its solution is the empty path with cost zero.

```

if( $s = t$ ) then
    return  $\langle \emptyset, 0 \rangle$ 
else
    % General case:
    % Try each possible bird answer.
    for each of the  $d$  edges  $\langle s, v_k \rangle$ 
        % The bird-and-friend algorithm: The bird tells us that the first
        % edge in an optimal path from  $s$  to  $t$  is  $\langle s, v_k \rangle$ . We ask the friend
        % for an optimal path from  $v_k$  to  $t$ . They solve this recursively,
        % giving us  $optSubSol$ . To this, we add the bird's edge, giving us
        %  $optSol_k$ . This  $optSol_k$  is a best path from  $s$  to  $t$  from among
        % those paths consistent with the bird's answer.
         $\langle optSubSol, optSubCost \rangle = LeveledGraph(\langle G, v_k, t \rangle)$ 
         $optSol_k = \langle s, v_k \rangle + optSubSol$ 
         $optCost_k = w_{\langle s, v_k \rangle} + optSubCost$ 
    end for
    % Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the
    % best of these best.
     $k_{min} = \text{"a } k \text{ that minimizes } optCost_k\text{"}$ 
     $optSol = optSol_{k_{min}}$ 
     $optCost = optCost_{k_{min}}$ 
    return  $\langle optSol, optCost \rangle$ 
end if
end algorithm

```

9) Recurrence Relations: The recurrence relation at the core of this recursive backtracking algorithm is the following.

$$\begin{aligned}
 LeveledGraphSolution(G, s, t) &= \text{Min}_{v_k \in N(s)} \langle s, v_k \rangle \\
 &\quad + LeveledGraphSolution(\langle G, v_k, t \rangle); \\
 LeveledGraphCost(G, s, t) &= \text{Min}_{v_k \in N(s)} w_{\langle s, v_k \rangle} + LeveledGraphCost(\langle G, v_k, t \rangle),
 \end{aligned}$$

where $N(s)$ is the set of nodes v_k with edge $\langle s, v_k \rangle$.

Running Time: The recursive backtracking algorithm faithfully enumerates all solutions for your instance and hence requires exponential time.

Exercise 24.1.1 Give a directed leveled graph on n nodes that has a small number of edges and as many paths from s to t as possible.

24.2 The Steps in Developing a Dynamic Programming Algorithm

Though the recursive backtracking algorithm for [Example 24.1.1](#) may seem complex, how else would you iterate through all paths? We will now use *memoization* techniques

to mechanically convert this algorithm into a dynamic programming algorithm that runs in polynomial time. The word “memoization” comes from “memo.” This technique speeds up a recursive algorithm by saving the result for each subinstance it encounters so that it does not need to be recomputed. Dynamic programming takes the idea of memoization one step further. Instead of traversing the tree of recursive stack frames, keeping track of which friends are waiting for answers from which friends, it first determines the complete set of subinstances for which solutions are needed and then computes them in an order such that no friend must wait. As it goes, it fills out a table containing an optimal solution for each subinstance. The technique for finding an optimal solution for a given subinstance is identical to the technique used in the recursive backtracking algorithm. The only difference is that instead of recursing to solve a sub-subinstance, the algorithm looks up in the table an optimal solution found earlier. When the entire table has been completed, the last entry will contain an optimal solution for the original instance.

1) The Set of Subinstances: We obtain the set of subinstances that needs to be solved by the dynamic programming algorithm by tracing the recursive backtracking algorithm through the tree of stack frames starting with the given instance I . The set consists of the initial instance I , its subinstances, their subinstances, and so on. We ensure that this set contains all the required subinstances by making sure that it is closed under this *sub* operation, or equivalently that no subinstance is lonely, because if it's included then so are all its friends. See [Section 24.3.3](#). Also ensure that all (or at least most) of these subinstances are needed.

Leveled Graph

Include $\langle G, v_7, t \rangle$: On $\langle G, s, t \rangle$, the little bird, among other things, suggests taking the edge $\langle s, v_1 \rangle$ leading to the subinstance $\langle G, v_1, t \rangle$. On this subinstance, the little bird, among other things, suggests taking the edge $\langle v_1, v_4 \rangle$ leading to the subinstance $\langle G, v_4, t \rangle$. On this subinstance, the little bird, among other things, suggests taking the edge $\langle v_4, v_7 \rangle$ leading, as said, to the subinstance $\langle G, v_7, t \rangle$. Hence, this subinstance must be considered by the dynamic programming algorithm. See [Figure 24.2.a](#), $\langle G, v_7, t \rangle$.

Exclude $\langle G, v_{81}, t \rangle$: Given the same instance I , $\langle G, v_{81}, t \rangle$ is not a subinstance, because node v_{81} is not a node in the graph, so it never arises.

Exclude $\langle G, v_1, v_8 \rangle$: Neither is $\langle G, v_1, v_8 \rangle$ a required subinstance, because each subinstance that arises is looking for a path that ends in the node t .

Guess the Set: Starting with the instance $\langle G, s, t \rangle$, the complete set of subinstances called will be $\{\langle G, v_i, t \rangle \mid v_i \text{ above } t\}$. See [Figure 24.2.b](#).

Redundancy: We can speed up the recursive backtracking algorithm only when it solves the same subinstance many times.

Leveled Graph: The recursive backtracking algorithm of [Section 24.1](#) traverses each of the exponentially many paths from s to t . Within this exponential amount

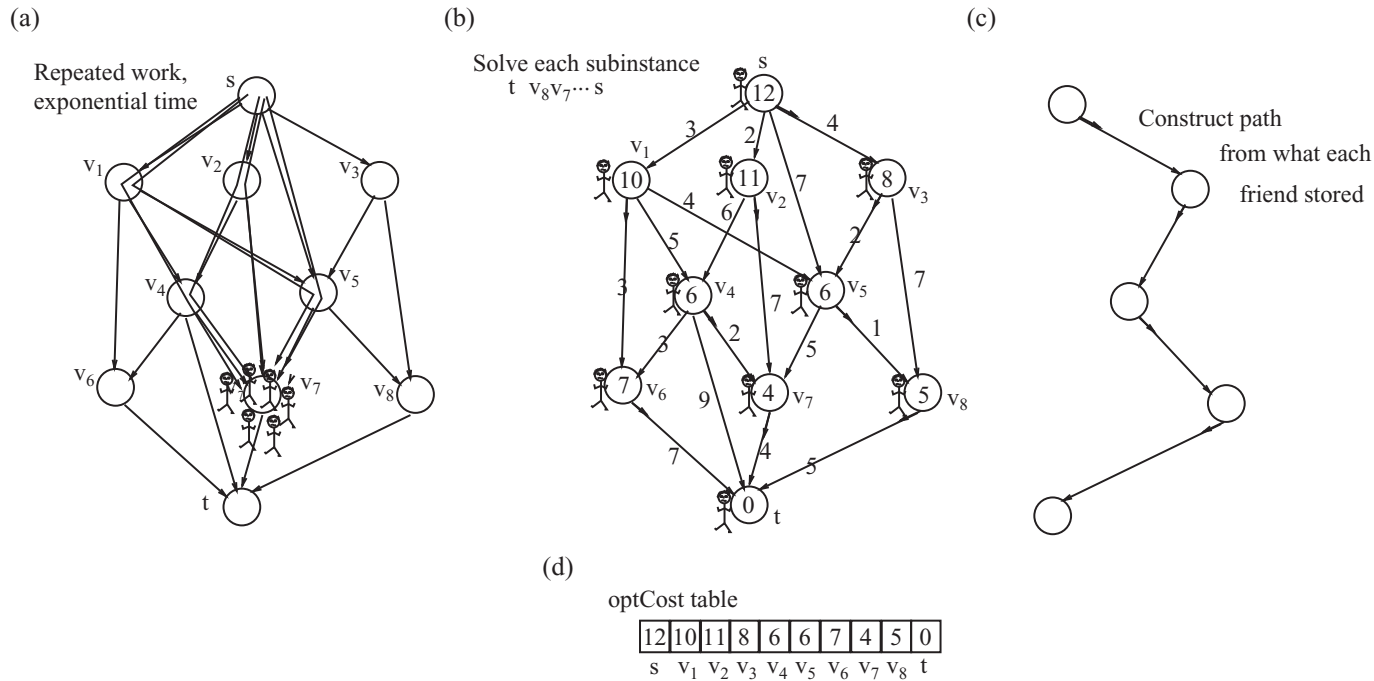


Fig. 24.2 (a) The recursive algorithm. (b) The dynamic programming algorithm: The little arrow out of node v_i indicates the first edge in an optimal path from v_i to t , and the value within the circle on the node gives the cost of this path. (c) The optimal path from s to t . (d) The contents of the *optCost* table. It is filled in backwards.

of work, there is a great deal of redundancy. Different friends are assigned the exact same task. In fact, for each path from s to v_i , some friend is asked to solve the subinstance $\langle G, v_i, t \rangle$. See Figure 24.2(a).

One Friend per Subinstance: To save time, the dynamic programming algorithm solves each of these subinstances only once. We allocate one friend to each of these subinstances, whose job it is to find an optimal solution for it and to provide this solution to any other friend who needs it.

2) Count the Subinstances: The running time of the dynamic programming algorithm is proportional to the number of subinstances. At this point in the design of the algorithm, you should count how many subinstances an instance I has as a function of the size $n = |I|$ of the instance. If there are too many of them, then start at the very beginning, designing a new recursive backtracking algorithm with a different question for the little bird.

Leveled Graph: The number of subinstances in the set $\{\langle G, v_i, t \rangle \mid v_i \text{ above } t\}$ is n , the number of nodes in the graph G .

3) Construct a Table Indexed by Subinstances: The algorithm designer constructs a table. It must have one entry for each subinstance. Generally, the table will have one dimension for each parameter used to specify a particular subinstance. To be consistent, we will always use the letter i and if necessary j to index the subinstances. Each entry in the table is used to store an optimal solution for the subinstance along with its cost. Often we split this table into two tables: one for the solution and one for the cost.

Leveled Graph: The single parameter used to specify a particular subinstance is i . Hence, suitable tables would be $optSol[0 \dots n]$ and $optCost[0 \dots n]$, where $optSol[i]$ will store the best path from node v_i to node t and $optCost[i]$ will store its cost. See Figure 24.2(d).

4) Solution from Subsolutions: The dynamic programming algorithm for finding an optimal solution to a given instance from an optimal solution to a subinstance is identical to that within the recursive backtracking algorithm, except that instead of recursing to solve a subinstance, the algorithm finds its optimal solution in the table.

Leveled Graph: The task of $Friend_i$ in the dynamic programming algorithm is not to solve $\langle G, s, t \rangle$ but $\langle G, v_i, t \rangle$, looking for a best path from v_i to t . They do this as follows. They ask the little bird for the first edge in their path and try each of the bird's possible answers. When the bird suggests the edge $\langle v_i, v_k \rangle$, the friend asks a friend for a best path from v_k to t . This task is the subinstance $\langle G, v_k, t \rangle$, which has been allocated to $Friend_k$ and whose solution has been stored in $optSol[k]$. The recursive backtracking code

$$\langle optSubSol, optSubCost \rangle = LeveledGraph(\langle G, v_k, t \rangle)$$

$$optSol_k = \langle s, v_k \rangle + optSubSol$$

$$optCost_k = w_{\langle s, v_k \rangle} + optSubCost$$

is changed to simply

$$\begin{aligned} \text{optSol}_k &= \langle v_i, v_k \rangle + \text{optSol}[k] \\ \text{optCost}_k &= w_{\langle v_i, v_k \rangle} + \text{optCost}[k]. \end{aligned}$$

Friend_i tacks the bird's edge $\langle v_i, v_k \rangle$ onto this path from v_k to t , giving a path that is the best path from v_i to t among those consistent with the bird's answer $\langle v_i, v_k \rangle$. After trying each bird's answer, *Friend_i* saves the best of these best paths in the table.

5) Base Cases: The base case instances are exactly the same as with the recursive backtracking algorithm. The dynamic programming algorithm starts its computation by storing in the table an optimal solution for each of these and their costs.

Leveled Graph: The only base case is finding a best path from s to t when s and t are the very same node. This only occurs with the subinstance $\langle G, v_n, t \rangle$, where v_n is another name for t . The recursive backtracking code

```
if ( $s = t$ ) then
    return  $\langle \emptyset, 0 \rangle$ 
```

is changed to

```
% Base Case:
optSol[n] =  $\emptyset$ 
optCost[n] = 0
```

6) The Order in Which to Fill the Table: When a friend in the recursive backtracking algorithm needs help from a friend, the algorithm recurses, and the stack frame for the first friend waits until the stack frame for the second friend returns. This forms tree recursive stack frames, keeping track of which friends are waiting for answers from which friends. In contrast, in a dynamic programming algorithm, the friends solve their subinstances in an order such that nobody has to wait. Every recursive algorithm must guarantee that it recurses only on *smaller* instances. Hence, if the dynamic programming algorithm fills in the table from smaller to larger instances, then when an instance is being solved, the solution for each of its subinstances is already available. Alternatively, the algorithm designer can simply choose any order to fill the table that respects the dependences between the instances and their subinstances. The table must be indexed by subinstances. When allocating the table, be clear what subinstance each entry of the table represents.

Leveled Graph: *Friend_i*, with instance $\langle G, v_i, t \rangle$, depends on *Friend_k* when there is an edge $\langle v_i, v_k \rangle$. From the precondition of the problem, we know that each edge must go from a higher level to a lower one, and hence we know that $k > i$. Filling the table in the order $t = v_n, v_{n-1}, v_{n-2}, \dots, v_2, v_1, v_0 = s$ ensures that when *Friend_i* does his work, *Friend_k* has already stored their answer in the table. The subinstance $\langle G, v_n, t \rangle$ has been solved already. The following loop is put around the general case code of the recursive backtracking algorithm:

```
% General Cases: Loop over subinstances in the table.
for  $i = n - 1$  to 0
    % Solve instance  $\langle G, v_i, t \rangle$  and fill in table entry  $\langle i \rangle$ .
```

Viewing the dynamic programming algorithm as an iterative algorithm, the loop invariant when working on a particular subinstance is that all *smaller* subinstances that will be needed have been solved. Each iteration maintains the loop invariant while making progress by solving this next subinstance.

7) The Final Solution: The original instance will be the last subinstance to be solved. When complete the dynamic program simply returns this answer.

Leveled Graph: The original instance $\langle G, s, t \rangle$ is the same as the subinstance $\langle G, v_0, t \rangle$, where v_0 is another name for s . The dynamic program ends with the code

```
return  $\langle \text{optSol}[0], \text{optCost}[0] \rangle$ 
```

8) Code: From steps 1–7, the code can always be put together using the same basic structure:

algorithm *LeveledGraph*(G, s, t)

pre-cond: G is a weighted directed layered graph, and s and t are nodes.

post-cond: optSol is a path with minimum total weight from s to t , and optCost is its weight.

begin

```
% Table:  $\text{optSol}[i]$  stores an optimal path from  $v_i$  to  $t$ , and
 $\text{optCost}[i]$  its cost.
 $\text{table}[0 \dots n] \text{ optSol}, \text{optCost}$ 
```

```
% Base case: The only base case is for the best path from  $t$  to  $t$ .
Its solution is the empty path with cost zero.
```

```
 $\text{optSol}[n] = \emptyset$ 
 $\text{optCost}[n] = 0$ 
```

```
% General cases: Loop over subinstances in the table.
for  $i = n - 1$  to 0
```

```
    % Solve instance  $\langle G, v_i, t \rangle$  and fill in table entry  $\langle i \rangle$ .
```

```
    % Try each possible bird answer.
```

```
    for each of the  $d$  edges  $\langle v_i, v_k \rangle$ 
```

```
        % The bird-and-friend algorithm: The bird tells us that the
        first edge in an optimal path from  $v_i$  to  $t$  is  $\langle v_i, v_k \rangle$ . We ask
        the friend for an optimal path from  $v_k$  to  $t$ . They give us  $\text{optSol}[k]$ ,
        which they had stored in the table. To this we add the bird's edge.
        This gives us  $\text{optSol}_k$  which is a best path from  $v_i$  to  $t$  from
        among those paths consistent with the bird's answer.
```



```

     $optSol_k = \langle v_i, v_k \rangle + optSol[k]$ 
     $optCost_k = w_{\langle v_i, v_k \rangle} + optCost[k]$ 
end for
% Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the best
of these best.
 $k_{min} = \text{a } k \text{ that minimizes } optCost_k$ 
 $optSol[i] = optSol_{k_{min}}$ 
 $optCost[i] = optCost_{k_{min}}$ 
end for
return  $\langle optSol[0], optCost[0] \rangle$ 
end algorithm

```

Consistent Structure: To be consistent, we will always use this same structure for all dynamic programming code. Even when there are small ways that the code could be optimized, we stick to this same structure. Even when different variable names would be more meaningful, we stick to i and if necessary j to index the subinstances, and k to index the bird answers. I believe that this consistency will make it easier for you (and for the marker) to understand the many dynamic programming algorithms.

9) Running Time: We can see that the code loops over each subinstance and, for each, loops over each bird answer. From this, the running time seems to be the number of subinstances in the table times the number K of answers to the bird's question. We will see in [Section 24.3.4](#) that actually the running time of this version of the algorithm is a factor of n bigger than this. The same section will tell how to remove this extra factor of n .

Leveled Graph: The running time of this algorithm is now polynomial. There are only n friends, one for each node in the graph. For the instance $\langle G, v_i, t \rangle$, there is a bird answer for each edge out of its source node v_i . There are at most d of these. The running time is then only $O(n \cdot d)$ times this extra factor of n .

24.3 Subtle Points

Before listing a few of the more subtle points in developing recursive backtracking and a dynamic programming algorithms, I give another example problem.

Example 24.3.1 (Printing Neatly.) Consider the problem of printing a paragraph neatly on a printer. The input text is a sequence of n words with lengths $l_1, l_2, \dots, l_n$, measured in characters. Each printer line can hold a maximum of M characters. Our criterion for neatness is for there to be as few spaces on the ends of the lines as possible.

Preconditions: An instance $\langle M; l_1, \dots, l_n \rangle$ consists of the line length and the word lengths. Generally, M will be thought of as a constant, so we will leave it out when it is clear from the context.

Postconditions: The goal is to split the text into lines in a way that minimizes the cost.

Solutions: A solution for an instance is a list giving the number of words for each line, $\langle k_1, \dots, k_r \rangle$.

Cost of Solution: Given the number of words in each line, the cost of this solution is the sum of the cubes of the numbers blanks on the end of each line (including for now the last line).

Example: Suppose that one way of breaking the text into lines gives 10 blanks on the end of one of the lines, while another way gives 5 blanks on the end of one line and 5 on the end of another. Our sense of esthetics dictates that the second way is “neater.” Our cost heavily penalizes having a large number of blanks on a single line by cubing the number. The cost of the first solution is $10^3 = 1,000$ whereas the cost of the second is only $5^3 + 5^3 = 250$.

Example: Consider printing neatly the silly text “This week has seven dates in it ok” in column with width $M = 11$. This is represented as the printing neatly instance $\langle M; l_1, \dots, l_n \rangle = \langle 11; 4, 4, 3, 5, 5, 2, 2, 2 \rangle$. Three of the possible ways to print this text are as follows:

$\langle k_1, k_2, \dots, k_r \rangle = \langle 2, 2, 2, 2 \rangle$ $\langle k_1, k_2, \dots, k_r \rangle = \langle 1, 2, 2, 3 \rangle$ $\langle k_1, k_2, \dots, k_r \rangle = \langle 2, 2, 1, 3 \rangle$					
This.week..	2^3	This.	7^3	This.week..	2^3
has.seven..	2^3	week.has. .	3^3	has.seven..	2^3
dates.in. .	3^3	seven.dates	0^3	dates.	6^3
it.ok.	6^3	in.it.ok. .	3^3	in.it.ok. .	3^3
Cost	= 259	Cost	= 397	Cost	= 259

Of these three, the first and the last are the cheapest and are likely the cheapest of all the possible solutions.

24.3.1 The Question for the Little Bird

The designer of a recursive backtracking algorithm, a dynamic programming algorithm, or a greedy algorithm must decide which question to ask the little bird. That is, the algorithm designer must decide which sequence of decisions will specify the solution constructed by the algorithm: which things will the algorithm try before backtracking to try something else? This is one of the main creative steps in designing the algorithm.

Local vs. Global Considerations: One of the reasons that optimization problems are difficult is that we are able to make what we call *local* observations and decisions but it is hard to see the *global* consequences of these decisions.

Leveled Graph: Which edge out of s is cheapest is a local question. Which path is the overall cheapest is a global question. We were tempted to follow the

cheapest edge out of the source s . However, sometimes one can arrive at a better overall path by starting with a first edge that is not the cheapest.

Printing Neatly: If we follow a greedy algorithm, we will put as many words on the first line as possible. However, a local *sacrifice* of putting fewer words on this line may lead globally to a better overall solution.

Ask about a Local Property: The question that we ask the bird is about some *local* property of the solution:

First Object: If the solution is a sequence of objects, a good question would be “What is the first object in the sequence?”

Leveled Graph: If the solution is a path though a graph, we might ask, “What is the first edge in the path?”

Printing Neatly: If the solution is a sequence of how many words to put on each line, we ask, “How many words k do we put on the first line?”

Yes or No: If the instance is a sequence of objects and a solution is a subset of these object, a good question would be “Is the first object of the instance included in the optimal solution?”

Event Scheduling: If the solution states which events to schedule, we will ask, “Do we schedule the first event?”

Which Root: If a solution is a binary tree of objects, a good question would be “What object is at the root of the tree?”

The Best Binary Search Tree: Note that the first question is not whether to take the left or right branch of the given binary search tree, but what the root should be when constructing the binary search tree.

In contrast, asking the bird for the number of edges in the best path in the leveled graph is a global, not a local, question.

The Number K of Different Bird Answers: You can only ask the bird a little question. (It is only a *little* bird.) In a little question, the number K of different answers $A_1, A_2, \dots, A_K$ that the bird might give must be small. The smaller K is, the more efficient the final algorithm will be.

Leveled Graph: When asking for an edge out of s , the number K of answers is the degree of the node. This gives a bound on K .

Printing Neatly: K is the maximum number of the first words of the text that would fit on the first line.

Event Scheduling: “Do we schedule the first event?” has $K = 2$ answers, yes and no.

Brute Force: The obvious question to ask the little bird is for them to tell you an entire optimal solution. However, the number of solutions for your instance

I is likely exponential; each solution is a possible answer. Hence, K would be exponential. After getting rid of the bird, the resulting algorithm would be the usual brute force algorithm.

Repeated Questions: Although you want to avoid thinking about it, each of your recursive friends will have to ask their little bird a similar question. Hence, you should choose a question that provides a reasonable follow-up question of a similar form. For example:

- “What is the second object in the sequence?”
- “Is the second object of the instance included in the optimal solution?”
- “What is the root in the left (the right) subtree?”

In contrast, asking the bird for the number of edges in the best path in the leveled graph does not have a good follow-up question.

Reversing the Order: This change is purely for aesthetic reasons. The dynamic program loops backward through the nodes of the graph, $t = v_n, v_{n-1}, v_{n-2}, \dots, v_2, v_1, v_0 = s$. The standard way to do it is to work forward. The recursive backtracking algorithm worked forward from s . The dynamic programming technique reverses the recursive backtracking algorithm by completing the subinstances from smallest to largest. In order to have the final algorithm move forward, the recursive backtracking algorithm needs to go backward. To do this, the little bird should ask something about the end of the solution and not about the beginning. Compare [Figure 24.2](#) and [Figure 24.3](#), and see the final code in [Section 24.3.6](#).

Last Object: “What is the last object in the sequence?” What is the last edge in the optimal path, and how many words do we put on the last line?

Yes or No: “Is the last object of the instance included in the optimal solution?” Do we schedule the last event?

Which Root: We still ask about the root. It is not useful to ask about the leaves.

Exercise 24.3.1 Start over and redevelop the leveled graph dynamic programming algorithm with this new question for the little bird, so that the work is completed starting at the top of the graph.

24.3.2 Subinstances and Subsolutions

Getting a trustworthy answer from the little bird narrows our search problem down to the task of finding the best solution from among those solutions consistent with this answer. It would be great if we could simply ask a friend to find us such a solution; however, we are only allowed to ask our friend to solve subinstances of the original computational problem. Our task within this subsection is to formulate a subinstance to our computational problem such that the search for its optimal solutions somehow parallels our narrowed search task.

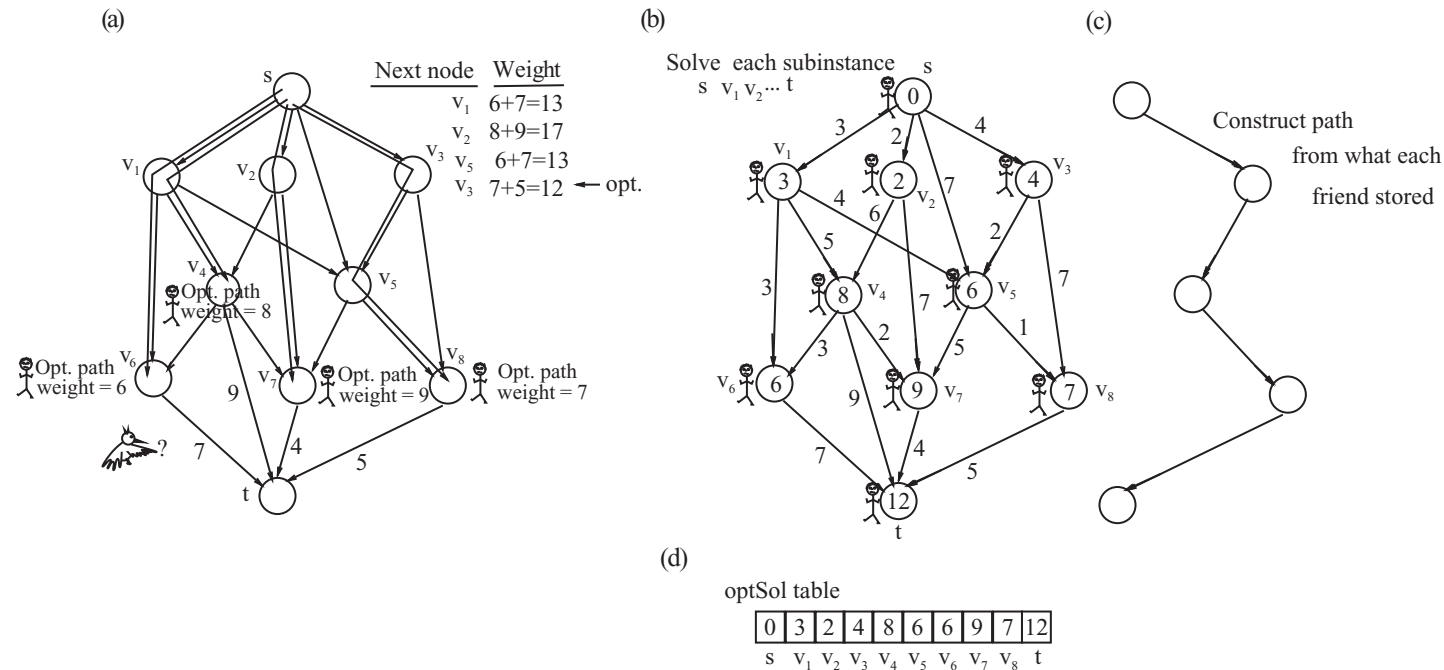


Fig. 24.3 (a) The recursive algorithm. (b) The dynamic programming algorithm: The little arrow out of node v_i indicates the last edge in an optimal path from s to v_i , and the value within the circle at a node gives the cost of this path. (c) The optimal path from s to t . (d) The contents of the *optCost* table. In contrast with that in Figure 24.2, it is filled in forward.

The Recursive Structure of the Problem: In order for us to be able to design a recursive backtracking algorithm for an optimization problem, the problem needs to have a recursive structure. For a solution of the instance to be optimal, some part of the solution must itself be optimal. The computational problem has a recursive structure if the task of finding an optimal way to construct this part of the solution is a subinstance of the same computational problem.

Leveled Graph: For a path from s to t to be optimal, the subpath from some v_i to some v_j along the path must itself be an optimal path between these nodes. The computational problem has a recursive structure because the task of finding an optimal way to construct this part of the path is a subinstance of the same computational problem.

Printing Neatly: In order for all the words to be printed neatly, the words that are on the last ten lines need to be printed neatly on these ten lines.

Question from Answer: Sometimes it is a challenge to know what subinstance to ask our friend. It turns out that it is easier to know what answer (subsolution) we want from him. Knowing the answer we want will be a huge hint as to what the question should be.

Each Solution as a Sequence of Answers: One task that you as the algorithm designer must do is to organize the information needed to specify a solution into a sequence of fields, $sol = \langle field_1, field_2, \dots, field_m \rangle$.

Best Animal: In the zoo problem, each solution consists of an animal, which we will identify with the sequence of answers to the little bird's questions, $sol = \langle \text{vertebrate}, \text{mammal}, \text{cat}, \text{cheetah} \rangle$.

Leveled Graph: In the leveled graph problem, a solution consists of a path, $\langle s, v_1, v_6, t \rangle$, which we will identify with the sequence of edges $sol = \langle \langle s, v_1 \rangle, \langle v_1, v_6 \rangle, \langle v_6, t \rangle \rangle$.

Printing Neatly: A solution for the Printing Neatly problem gives the number of words for each line, $\langle k_1, \dots, k_r \rangle$.

Bird's Question and Remaining Task: The algorithm asks the little bird for the last field $field_m$ of one of the instance's optimal solutions and asks the friend for the remaining fields $\langle field_1, \dots, field_{m-1} \rangle$. We will let k denote the answer provided by the bird, and $optSubSol$ the one provided by the friend. Given both, the algorithm constructs the final solution by simply concatenating these two parts together, namely, $optSol = \langle optSubSol, k \rangle = \langle \langle field_1, \dots, field_{m-1} \rangle, field_m \rangle$.

Leveled Graph: Asking for the last field of an optimal solution $optSol = \langle \langle s, v_1 \rangle, \langle v_1, v_6 \rangle, \langle v_6, t \rangle \rangle$ amounts to asking for the last edge that the path should take. The bird answers $\langle v_6, t \rangle$. The friend provides $optSubSol = \langle \langle s, v_1 \rangle, \langle v_1, v_6 \rangle \rangle$. Concatenating these forms our solution.

Printing Neatly: Asking for the last field of an optimal solution $optSol = \langle k_1, \dots, k_r \rangle$ amounts to asking how many words should go on the last line.

Formulating the Subinstance: We need to find an instance of the computational problem whose optimal solution is $optSubSol = \langle field_1, \dots, field_{m-1} \rangle$. The instance is that whose set of valid solutions is $setSubSol = \{subSol \mid \langle subSol, k \rangle \in setSol\}$.

Leveled Graph: The instance whose solution is $optSubSol = \langle \langle s, v_1 \rangle, \langle v_1, v_6 \rangle \rangle$ is $\langle G, s, v_6 \rangle$, asking for the optimal path from s to v_6 .

Printing Neatly: If the bird told you that an optimal number of words to put on the last line is k , then an optimal printing of the words is an optimal printing of the first $n - k$ words followed by the remaining k words on a line by themselves. Your friend can find an optimal way of printing these first words by solving the subinstance $\langle M; l_1, \dots, l_{n-k} \rangle$. All you need to do then is to add the last k words, i.e., $optSol = \langle optSubSol, k \rangle$.

Costs of Solutions: In addition to finding an optimal solution for the instance I , the algorithm must also produce the cost of this solution. To be helpful, the friend provides the cost of their solution, $optSubSol$. Due to the recursive structure of the problem, the costs of these solutions $optSol = \langle \langle field_1, \dots, field_{m-1} \rangle, field_m \rangle$ and $optSubSol = \langle field_1, \dots, field_{m-1} \rangle$ usually differ in some uniform way. For example, often the cost is the sum of the costs of the fields, that is, $cost(optSol) = \sum_{i=1}^m cost(field_i)$. In this case, we have that $cost(optSol) = cost(optSubSol) + cost(field_m)$.

Leveled Graph: The cost of a path from s to t is the cost of the last edge plus the cost of the rest of the path.

Printing Neatly: The total cost of an optimal solution for the given instance $\langle M; l_1, \dots, l_n \rangle$ is the cost of the optimal solution for the subinstance $\langle M; l_1, \dots, l_{n-k} \rangle$ plus the cube of the number of blanks on the end of the line that contains the last k words, i.e., $optCost = optSubCost + (M - k + 1 - \sum_{j=n-k+1}^n l_j)^3$.

Formal Proof of Correctness:

Recursive Structure of Costs: In order for this recursive backtracking method to solve an optimization problem, the costs that the problem allocates to the solutions must have the following recursive structure. Consider two solutions $sol = \langle subSol, k \rangle$ and $sol' = \langle subSol', k \rangle$ both consistent with the same bird's answer k . If the given cost function dictates that the solution sol is better than the solution sol' , then the subsolution $subSol$ of sol will also be better than the subsolution $subSol'$ of sol' . This ensures that any optimal subsolution of the subinstance leads to an optimal solution of the original instance.

Theorem 24.3.1 *The solution $optSol$ returned is a best solution for I from amongst those that are consistent with the information k provided by the bird.*

Proof: By way of contradiction, assume not. Then there must be another solution *betterSol* consistent with *k* whose cost is strictly better than that for *optSol*. From the way we constructed our friend's subinstance *subI*, this better solution must have the form $\text{betterSol} = \langle \text{betterSubSol}, k \rangle$, where *betterSubSol* is a solution for *subI*. We proved in [Section 11.7](#), using strong induction, that we can trust the friend to provide an optimal solution to the subinstance *subI*. Because the cost of *betterSol* is better than that of *optSol*, it follows that the cost of *betterSubSol* is better than that of *optSubSol*. This contradicts the statement that *optSubSol* is an optimal solution for the subinstance *subI*.

Size of an Instance: In order to avoid recursing indefinitely, the subinstance that you give your friend must be *smaller* than your own instance according to some measure of size. By the way that we formulated the subinstance, we know that its valid solutions $\text{subSol} = \langle \text{field}_2, \dots, \text{field}_m \rangle$ are shorter than the valid solutions $\text{sol} = \langle \text{field}_1, \text{field}_2, \dots, \text{field}_m \rangle$ of the instance. Hence, a reasonable measure of the size of an instance is the length of its longest valid solution. This measure only fails to work when an instance has valid solutions that are infinitely long.

Leveled Graph: The size of the instance $\langle G, s, t \rangle$ is the length of the longest path, or simply the number of levels, between *s* and *t*. Given this, the size of the subinstance $\langle G, s, v_k \rangle$, which is the number of levels between *s* and *v_k*, is smaller.

Each Solution as a Tree of Answers: A few recursive backtracking, dynamic programming, and greedy algorithms have the following more complex structure than in the previous sections. In these, the fields specifying a solution are organized into a tree instead of a sequence. For example, if the problem is to find the best binary search tree, then it is quite reasonable that the fields are the nodes of the tree and these fields should be organized as the tree itself. The algorithm asks the little bird to tell it the field at the root of one of the instance's optimal solutions. One friend is asked to fill in the left subtree, and another the right. See [Sections 27.2](#) and [27.3](#).

24.3.3 The Set of Subinstances

Can Be Difficult: When using the memoization technique to mechanically convert a recursive algorithm into an iterative algorithm, the most difficult step is determining for each input instance the complete set of subinstances that will get called by the recursive algorithm, starting with this instance.

Leveled Graph: We speculated that a subinstance for the leveled graph problem consists of $\{\langle G, v_i, v_j \rangle \mid \text{pair } v_i, v_j\}$, namely, the task of finding the best path between each pair of nodes. Later, by tracing the recursive algorithm, we saw that these subinstances are not all needed, because the subinstances called always search for a path ending in the same fixed node *t*. After changing the bird question to ask about the last edge, all subinstances called always search for a path beginning with the fixed node *s*.

Guess and Check: The technique to find the set of subinstances is to first try to trace out the recursive algorithm on a small example and guess what the set of subinstances will be. Then [Lemma 24.3.2](#) can be used to check if this set is big enough and not too big.

A Set Being Closed under an Operation: We say that the set of even integers is *closed under* addition and multiplication because the sum and the product of any two even numbers is even. In general, we say a set is closed under an operation if applying the operation to any elements in the set results in an element that is also in the set.

The Construction Game: Consider the following game: I give you the integer 2. You are allowed to construct new objects by taking objects you already have and either adding them or multiplying them. What is the complete set of numbers that you are able to construct?

Guess a Set: You might guess that you are able to construct the set of positive even integers. How do you know that this set is big enough and not too big?

Big Enough: Because the set of positive even integers is closed under addition and multiplication, I claim you will never construct an object that is not a positive even number.

Proof: We prove by induction on $t \geq 0$ that after t steps you only have positive even numbers. This is true for $t = 0$, because initially you only have the positive even integer 2. If it is true for t , the object constructed in step $t + 1$ is either the sum or the product of previously constructed objects, which are all positive even integers. Because the set of positive even integers is closed under these operations, the resulting object must also be positive even. This completes the inductive step.

Not Too Big: Every positive even integer can be generated by this game.

Proof: Consider some positive even number $i = 2j$. Initially, we have only 2. We construct i by adding $2 + 2 + 2 + \dots + 2$ a total of j times.

Conclusion: The set of positive even integers accurately characterizes which numbers can be generated by this game, no less and no more.

Lemma 24.3.2 *The set $\mathcal{S}$ will be the complete set of subinstances called starting from our initial instance I_{start} iff.*

1. $I_{\text{start}} \in \mathcal{S}$.
2. $\mathcal{S}$ is closed under the sub operator. ($\mathcal{S}$ is big enough.) The sub operator is defined as follows: Given a particular instance of the problem, applying the sub operator produces all the subinstances constructed from it by a single stack frame of the recursive algorithm.
3. Every subinstance $I \in \mathcal{S}$ can be generated from I_{start} using the sub operator. ($\mathcal{S}$ is not too big.) This ensures that there are not any instances in $\mathcal{S}$ that are not needed. The dynamic programming algorithm will work fine if your set of subinstances contains

subinstances that are not called. However, you do not want the set too much larger than necessary, because the running time depends on its size.

Examples:

The Wedding Invitation List: One of the nightmares when getting married is deciding upon the invitation list. The goal is to make everyone there happy while keeping the number of people small. The three rules in the lemma also apply here.

$I_{start} \in \mathcal{S}$: Clearly the couple getting married needs to be invited.

Closure Ensures That Everyone Is Happy: If you invite Aunt Hilda, then in order to keep her happy, you need to invite her obnoxious son. Similarly, you need to keep all of your subinstances happy, by being sure that for every subinstance included, its immediate friends are also included. In the wedding list problem, you will quickly invite the entire world. The six-degrees-of-separation principle states that the set consisting of your friends' friends' friends' friends' friends' friends includes everyone. Similarly, for most optimization problems the number of subinstances needed tends to be exponential in the size of the instance. A problem has a good dynamic programming algorithm when the number of subinstances is small.

Everyone Needed: We see that everyone on the list must be invited. Even the obnoxious son must come. Because the bride must come, her mother must come. Because her mother must come, Aunt Hilda must come, and hence the son.

Leveled Graph:

Guess a Set: The guessed set is $\{\langle G, s, v_i \rangle \mid v_i \text{ below } s\}$.

Closed: Consider an arbitrary subinstance $\langle G, s, v_i \rangle$ from this set. The sub operator considers some edge $\langle v_k, v_i \rangle$ and forms the subinstance $\langle G, s, v_k \rangle$. This is contained in the stated set of subinstances.

Generating: Consider an arbitrary subinstance $\langle G, s, v_i \rangle$. It will be called by the recursive algorithm if and only if there is a path $\langle v_i, v_{k_1}, v_{k_2}, \dots, v_{k_r}, t \rangle$ from v_i to the original destination t . The initial stack frame on the instance $\langle G, s, t \rangle$, among other things, recurses on $\langle G, s, v_{k_r} \rangle$, which recurses on $\langle G, s, v_{k_{r-1}} \rangle, \dots$, which recurses on $\langle G, s, v_{k_1} \rangle$, which recurses on $\langle G, s, v_i \rangle$.

If the node v_i cannot be reached from node s , then the subinstance $\langle G, s, v_i \rangle$ will never be called by the recursive algorithm. Despite this, we will include it in our dynamic program, because this is not known about v_i until after the algorithm has run.

Printing Neatly: By tracing the recursive algorithm, we see that the set of subinstance used consists only of prefixes of the words, namely, $\{\langle M; l_1, \dots, l_i \rangle \mid i \in [0, n]\}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm, because it contains the initial instance and is closed under the sub operator. Consider an arbitrary subinstance $\langle M; l_1, \dots, l_i \rangle$ from this set. Applying the sub operator constructs the subinstances $\langle M; l_1, \dots, l_{i-w} \rangle$ for $1 \leq w \leq i$, which are contained in the stated set of subinstances.

Generating: Consider the arbitrary subinstance $\langle M; l_1, \dots, l_i \rangle$. We demonstrate that it is called by the recursive algorithm as follows: The initial stack frame on the instance $\langle M; l_1, \dots, l_n \rangle$, among other things, sets w to 1 and recurses on $\langle M; l_1, \dots, l_{n-1} \rangle$. This stack frame also sets w to 1 and recurses on $\langle M; l_1, \dots, l_{n-2} \rangle$. This continues $n - i$ times, until the desired $\langle M; l_1, \dots, l_i \rangle$ is called.

The Number of Subinstances: A dynamic programming algorithm is fast only if the given instance does not have many subinstances.

Subinstance Is a Subsequence: A common reason for the number of subinstances of a given instance being polynomial is that the instance consists of a *sequence* of things rather than a *set* of things. In such a case, each subinstance can be a contiguous (continuous) subsequence of the things rather than an arbitrary subset of them. There are only $O(n^2)$ contiguous subsequences of a sequence of length n , because one can be specified by specifying the two end points. Even better, there are even fewer subinstances if they are defined to be a prefix of the sequence. There are only n prefixes, because one can be specified by specifying the one end point. On the other hand, there are 2^n subsets of a set, because for each object you must decide whether or not to include it.

Leveled Graph: As stated in the definition of the problem, it is easiest to assume that the nodes are ordered so that an edge can go from node v_i to node v_j only if $i < j$. We initially guessed that there were $O(n^2)$ subinstances consisting of subsequences between two nodes, v_i and v_j . We then decreased this to only the $O(n)$ postfixes from the fixed source s to some node v_i . Note that subinstances cannot be subsequences of the instance if the input graph is not required to be leveled.

Printing Neatly: Because the subinstance used consists only of prefixes of the words, namely, $\{\langle M; l_1, \dots, l_i \rangle \mid i \in [0, n]\}$, the number of them is $O(n)$. The single parameter used to specify a particular subinstance is i . Hence, suitable tables would be *birdAdvice*[0... n] and *cost*[0... n]. The *size* of subinstance is simply the number of words, i . Hence, the table is filled in by looping with i from 0 to n .

Reusing the Table: Sometimes you can solve many related instances of the same problem using the same table.

Leveled Graph: The algorithm gives you for free the shortest path from s to each of the nodes.

Printing Neatly: When actually printing text neatly, it does not matter how many spaces are on the end of the very last line. Hence, the cube of this number should not be included in the cost. We could use the original algorithm to find for $k = 1, 2, 3, \dots$ how to print all but the last k words and then put these last k on the last line. However, this would take a total of $O(n \cdot n^2)$ time. Instead, time can be saved by filling in the table only once. One can get the costs for these different instances off this single table. After determining which is best, call *PrintingNeatlyWithAdvice* once to construct the solution for this instance. The total time is reduced to only $O(n^2)$.

24.3.4 Decreasing Time and Space

Recap of a Dynamic Programming Algorithm: A dynamic programming algorithm has two nested loops (or sets of loops). The first iterates through all the subinstances represented in the table, finding an optimal solution for each. When finding an optimal solution for the current subinstance, the second loop iterates through the K possible answers to the little bird's question, trying each of them. Within this inner loop, the algorithm must find a best solution for the current subinstance from among those consistent with the current bird's answer. This step seems to require only a constant amount of work. It involves looking up in the table an optimal solution for a sub-subinstance of the current subinstance and using this to construct a solution for the current subinstance.

Running Time? The running time is clearly the number of subinstances in the table times the number K of answers to the bird's question times what appears (falsely) to be constant time.

Friend-to-Friend Information Transfer: In both a recursive backtracking and a dynamic programming algorithm, information is transferred from subfriend to friend. In recursive backtracking, this information is transferred by returning it from a subroutine call. In dynamic programming, this information is transferred by having the subfriend store the information in the table entry associated with their subinstance and having the friend look this information up from the table. The information transferred is an optimal solution and its cost. The cost, being only an integer, is not a big deal. However, an optimal solution generally requires $\Theta(n)$ characters to write down. Hence, transferring this information requires this much time.

Leveled Graph: In the line of code " $optSol_k = optSol[k] + \langle v_k, v_i \rangle$," $Friend_i$ asks $Friend_k$ for their best path. This path may contain n nodes. Hence, it could take $Friend_i$ $O(n)$ time steps simply to transfer the answer from $Friend_k$.

Time and Space Bottleneck: Being within these two nested loops, this information transfer is the bottleneck on the running time of the algorithm. In a dynamic

programming algorithm, this information for each subinstance is stored in the table for the duration of the algorithm. Hence, this is a bottleneck on the memory space requirements of the algorithm.

Leveled Graph: The total time is $O(n \cdot d \cdot n)$. The total space is $\Theta(n \cdot n)$, being $O(n)$ for each of the n table entries.

A Faster Dynamic Programming Algorithm: We will now modify the dynamic programming algorithm to decrease its time and space requirements. The key idea is to reduce the amount of information transferred.

Cost from Subcost: The subfriends do not need to provide an optimal subsolution in order to find the cost of an optimal solution to the current subinstance. The subfriends need only provide the cost of this optimal subsolution. Transferring only the costs speeds up the algorithm.

Leveled Graph: In order for $Friend_i$ to find the cost of a best path from s to v_i , they need to receive only the best *subcost* from their friends. (See the numbers within the circles in Figure 24.3(b).) For each of the edges $\langle v_k, v_i \rangle$ from their destination node v_i , he learns from $Friend_k$ the cost of a best path from s to v_k . They add the cost of the edge $\langle v_k, v_i \rangle$ to this to determine the cost of a best path from s to v_i from among those that take this edge. Then they determine the cost of an overall best path from s to v_i by taking the best of these best costs.

Note that this algorithm requires $O(n \cdot d)$ – not $O(n \cdot d \cdot n)$ – time, because this best cost can be transferred from $Friend_k$ to $Friend_i$ in constant time. However, this algorithm finds only the cost of the best path; it does not find a best path.

The Little Bird's Advice

Definition of Advice: A friend trying to find an optimal solution to their subinstance asks the little bird a question about this optimal solution. The answer, usually denoted k , to this question classifies the solutions. If this friend had an all-powerful little bird, then that bird could advise the which class of solutions to search in to find an optimal solution. Given that they do not have such a bird, they must simply try all K of the possible answers and determine themselves which answer is best. Either way, we will refer to this best answer as *the little bird's advice*.

Leveled Graph: The bird's advice to $Friend_i$, who is trying to find a best path from s to v_i , is which edge to take last. This edge for each friend is indicated by the little arrows in Figure 24.3(b).

Advice from Cost: Within the algorithm that transfers only the cost of an optimal solution, each friend is able to determine the little bird's advice to them.

Leveled Graph: $Friend_i$ determines, for each of the edges $\langle v_k, v_i \rangle$ into their node, the cost of a best path from s to v_i from among those that take

this edge, and then determines which of these is best. Hence, though this $Friend_i$ never learns a best path from s to v_i in its entirety, they do learn which edge is taken last. In other words, they do determine, even without help from the little bird, what the little bird's advice would be.

Transferring the Bird's Advice: The bird's advice does not need to be transferred from $Friend_k$ to $Friend_i$, because $Friend_i$ does not need it. However, $Friend_k$ will store this advice in the table so that it can be used at the end of the algorithm. This advice can usually be stored in constant space. Hence, it can be stored along with the best cost in constant time without slowing down the algorithm.

Leveled Graph: The advice indicates a single edge. Theoretically, taking $O(\log n)$ bits, this takes more than constant space; practically, however, it can be stored using two integers.

Information Stored in Table: In order to make the dynamic programming algorithm faster, the information stored in the table will no longer be an optimal solution and its cost. Instead, only the cost of an optimal solution and the little bird's advice k are stored.

Leveled Graph: The dynamic programming code in [Section 24.2](#) for *Leveled-Graph* has only two small changes. The line “ $optSol_k = optSol[k] + \langle v_k, v_i \rangle$ ” within the inner loop and the line “ $optSol[i] = optSol_{k_{min}}$,” which stores the optimal solution, are commented out (I recommend leaving them in as comments to add clarity for the reader). The second of these is replaced with the line “ $birdAdvice[i] = k_{min}$,” which stores the bird's advice. See [Section 24.3.6](#) for the new code.

Time and Space Requirements: The running time of the algorithm computing the costs and the bird's advice is

$$\begin{aligned} \text{Time} &= (\text{the number of subinstances indexing your table}) \\ &\quad \times (\text{the number of different answers } K \text{ to the bird's question}). \end{aligned}$$

The space requirement is

$$\text{Space} = \text{the number of subinstances indexing your table.}$$

Leveled Graph: As said, there are n subinstances and a bird answer for each edge out of his source node v_i . Hence, the running time is $O(n \cdot d)$.

Printing Neatly: There are $\Theta(n)$ subinstances in the table, and the number of possible answers for the bird is $\Theta(n)$, because it has the option of telling you pretty well any number of words to put on the last line. Hence, the total running time is $\Theta(n) \cdot \Theta(n) = \Theta(n^2)$, and the space requirement is $\Theta(n)$.

Constructing an Optimal Solution: With these modifications, the algorithm no longer constructs an optimal solution. An optimal solution for the original instance is required, but not for the subinstances. We construct an optimal solution for the instance

using a separate algorithm that is run after the faster dynamic programming algorithm fills the table in with costs and the bird's advice. This new algorithm starts over from the beginning, solving the optimization problem. However, now we know what answer the little bird would give for every subinstance considered. Hence, we can simply run the bird–friend algorithm.

Recursive Bird–Friend Algorithm: The second run of the algorithm will be identical to the recursive algorithm, except now we only need to follow one path down the recursion tree. Each stack frame, instead of branching for each of the K answers that the bird might give, recurses only on the single answer given by the bird. This algorithm runs very quickly. Its running time is proportional to the number of fields needed to represent the optimal solution.

Leveled Graph: A best path from $s = v_0$ to $t = v_n$ is found as follows. Each friend knows which edge is the last edge taken to get to their node. What remains is to put these pieces together by walking backward through the graph, following the indicated directions. See [Figure 24.3\(c\)](#). *Friend_t* knows that the last edge is $\langle v_8, t \rangle$. *Friend₈* knows that the previous one is $\langle v_5, v_8 \rangle$. *Friend₅* knows the edge $\langle v_3, v_5 \rangle$. Finally, *Friend₃* knows the edge $\langle s, v_3 \rangle$. This completes the path.

algorithm *LeveledGraphWithAdvice*($\langle G, s, v_i \rangle, \text{birdAdvice}$)

(pre- & post-cond): Same as *LeveledGraph* except with advice.

```
begin
  if( $s = v_i$ ) then return ( $\emptyset$ )
   $k_{\min} = \text{birdAdvice}[i]$ 
   $\text{optSubSol} = \text{LeveledGraphWithAdvice}(\langle G, s, v_{k_{\min}} \rangle, \text{birdAdvice})$ 
   $\text{optSol} = \text{optSubSol} + \langle v_k, v_i \rangle$ 
  return  $\text{optSol}$ 
end algorithm
```

Exercise 24.3.2 Because the algorithm for leveled graphs with advice only recurses once per stack frame, it is easy to turn it into an iterative algorithm. The algorithm is a lot like a greedy algorithm. In that it always knows which greedy choice to make. Design this iterative algorithm.

24.3.5 Counting the Number of Solutions

The dynamic programming algorithms we have considered so far return one of the possibly many optimal solutions for the given instance. There may be an exponential number of solutions. In this section, we see how to change the algorithms so that they also output the number of possible optimal solutions.

Counting Fruit: If I want to count the number of pieces of fruit in a bowl, I can ask one friend to count for me the red fruit and another the green fruit and another the orange fruit. My answer is the sum of these three. If all the orange fruit is rotten, then I might not include the number of orange fruits in my sum.

Double Counting: To be sure that we do not double-count some optimal solutions, we need the set of solutions consistent with one bird answer to be disjoint from those consistent with another bird answer.

Fruit: If some fruits are half red and half green, these might get double-counted. We want to make sure that color disjointly partitions the fruit according to the answer given.

Leveled Graph: If the bird's answer tells us the first edge of the optimal path, then those paths beginning in the first edge are clearly different than the paths ending in the second.

Computing the Count: In the new dynamic programming algorithm, each friend stores the number of optimal solutions for the subinstance, in addition to the cost of the optimal solution for their subinstance and the bird's advice. This is computed as follows. A friend with one of these subinstances tries each of the K possible bird's answers. When trying k , they find the best solution to his instance from amongst its solutions that are consistent with this bird's answer k by asking a friend of theirs to solve some subinstance. This friend tells them the number of optimal solutions to this subinstance. Let num_k be this number. Generally, there is a one-to-one mapping between our friend's optimal solutions and solutions to their instance that are consistent with this bird's answer k . Hence, he knows that there are num_k of these. As before, let $optCost_k$ be the cost of the best solution to my instance from among its solutions that are consistent with this bird's answer k . The friend computes the cost of their optimal solution simply by $optCost = \max_{k \in [K]} optCost_k$. There may be a number of bird answers that lead to this same optimal cost $optCost$. Each of these lead to optimal solutions. We were careful that the set of solutions consistent with one bird answer is disjoint from that consistent with another bird answer. Hence, the total number of optimal solutions to my instance is $num = \sum_{k \in \{k | optCost_k = optCost\}} num_k$. See [Section 24.3.6](#) for the new leveled graph code.

Bounding the Number of Solutions: Let $Num(n)$ denote the maximum number of possible optimal solutions that any dynamic program as described above might output, given an instance of size n . The number num_k of optimal solutions reported by our friend's friend is at most $Num(n-1)$, because their subinstance has size at most $n-1$. There are at most K bird answers. Hence, $num = \sum_{k \in \{k | optCost_k = optCost\}} num_k$ can be at most $Num(n) = \sum_{k \in [1..K]} Num(n-1) = K \times Num(n-1) = K^n$. Note that it would take exponential time to output these optimal solutions. However, the number of bits to represent this number is only $\log_2(K^n) = \log_2(K) \times n = \Theta(n)$. Hence, this number can be outputted.

24.3.6 The New Code

Three Changes: The dynamic programming algorithm for the leveled graph problem developed in [Section 24.2](#) has been changed in three ways.

Reversing the Order: As described in [Section 24.3.1](#), the algorithm was redeveloped with the little bird being asked for the last edge in the path instead of for the first edge. This has the aesthetic advantage of having the algorithm now branch out forward from s instead of backward from t .

Storing Bird's Advice Instead of Solution: As described in [Section 24.3.4](#), the algorithm now stores the little bird's advice instead of the optimal solution. This is done to decrease the time and space used by the algorithm by a factor of n .

Counting the Number of Solutions: As described in [Section 24.3.5](#), the algorithm, in addition to one of the many optimal solutions for the given instance, now also outputs the number of possible optimal solutions.

Code

algorithm *LeveledGraph*(G, s, t)

(pre-cond): G is a weighted directed layered graph, and s and t are nodes.

(post-cond): $optSol$ is a path with minimum total weight from s to t , and $optCost$ is its weight, and $optNum$ is the number of possible optimal solutions.

begin

 % Table: $optSol[i]$ would store an optimal path from s to v_i ,
 but actually we store only the bird's advice for the subinstance
 and the cost of its solution.

$table[0 \dots n]$ $birdAdvice, optCost, optNum$

 % Base case: The only base case is for the best path from s to s .

 It only has one optimal solution, which is the empty path with cost zero.

 % $optSol[0] = \emptyset$

$optCost[0] = 0$

$birdAdvice[0] = \emptyset$

$optNum[0] = 1$

 % General cases: Loop over subinstances in the table.

 for $i = 1$ to n

 % Solve instance $\langle G, s, v_i \rangle$ and fill in table entry $\langle i \rangle$.

 % Try each possible bird answer.

 for each of the d edges $\langle v_k, v_i \rangle$

 % The bird-and-friend algorithm: The bird tells us that the last
 edge in an optimal path from s to v_i is $\langle v_k, v_i \rangle$. We ask the
 friend for an optimal path from s to v_k . They give us $optSol[k]$,
 which they have stored in the table. To this we add the bird's edge.
 This gives us $optSol_k$, which is a best path from s to v_i from
 amongst those paths consistent with the bird's answer.

 % $optSol_k = optSol[k] + \langle v_k, v_i \rangle$

$optCost_k = optCost[k] + w_{\langle v_k, v_i \rangle}$

 end for

```

% Having the best,  $optSol_k$ , for each bird's answer  $k$ , we keep the
best of these best.
 $k_{min}$  = a  $k$  that minimizes  $optCost_k$ 
%  $optSol[i] = optSol_{k_{min}}$ 
 $optCost[i] = optCost_{k_{min}}$ 
 $birdAdvice[i] = k_{min}$ 
 $optNum[i] = \sum_{k \in \{k | optCost_k = optCost_{k_{min}}\}} optNum[k]$ .
end for
 $optSol = LeveledGraphWithAdvice(\langle G, s, v_n \rangle, birdAdvice)$ 
return  $\langle optSol, optCost[n], optNum[n] \rangle$ 
end algorithm

```

Exercise 24.3.3 (See solution in Exercise Solutions.) Give the code for printing neatly.

Exercise 24.3.4 Consider printing neatly the silly text “This week has seven dates in it OK” in a column with width $M = 11$. This is represented as the Printing Neatly instance $\langle M; l_1, \dots, l_n \rangle = \langle 11; 4, 4, 3, 5, 5, 2, 2, 2 \rangle$. (a) Fill in the $birdAdvice[0 \dots n]$ and $cost[0 \dots n]$ tables for this example. The original instance, its solution, and its cost are in the bottom row. (b) When filling in this last row, give solutions and costs associated with each of the possible bird answers.

Exercise 24.3.5 I saw this puzzle on a Toronto subway. The question is how many times the word “TRAINS” appears in Figure 24.4. Each occurrence of the word must follow a connected path so that each of its letters are adjacent to its previous letter. In order to learn the number of such occurrences, we could count them, but this might be exponential in the number of squares. Instead, for each box do a constant amount of work and write one integer. In the end, the answer should appear in the box with a “T.” You should give a few sentences explaining the order in which you fill the boxes, how you do it, and how much work it is.

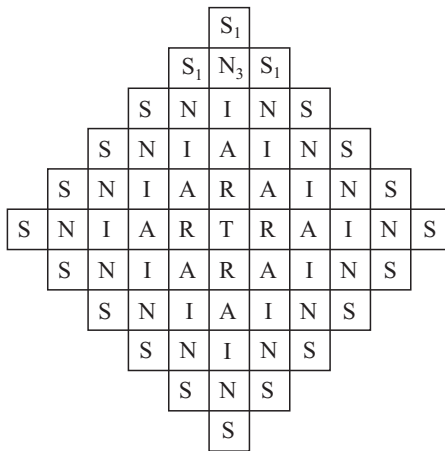


Fig. 24.4 Exercise 24.3.5 searches for “TRAINS” in this figure.

This completes the presentation of the general techniques and the theory behind dynamic programming algorithms. We will now develop algorithms for other optimization problems.

24.4 The Longest-Common-Subsequence Problem

There is a big demand for algorithms that find patterns in strings, for example, DNA. The following optimization problem is called the *longest common subsequence* (LCS).

Longest Common Subsequence

Instances: An instance consists of two sequences, $X = \langle x_1, \dots, x_n \rangle$ and $Y = \langle y_1, \dots, y_m \rangle$. For example, $X = \langle B, D, C, A, B, A \rangle$ and $Y = \langle A, B, C, B, D, A, B \rangle$.

Solutions: A subsequence of a sequence is a subset of the elements taken in the same order. A solution is a subsequence $Z = \langle z_1, \dots, z_l \rangle$ that is common to both X and Y .

Measure of Success: The cost (or success) of a solution is the length of the common subsequence.

Goal: Given two sequences X and Y , the goal is to find the LCS.

Example: $Z = \langle B, C, A \rangle$ is a solution because it is a subsequence of $X = \langle \underline{B}, D, \underline{C}, \underline{A}, B, A \rangle$ and $Y = \langle A, \underline{B}, \underline{C}, B, D, \underline{A}, B \rangle$. The cost (success) of this subsequence is $|Z| = 3$. Here $Z = \langle B, C, B, A \rangle$ would be a longer common subsequence with cost 4.

Greedy Algorithm: Suppose $X = \langle A, B, C, D \rangle$ and $Y = \langle B, C, D, A \rangle$. A greedy algorithm might commit to matching the two A s. However, this would be a mistake, because the optimal answer is $Z = \langle B, C, D \rangle$.

Possible Little Bird Answers: Typically, the question asked of the little bird is for some detail about the end of an optimal solution.

Case $x_n \neq z_l$: Suppose that the bird assures us that the last character of X is not the last character of at least one LCS Z of X and Y . We could then simply ignore this last character of X . We could ask a friend to give us an LCS of $X' = \langle x_1, \dots, x_{n-1} \rangle$ and Y , and this would be an LCS of X and Y .

Case $y_m \neq z_l$: Similarly, if we are told that the last character of Y is not used, then we can ignore it.

Case $x_n = y_m = z_l$: Suppose that the little bird tells us that the last character of both X and Y is the last character of an optimal Z . This, of course, implies that the last characters of X and Y are the same. In this case, we could simply ignore this last character of both X and Y . We could ask a friend to give us a longest

common subsequence of $X' = \langle x_1, \dots, x_{n-1} \rangle$ and $Y' = \langle y_1, \dots, y_{m-1} \rangle$. An LCS of X and Y would be the same, except with the character $x_n = y_m$ tacked on to the end, that is, $Z = Z'x_n$. On the other hand, if the little bird answers this case with $x_n = y_m = z_l$ and we on our own observe that $x_n \neq y_n$, then we know that the little bird is wrong.

Case $z_l = ?$: Even more extreme, suppose that the little bird goes as far as to tell us the last character of an LCS is Z . We could then delete the last characters of X and Y up to and including the last occurrence of this character. A friend could give us an LCS of the remaining X and Y , and then we could add on the known character to give us Z .

Case $x_n = z_l \neq y_m$: Suppose that the bird assures us that the last character of X is the last character of an optimal Z . This will tell us the last character of Z , and hence the last case will apply.

The Question for the Little Bird: We have a number of different answers that the little bird might give, each of which would help us find an optimal Z . We could add even more possible answers to the list. However, the larger the number K of possible answers is, the more work our algorithm will have to do. Hence, we want to narrow this list of possibilities down as far as possible. We will consider only the first three bird answers. This is sufficient because for every possible solution at least one of these is true, namely, either (1) $x_n \neq z_l$, (2) $y_m \neq z_l$, or (3) $x_n = y_m = z_l$. Of course, it may be the case that $x_n \neq z_l$ and $y_m \neq z_l$, but if this is true, then the bird has a choice of whether to answer (1) or (2).

The Set of Subinstances: We guess that the set of subinstances of the instance $\langle \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle \rangle$ is $\{ \langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle \mid i \leq n, j \leq m \}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm, because it contains the initial instance and is closed under the sub operation. Consider an arbitrary subinstance, $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$. Applying the sub operator constructs the subinstances $\langle \langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_{j-1} \rangle \rangle$, $\langle \langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_j \rangle \rangle$, and $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_{j-1} \rangle \rangle$, which are all contained in the stated set of subinstances.

Generating: We know that the specified set of subinstances does not contain subinstances not called by the recursive program, because we can construct any arbitrary subinstance from the set with the sub operator. Consider an arbitrary subinstance $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$. The recursive program on the instance $\langle \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle \rangle$ can recurse on the first option $n - i$ times and then on the second option $m - j$ times. This results in the subinstance $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$.

Constructing a Table Indexed by Subinstances: We now construct a table having one entry for each subinstance. It will have a dimension for each of the parameters i

and j used to specify a particular subinstance. The tables will be $cost[0 \dots n, 0 \dots m]$ and $birdAdvice[0 \dots n, 0 \dots m]$.

Base Cases: The subinstance represented by $cost[0, j]$ is $\langle \emptyset, \langle y_1, \dots, y_m \rangle \rangle$ and has no characters in X . Hence, it is not reasonable to ask the bird about the last character of X . Besides, this is an easy case to handle as a base case. The only subsequence of the empty string is the empty string. Hence, the LCS is the empty string. The cost of this solution is $cost[0, j] = 0$.

Order in Which to Fill the Table: The official order in which to fill the table with subinstances is from smaller to larger. Here the size of the subinstance $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$ is $i + j$. Thus, you would fill in the table along the diagonals. However, the obvious order of looping, for $i = 0$ to n and from $j = 0$ to m , also respects the dependences between the instances and thus could be used instead.

Code

algorithm $LCS(\langle \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle \rangle)$

<pre-cond>: An instance consists of two sequences.

<post-cond>: $optSol$ is an LCS, and $optCost$ is its length.

begin

% Table: $optSol[i, j]$ would store an optimal LCS for $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$, but actually we store only the bird's advice for the subinstance and the cost of its solution.
 $table[0 \dots n, 0 \dots m]$ $birdAdvice$, $cost$

% Base cases: The base cases consist of when one string or the other is empty, i.e., when $i = 0$ or when $j = 0$.

For each, the solution is the empty string with cost zero.

for $j = 0$ to m

% $optSol[0, j] = \emptyset$

$cost[0, j] = 0$

$birdAdvice[0, j] = \emptyset$

end for

for $i = 0$ to n

% $optSol[i, 0] = \emptyset$

$cost[i, 0] = 0$

$birdAdvice[i, 0] = \emptyset$

end for

% General cases: Loop over subinstances in the table.

for $i = 1$ to n

for $j = 1$ to m

% Solve instance $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$, and fill in table entry $\langle i, j \rangle$.

% The bird-and-friend algorithm: The bird tells us either (1)

```

 $x_i \neq z_{l'}$ , (2)  $y_j \neq z_{l'}$ , or (3)  $x_i = y_j = z_{l'}$ . We remove this last
letter (1)  $x_i$ , (2)  $y_j$ , or (3) both and ask the friend for an
optimal LCS for the remaining words. They give us (1)
 $optSol[i - 1, j]$ , (2)  $optSol[i, j - 1]$ , or (3)  $optSol[i - 1, j - 1]$ ,
which they have stored in the table. For cases 1 and 2, we leave
this the same. For case 3, we add on the bird's letter, assuming
that  $x_i = y_j$ . This gives us  $optSol_k$  which is an LCS for  $\langle i, j \rangle$ 
from amongst those printings consistent with the bird's answer.
% Try each possible bird answers.
% Case  $k = 1$ :  $x_i \neq z_{l'}$ 
    %  $optSol_1 = optSol[i - 1, j]$ 
     $cost_1 = cost[i - 1, j]$ 
% Case  $k = 2$ :  $y_j \neq z_{l'}$ 
    %  $optSol_2 = optSol[i, j - 1]$ 
     $cost_2 = cost[i, j - 1]$ 
% Case  $k = 3$ :  $x_i = y_j = z_{l'}$ 
    if  $x_i = y_j$  then
        %  $optSol_3 = optSol[i - 1, j - 1] + x_i$ 
         $cost_3 = cost[i - 1, j - 1] + 1$ 
    else
        % Bird was wrong.
        %  $optSol_3 = ?$ 
         $cost_3 = -\infty$ 
    end if
% end cases
% Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
we keep the best of these best.
 $k_{max} = \text{a } k \in [1, 2, 3] \text{ that maximizes } cost_k$ 
%  $optSol[i, j] = optSol_{k_{max}}$ 
 $cost[i, j] = cost_{k_{max}}$ 
 $birdAdvice[i, j] = k_{max}$ 
end for
end for
 $optSol = LCSWithAdvice(\langle \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_m \rangle \rangle, birdAdvice)$ 
return  $\langle optSol, cost[n, m] \rangle$ 
end algorithm

```

Using Information about the Subinstance: This algorithm only has three different bird answers. Surely, this is good enough. However, the number of the possible answers can be narrowed even further. We know the instance is $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$; hence, without asking the bird, we know whether or not the last characters are the same, that is, whether or not $x_i = y_j$. If $x_i \neq y_j$, the third case with $x_i = y_j = z_{l'}$ is clearly not possible. Conversely, when $x_i = y_j$, one might also wonder whether the case $x_i \neq z_{l'}$ could ever lead to an optimal solution. [Exercise 24.4.1](#) shows that in this case we only

need to consider $x_i = y_j = z_{i'}$. As in a greedy algorithm, we know the answer even before asking the question.

Constructing an Optimal Solution:

algorithm *LCSWithAdvice*($\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle\rangle, \text{birdAdvice}$)

(pre- & post-cond): Same as *LCS* except with advice.

```

begin
  if ( $i = 0$  or  $j = 0$ ) then
     $\text{optSol} = \emptyset$ 
    return  $\text{optSol}$ 
  end if
   $k_{\max} = \text{birdAdvice}[i, j]$ 
  if  $k_{\max} = 1$  then
     $\text{optSubSol} = \text{LCSWithAdvice}(\langle\langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_j \rangle\rangle, \text{birdAdvice})$ 
     $\text{optSol} = \text{optSubSol}$ 
  else if  $k_{\max} = 2$  then
     $\text{optSubSol} = \text{LCSWithAdvice}(\langle\langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_{j-1} \rangle\rangle, \text{birdAdvice})$ 
     $\text{optSol} = \text{optSubSol}$ 
  else if  $k_{\max} = 3$  then
     $\text{optSubSol} = \text{LCSWithAdvice}(\langle\langle x_1, \dots, x_{i-1} \rangle, \langle y_1, \dots, y_{j-1} \rangle\rangle, \text{birdAdvice})$ 
     $\text{optSol} = \langle \text{optSubSol}, x_i \rangle$ 
  end if
  return  $\text{optSol}$ 
end algorithm

```

Time and Space Requirements: See [Exercise 24.4.2](#).

Example: See [Figure 24.5](#).

Exercise 24.4.1 (See solution in Exercise Solutions.) (a) Prove that if $x_i = y_j$, then we only need to consider the third case, and if $x_i \neq y_j$, then we only need to consider the first two cases. (b) Show how this changes the code.

Exercise 24.4.2 (See solution in Exercise Solutions.) Calculate the running time of this algorithm.

24.5 Dynamic Programs as More-of-the-Input Iterative Loop Invariant Algorithms

A dynamic program can be thought of from two very different perspectives: as an optimized recursive backtracking algorithm and as an iterative loop invariant algorithm that fills in a table. From the perspective of an iterative algorithm, the loop invariant maintained is that the previous table entries have been filled in correctly. Progress

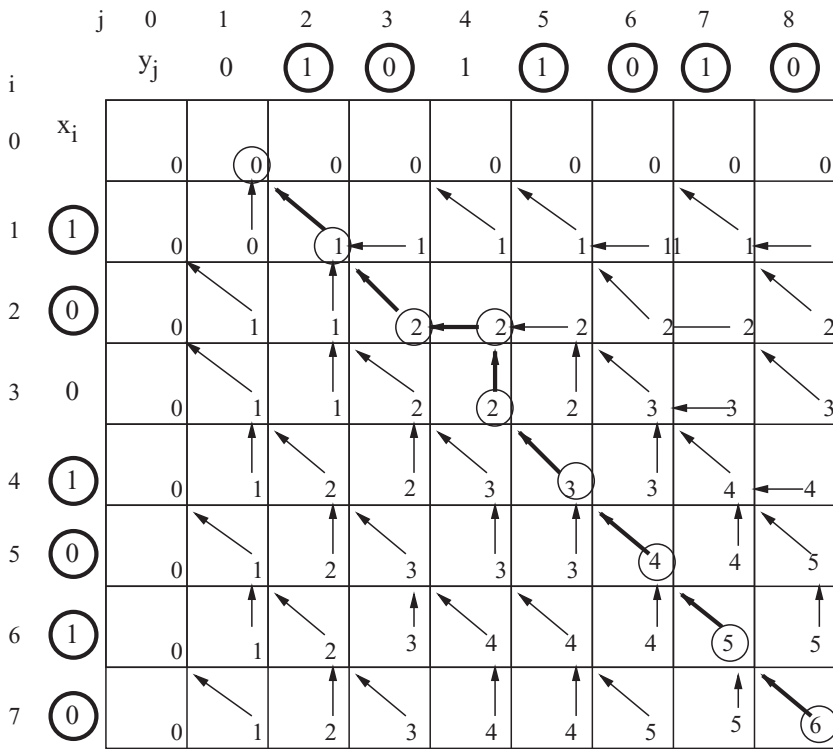


Fig. 24.5 The tables generated for the instance $X = 1001010$ and $Y = 01011010$. Each number is $cost[i, j]$, which is the length of the longest common subsequence of the first i characters of X and the first j characters of Y . The arrow indicates whether the bird's advice is to include $x_i = y_j$, to exclude x_i , or to exclude y_j . The circled digits of X and Y give an optimal solution.

is made while maintaining this loop invariant by filling in the next entry. This is accomplished using the solutions stored in the previous entries.

In *more-of-the-input* iterative algorithms (see [Section 1.2](#)), the subinstances are prefixes of the instance, and hence the algorithm iterates through the subinstances by iterating in some way through the elements of the instance.

Simple Example 24.5.1 (Longest Increasing Contiguous Subsequence)

Suppose that the input consists of a sequence $A[1 \dots n]$ of integers and we want to find the longest contiguous subsequence $A[k_1, k_2]$ such that the elements are monotonically increasing. For example, the optimal solution for $[5, 3, 1, 3, 7, 9, 8]$ is $[1, 3, 7, 9]$.

Longest Block of Ones: The problem is very similar to the longest-block-of-ones problem given in [Section 2.2](#).

Deterministic Nonfinite Automation: The algorithm will read the input characters one at a time. Let $A[1 \dots i]$ denote the subsequence read so far. The loop invariant will be that some information about this prefix $A[1 \dots i]$ is stored. From this information about $A[1 \dots i]$ and the element $A[i + 1]$, the algorithm must be able to determine the

required information about the prefix $A[1 \dots i + 1]$. In the end, the algorithm must be able to determine the solution from this information about the entire sequence $A[1 \dots n]$. Such an algorithm is a deterministic finite automaton (DFA) if only a constant amount of information is stored at each point in time. (See [Section 2.2.](#)) However, in this chapter more memory will be required.

The Algorithm: After reading $A[1 \dots i]$, remember the longest increasing contiguous subsequence $A[k_1 \dots k_2]$ read so far and its size. In addition, so that you know whether the current increasing contiguous subsequence gets to be longer than the previous one, save the longest one ending in the value $A[i]$, and its size.

If you have this information about $A[1 \dots i - 1]$, then you can learn it about $A[1 \dots i]$ as follows. If $A[i - 1] \leq A[i]$, then the longest increasing contiguous subsequence ending in the current value increases in length by one. Otherwise, it shrinks to being only the one element $A[i]$. If this subsequence increases to be longer than our previous longest, then it replaces the previous longest. In the end, we know the longest increasing contiguous subsequence.

The running time is $\Theta(n)$. This is not a DFA, because the amount of space to remember an index and a count (amounting to $\Theta(n)$ values) is $\Theta(\log n)$ bits.

Harder Example 24.5.2 (Longest Increasing Subsequence)

Again the input consists of a sequence A integers of size n . However, now we want to find the longest (not necessarily contiguous) subsequence $S \subseteq [1 \dots n]$ such that the elements, in the order that they appear in A , are monotonically increasing. For example, an optimal solution for $[5, 1, 5, 7, 2, 4, 9, 8]$ is $[1, 5, 7, 9]$, and so is $[1, 2, 4, 8]$.

Dynamic Programming Deterministic Nonfinite Automaton: Again the algorithm will read the input characters one at a time. But now the algorithm will store suitable information not only about the current subsequence $A[1 \dots i]$ but also about each previous subsequence $\forall j \leq i, A[1 \dots j]$. Each of these subsequences $A[1 \dots j]$ will be referred to as a *subinstance* of the original instance $A[1 \dots n]$.

The Algorithm: As before, we will store both the longest increasing sequence seen so far and the longest one(s) that we are currently growing.

The Loop Invariant: Suppose that the subsequence read so far is 10, 20, 1, 30, 40, 2, 50. Then 10, 20, 30, 40, 50 is the longest increasing subsequence so far. A shorter one is 1, 2, 50. The problem is that these end in a relatively large number, so we may not be able to extend them further. If the rest of the string is 3, 4, 5, 6, 7, 8, we will have to have remembered that 1, 2 is the longest increasing subsequence that ends in the value 2. In fact, for many values v , we need to remember the longest increasing subsequence ending in this value v (or a smaller one), because in the end, it may be that many of the remaining elements increase starting from this value. We only need to do this for values v that have been seen so far in the array. Hence, one possibility is to store, for each $j \leq i$, the longest increasing subsequence in $A[1 \dots j]$ that ends with the value $A[j]$.

Maintaining the Loop Invariant: If we have this information for each $j \leq i - 1$, then we learn it about $A[i]$ as follows. For each $j \leq i - 1$, if $A[j] \leq A[i]$, then $A[i]$ can extend the subsequence ending with $A[j]$. Given this construction, the maximum length for i will then be one more than that for j . We get the longest one for i by taking the best of these overall such j . If there is no such j , then the count for i will be 1, namely, simply $A[i]$ itself.

Final Solution: In the end, the solution is the increasing subsequence ending in $A[j]$, where $j \in [1 \dots n]$ is that for which this count is the largest.

Running Time: The time for finding this best subsequence ending in $A[j]$ from which to extend to $A[i]$ would be $\Theta(i)$ if each $j \in [1 \dots i - 1]$ needed to be checked. However, by storing this information in a heap, the best j can be found in $\Theta(\log i)$ time. This gives a total time of $\Theta(\sum_{i=1}^n \log i) = \Theta(n \log n)$ for this algorithm.

Recursive Backtracking: We can also understand this same algorithm from the recursive backtracking perspective. Given the goal of finding the longest increasing subsequence of $A[1 \dots i]$, we might ask the little bird whether or not the last element $A[i]$ should be included. Both options need to be tried. If $A[i]$ is not to be included, then the remaining subtask is to find the longest increasing subsequence of $A[1 \dots i - 1]$. This is clearly a subinstance of the same problem. However, if $A[i]$ is to be included, then the remaining subtask is to find the longest increasing subsequence of $A[1 \dots i - 1]$ that ends in a value that is smaller than or equal to this last value $A[i]$. This is another way of seeing that we need to learn both the longest increasing sequence of the prefix $A[1 \dots i]$ and the longest one ending in $A[i]$.

24.6 A Greedy Dynamic Program: The Weighted Job/Event Scheduling Problem

We revisit the event scheduling problem from [Section 22.2.1](#), now prioritizing the events. Our original greedy algorithm won't work, but dynamic programming will.

The Weighted Event Scheduling Problem: Suppose that many events want to use your conference room. Some of these events are given a higher priority than others. Your goal is to schedule the room in the optimal way.

Instances: An instance is $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_n, f_n, w_n \rangle \rangle$, where $0 \leq s_i \leq f_i$ are the starting and finishing times and w_i the priority weight for n events.

Solutions: A solution for an instance is a schedule S . This consists of a subset $S \subseteq [1 \dots n]$ of the events that don't conflict by overlapping in time.

Measure of Success: The cost (or success) $C(S)$ of a solution S is the sum of the weights of the events scheduled, i.e., $\sum_{i \in S} w_i$.

Goal: The goal of the algorithm is to find the *optimal solution*, that is, that maximizes the total scheduled weight.

Failed Algorithms

Greedy Earliest Finishing Time: The greedy algorithm used in [Section 22.2.1](#) for the unweighted version greedily selects the event with the earliest finishing time f_i . This algorithm fails when the events have weights. The following is a counterexample.

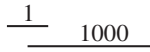


Fig. 24.6

The specified algorithm schedules the top event for a total weight of 1. The optimal schedule schedules the bottom event for a total weight of 1000.

Greedy Largest Weight: Another greedy algorithm selects the first event using the criterion of the largest weight w_i . [Figure 24.7](#) is a counterexample for this.



Fig. 24.7 Counter example for the greedy algorithm that selects largest weight event first.

The top event has weight 2 and the bottom ones each have weight 1. The specified algorithm schedules the top event for a total weight of 2. The optimal schedule schedules the bottom events for a total weight of 9.

Dynamic Programming: The obvious dynamic programming algorithm is for the little bird to tell you whether or not to schedule event J_n .

Bird-and-Friend Algorithm: Consider an instance $J = \langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_n, f_n, w_n \rangle \rangle$. The little bird considers an optimal schedule. We ask the little bird whether or not to schedule event J_n . If they say yes, then the remaining possible events to schedule are those in J , excluding event J_n and excluding all events that conflict with event J_n . We ask a friend to schedule these. Our schedule is theirs with event J_n added. If instead the bird tells us not to schedule event J_n , then the remaining possible events to schedule are those in J excluding event J_n .

The Set of Subinstances: When tracing the recursive algorithm on small examples, we see that the set of subinstances used can be exponentially large. See the left side of [Figure 24.8](#) for an example. The events in the instance are paired so that for $i \in [1 \dots \frac{n}{2}]$, job J_i conflicts with job $J_{\frac{n}{2}+i}$, but jobs between pairs do not conflict. After the little bird tells you whether or not to schedule jobs $J_{\frac{n}{2}+i}$ for $i \in [1 \dots \frac{n}{2}]$, job J_i will remain in the subinstance if and only if job $J_{\frac{n}{2}+i}$ was not scheduled. This results in at least $2^{n/2}$ different paths down the tree of stack frames in the recursive backtracking algorithm, each leading to a different subinstance.

In contrast, look at the instance on the right in [Figure 24.8](#). It only has a linear number of subinstances, because each subinstance is a prefix of the events. The only difference between these examples is the order of the events.

Job numbers								
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>
10	11	12	13	14	15	16	17	18
Substance after committing to keeping 18								
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	
10	11	12	13	14	15	16	17	
Substance after committing to rejecting 17								
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	
10	11	12	13	14	15	16		
Substance after committing to keeping 16								
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>		<u>8</u>	
10	11	12	13	14	15			
Substance after committing to rejecting 15								
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>		<u>8</u>	
10	11	12	13	14				
Substance after committing to keeping 14								
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>		<u>6</u>		<u>8</u>	
10	11	12	13					
Substance after committing to keeping 13								
<u>1</u>	<u>2</u>	<u>3</u>			<u>6</u>		<u>8</u>	
10	11	12						
and so on								

<u>1</u>	<u>3</u>	<u>5</u>	<u>7</u>	<u>9</u>	<u>11</u>	<u>13</u>	<u>15</u>	<u>17</u>
2	4	6	8	10	12	14	16	18
Substance after committing to keeping 18								
<u>1</u>	<u>3</u>	<u>5</u>	<u>7</u>	<u>9</u>	<u>11</u>	<u>13</u>	<u>15</u>	
2	4	6	8	10	12	14	16	
Substance after committing to rejecting 16								
<u>1</u>	<u>3</u>	<u>5</u>	<u>7</u>	<u>9</u>	<u>11</u>	<u>13</u>	<u>15</u>	
2	4	6	8	10	12	14		
Substance after committing to keeping 15								
<u>1</u>	<u>3</u>	<u>5</u>	<u>7</u>	<u>9</u>	<u>11</u>	<u>13</u>		
2	4	6	8	10	12	14		
Substance after committing to rejecting 14								
<u>1</u>	<u>3</u>	<u>5</u>	<u>7</u>	<u>9</u>	<u>11</u>	<u>13</u>		
2	4	6	8	10	12			
Substance after committing to keeping 13								
<u>1</u>	<u>3</u>	<u>5</u>	<u>7</u>	<u>9</u>				
2	4	6	8	10				

Fig. 24.8 Two examples of the subinstances formed from the recursive backtracking algorithm.

Greedy Dynamic Programming: First sort the events in increasing order of their finishing times f_i (a greedy thing to do). Then run the same dynamic programming algorithm in which the little bird tells you whether or not to schedule event J_n .

The Set of Subinstances: When tracing the recursive algorithm on small examples, it looks hopeful that the set of subinstances used is $\{\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_i, f_i, w_i \rangle \rangle \mid i \in [0 \dots n]\}$. If so, the algorithm is polynomial time. The danger is that when excluding those events that conflict with event J_n , the subinstance created will no longer have a contiguous prefix of the events. For example, if event J_2 conflicts with J_n but event J_3 does not, then

the new subinstance will need to be $\langle \langle s_1, f_1, w_1 \rangle, \langle s_3, f_3, w_3 \rangle, \dots, ?? \rangle$, which is not included. Needing to solve this subinstance as well may make the running time exponential.

Closed: For this algorithm, more than those previous, we must be careful to show that this set contains all subinstances generated by the recursive algorithm by showing that it is closed under the sub operation. Consider an arbitrary subinstance $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_i, f_i, w_i \rangle \rangle$ in the set. If we delete from this event J_i and all events that conflict with it, we must show that this new subinstance is again in our set. Let $i' \in [0 \dots i-1]$ be the largest index such that $f_{i'} \leq s_i$. Because the events have been sorted by their finishing time, we know that all events J_k in $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_{i'}, f_{i'}, w_{i'} \rangle \rangle$ also have $f_k \leq s_i$ and hence do not conflict with J_i . All events J_k in $\langle \langle s_{i'+1}, f_{i'+1}, w_{i'+1} \rangle, \dots, \langle s_i, f_i, w_i \rangle \rangle$ have $s_i < f_k \leq f_i$ and hence conflict with J_i . It follows that the resulting subinstance is $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_{i'}, f_{i'}, w_{i'} \rangle \rangle$, which is our set of subinstances. If, on the other hand, only event J_i is deleted, then the resulting subinstance is $\langle \langle s_1, f_1, w_1 \rangle, \dots, \langle s_{i-1}, f_{i-1}, w_{i-1} \rangle \rangle$, which is obviously in our set of subinstances. It is because this works out that the algorithm is polynomial time.

Generating: Consider the arbitrary subinstance $\langle \langle s_1, f_1, w_1 \rangle, \langle s_2, f_2, w_2 \rangle, \dots, \langle s_i, f_i, w_i \rangle \rangle$. It is generated by the recursive algorithm when the little bird states that none of the later events are included in the solution.

The Table: The dynamic programming table is a one-dimensional array indexed by $i \in [0 \dots n]$. The order to fill it in is with increasing i . As in the greedy algorithm, the events are considered to be ordered by earliest finishing time first. The i entry is filled in by trying each of the two answers the bird might give.

Time and Space Requirements: Generally, the running time is the number of subinstances times the number of possible bird answers, and the space is the number of subinstances. This would give $T = \Theta(n \times 2)$ and $S = \Theta(n)$. In this case, however, the running time is larger than that. The reason is that when the event J_i is to be included, it takes $O(\log n)$ time to do a binary search to find which earlier events conflict with it and hence need to be deleted. This gives a running time of $O(n \log n)$, apart from the initial $O(n \log n)$ time for sorting of the activities by finishing time.

Exercise 24.6.1 Write out the pseudocode for this algorithm.

25 Designing Dynamic Programming Algorithms via Reductions

Sometimes, when trying to develop an algorithm for a new problem, it is easier to look for the similarities between your new problem and a problem that you already have an algorithm for. With these insights you can make the new algorithm similar to the old. When done formally, this is called doing a *reduction* from the one problem to another. [Chapter 28](#) covers these ideas in depth. Here we will be looking for similarities between problems more informally. We will start by using this technique to develop a dynamic programming algorithm for a harder version of the event scheduling problem.

Event Scheduling Problem: The problem is to schedule a tour trying to attend the greatest worth of events possible. Unlike the version from [Section 24.6](#), the events occur at different locations. An instance is $\langle E, d \rangle$. Here, $E = \{e_1, e_2, \dots, e_n\}$ is a set of n events. For $j \in [n]$, event e_j is specified by $\langle s_j, f_j, w_j \rangle$, where s_j its start time, f_j its finishing time, and w_j its worth in dollars. The events occur at different locations. For each pair of events e_j and $e_{j'}$, $d_{j,j'}$ is the time (distance) required to travel between the locations of these events. Note that the second parameter d is an n -by- n matrix.

A solution is a schedule of events to attend. This includes which events you attend and the order that you attend them (e.g., $S = \langle e_5, e_{32}, e_{16}, \dots, e_{21} \rangle$). The restriction is that to attend an event, you must get to the location of the event before it starts and you must stay until it completes. More formally, suppose after you attend event E_j , you next attend event $e_{j'}$. Now e_j finishes at time f_j , and $e_{j'}$ starts at time $s_{j'}$. Hence, you have $s_{j'} - f_j$ time to travel the distance between them, when the time required is $d_{j,j'}$. This requires $d_{j,j'} \leq s_{j'} - f_j$. (Note that we assume that the times $d_{j,j'}$ meet the triangle inequality, so that if from event e_j you can reach $e_{j'}$ and from event $e_{j'}$ you can reach $e_{j''}$, then by transitivity from event e_j you can reach $e_{j''}$.)

The worth of a solution is the total of the worths w_j of the events attended. The goal is to maximize the worth of the schedule.

Dynamic Programming: Start by attempting to design a dynamic programming algorithm for this problem. Do not be surprised if you find it hard. This is why we are going to compare this problem with a previously known problem.

Similarity to Best Path: The solution to this problem is a schedule of events, which effectively is a path through a subset of the events. We have looked at many algorithms for finding a best path within graphs, so let us try to model this problem with a graph.

Reduction: Given an instance of $\langle E, d \rangle$ of the scheduling problem, we solve it as follows. We first map it to an instance of the graph problem. Our graph algorithm finds the best path within this graph. Then we map this best path to a solution of the scheduling problem.

Forming a Graph Instance: We now consider how to form a graph representing a given set of events.

Nodes: A solution to the scheduling problem is a *path* of events, and to the graph problem it is a *path* of nodes. This indicates a link between events and nodes. Hence, in the graph, we construct a node for each event e_j .

Edges: In the graph problem, whether or not there is an edge between two nodes indicates whether or not the path can travel from one node to the other. Hence, for each pair of events e_j and $e_{j'}$, we add the directed edge $\langle e_j, e_{j'} \rangle$ if e_j can proceed $e_{j'}$ in the schedule. More formally, $\langle e_j, e_{j'} \rangle$ is an edge if and only if $d_{j,j'} \leq s_{j'} - f_j$. So far, the correspondence between these problems is good, because every path through this graph corresponds to a legal schedule of events and vice versa.

Nodes s and t : The standard problems for finding paths in a graph assume that the input specifies a start node s and a finishing node t , whereas the scheduling algorithm does not specify a first or last event to attend. The standard way to get around this problem is to simply add an extra *start* event e_s and *final* event e_t . Giving event e_s finishing time $f_s = -\infty$ means that imposing the constraint that the schedule starts with e_s does not affect which schedules are legal, because one can get from event e_s to any other event. Giving event e_s worth $w_s = 0$ means that including event s does not change the worth of the final solution. Similarly, let $s_t = \infty$ and $w_t = 0$. From the graph perspective, we add two new nodes s and t and a directed edge from s to every other node and from every node to t .

Costs and Weights: The worth or cost of a path is also different in the two problems. The worth of a schedule is the sum of the worths of the events, which becomes the sum of the weights of the nodes, while the worth or cost of a path through a graph tends to be defined as the sum of the weights of the edges. This difference, however, should not be too significant. One option is to go back to the graph path algorithms and try to get them to work where the nodes, not the edges, have weights. Another option is to simply shift the weight of each event onto the outgoing edges. The edges $\langle e_i, e_j \rangle$ and $\langle e_i, t \rangle$ will have the value v_i of the first event e_i . The edges $\langle s, e_j \rangle$ will have the value zero.

Minimize or Maximize: Another difference between these problems is that the scheduling problem is looking for the schedule of maximum value, whereas the graph problem is looking for the path of minimum value. It turns out that finding the maximum weighted path in a general directed graph is a hard problem. This motivates looking at other structures related to the graph that we know.

The Longest Weighted Path within a Directed Level Graph: Recall the dynamic programming algorithm for finding the shortest weighted path within a

directed level graph that is given in Sections 24.1 and 24.2. Here the nodes can be leveled so that all the edges go forward. It turns out that minimal changes are needed to this algorithm so that it instead finds the longest weighted path within a directed level graph, i.e., solves the same problem but finds a path from s to t with the largest weight. When taking the best of the best, we simply take the max and not the min path. Because there are no cycles, we don't have to worry about paths that cycle in order to have longer weight. Hence, the proof of correctness goes through just as it did before.

Leveled: In order to use this leveled graph algorithm, we need to make sure that the graph that arises from the scheduling problem is in fact leveled. The level of a node e_j can be the start time s_j of the corresponding event. The rules for when edges are added ensure that each edge is directed from some node to a node in a lower level.

Mapping Back the Algorithm: Given this similarity between these two problems, let us recall the dynamic programming algorithm given in Section 24.2 for finding the shortest weighted path within a directed level graph, and let us use it to find a dynamic programming algorithm for the event scheduling problem.

Set of Instances: A key part of a dynamic programming algorithm is the set of subinstances solved. Starting with the instance $\langle G, s, t \rangle$ and looking for the shortest path from s to t , the algorithm finds the shortest path from s to each node v_j , so that the complete set of subinstances solved by the dynamic programming algorithm will be $\{\langle G, s, v_i \rangle \mid v_i\}$. Similarly, for the scheduling problem, given the instance $\langle E, d \rangle$, the set of subinstances will be $\{\langle E, d, i \rangle \mid e_i\}$, where the instance $\langle E, d, i \rangle$ asks for the best schedule of events $S = \langle e_5, e_{32}, e_{16}, \dots, e_{21}, e_i \rangle$ that ends in event e_i . Note that there may be events that occur after event e_i , but the solution schedule is not allowed to attend them.

The Bird-Friend Algorithm: Given an instance $\langle E, d, i \rangle$, we know that the second to last event before e_i in the optimal solution must be an event e_k for which $d_{k,i} \leq s_i - f_k$. We ask the bird to tell us this k . We get the friend to solve the instance $\langle E, d, k \rangle$. Our solution is the same with e_i added on the end. The value of our solution is the same with v_k added in.

The Code: The final code to solve the event scheduling problem is almost identical to that for finding the shortest weighted path within a directed level graph.

algorithm *Schedule*(E, d)

(pre-cond): An instance consists of a set of events $E = \{e_j\}$, with start time s_j , finishing time f_j , worth w_j , and distances $d_{j,j'}$ between them.

(post-cond): *optSol* is an optimal valid schedule of events that ends with event e_n .


```

begin
    Add an imaginary start event  $s = e_0$  with  $s_0 = f_0 = -\infty$ .
    Add an imaginary finishing event  $s = e_{n+1}$  with  $s_{n+1} = f_{n+1} = \infty$ .
    Sort the events by starting time.
    % Table:  $optSol[i]$  stores an optimal schedule of events ending with  $e_i$ 
    and  $costSol[i]$  its cost.
     $table[0..n+1] \text{ } optCost, birdAdvice$ 

    % Base case: The only base case is for the optimal set ending in  $e_0$ .
    Its solution consists of the empty set with cost zero.
    %  $optSol[0] = \emptyset$ 
     $optCost[0] = 0$ 
     $birdAdvice[0] = \emptyset$ 

    % General cases: Loop over subinstances in the table.
    for  $i = 1$  to  $n + 1$ 
        % Solve instance  $\langle E, d, i \rangle$ , and fill in table entry  $\langle i \rangle$ .
        % Try each possible bird answer.
        for each  $k$  for which  $d_{k,i} \leq s_i - f_k$ 
            % The bird–friend algorithm: The last event must be  $e_i$ . The
            bird tells us that the second to last event is  $e_k$ . We ask the
            friend for an optimal set ending in  $e_k$ . They give us  $optSol[k]$ ,
            which they have stored in the table. To this we add  $e_i$ . This gives
            us  $optSol_k$  which is a best solution ending in  $e_i$  from among
            those paths consistent with the bird’s answer.
            %  $optSol_k = optSol[k] + e_i$ 
             $optCost_k = optCost[k] + v_i$ 
        end for
        % Having the best,  $optSol_k$ , for each bird’s answer  $k$ , we keep the
        best of these best.
         $k_{min} = \text{a } k \text{ that maximizes } optCost_k$ 
        %  $optSol[i] = optSol_{k_{min}}$ 
         $optCost[i] = optCost_{k_{min}}$ 
         $birdAdvice[i] = k_{min}$ 
    end for
     $optSol = SchedulingWithAdvice(\langle E, d \rangle, birdAdvice)$ 
    return  $\langle optSol, optCost[n+1] \rangle$ 
end algorithm

```

Running Time: The number of subinstances is $n + 1$, and the number of bird answers is at most $n + 1$. Hence, the running time is $O(n^2)$.

Bigger-Is-Smarter Elephant Problem: We will now consider another problem.

Instances: A set of elephants $E = \{e_1, e_2, \dots, e_n\}$, where $e_i = \langle w_i, s_i, v_i \rangle$ represents the i th elephant, w_i its weight, s_i its intelligence, and v_i its value. (To make life easier, assume that the weights and intelligences are unique values, i.e., $w_i \neq w_j$ and $s_i \neq s_j$.)

Solutions: A subset of elephants $S \subseteq E$ for which bigger is smarter.

Formally, $\forall i, j \in S, [[w_i < w_j] \text{ iff } [s_i < s_j]]$. An equivalent way of looking at it is that if you were to sort the elephants in S in increasing order of their weight, then this same order would sort them with respect to intelligence. (Hint: It is useful to assume that the elephants in the solution are sorted in this way.)

Measure of Success: The cost of the solution is the sum of the values of the elephants, $\sum_{i \in S} v_i$.

Goal: We should find a maximum-valued solution.

Exercise 25.0.1 Design a dynamic programming algorithm for the bigger-is-smarter elephant problem by comparing it, as done previously, with the problem of finding the longest weighted path within a directed level graph problem.

Exercise 25.0.2 (See solution in Exercise solutions.) Design a dynamic programming algorithm for the bigger-is-smarter elephant problem by comparing it with the longest-common-subsequence problem given in [Section 24.4](#). To do this the LCS problem needs to be generalized to have weights on the letters.

26 The Game of Life

Recall that we have already considered two methods of designing such an algorithm. In [Chapter 24](#), we designed a bird algorithm. Trying all bird answers and recursing produced a recursive backtracking algorithm. Memorization produced a dynamic programming algorithm. In [Chapter 28](#), we instead reduced our optimization problem to finding the optimal path through a graph G . To provide an even deeper understanding of dynamic programming, we will now provide a third method of designing such an algorithm. This will involve understanding how to represent a dynamic programming computation for an instance as a graph G and how to represent an optimal solution for it as a path through that graph. With this in mind, we will learn how to think about the computation as walking through the game of life.

26.1 Graph G from Computation

Consider some problem that we already have a dynamic programming algorithm for. Consider a computation of this algorithm on some instance I . From this we will form a graph G such that there is a bijection between st -paths through G and valid solutions Sol for instance I . Moreover, the total edge weight of any path is equal to the value of the corresponding solution. Hence, our goal of finding an optimal solution is reduced to finding the optimal path in G .

Tree from Recursive Backtracking Stack Frames: Consider a recursive backtracking algorithm for our problem P . Consider the tree of stack frames on some instance I formed as the algorithm recursively calls itself. The subinstances $I, subI, subsubI$ are the nodes of the graph and the recursive calls $\langle subI, subsubI \rangle$ are the edges.

Path from Solution: We will now demonstrate a bijection between solutions Sol for our instance I and paths from the root of the stack frame tree to one of the leaves. Suppose we actually have a little bird. Ideally, they are holding an optimal solution $optSol$ for our original instance I ; however, if the bird is not quite as magical as we had hoped, they could be holding any solution Sol . Following all of the bird's answers, follows a path down the tree. For each node/subinstance $subI$ in the path, the friend handling it asks their little bird a question. Each bird answer corresponds to an edge from $subI$. The friend follows the edge corresponding to that given by the bird holding solution Sol . The friend's solution $subS$ to their instance $subI$ is a combination of

the information $subBirdAns$ given to them by their bird and friend's friend's solution $subsubS$ to subinstance $subsubI$, i.e., $subS = \langle subBirdAns, subsubS \rangle$.

Conversely, consider some path from the root to a leaf. At each subinstance/node $subI$ along the path, this friend's bird gives some information $subBirdAns$ about the solution $subS$ the bird is considering when at the followed edge. Combining all of these completely reveals a Sol for our instance I , namely,

$$\begin{aligned}
 Sol &= \langle BirdAns, subS \rangle \\
 &= \langle BirdAns, \langle subBirdAns, subsubS \rangle \rangle \\
 &= \langle BirdAns, \langle subBirdAns, \langle subsubBirdAns, subsubsubS \rangle \rangle \rangle \\
 &= \dots \\
 &= \langle BirdAns, subBirdAns, subsubBirdAns, \dots \rangle.
 \end{aligned}$$

Cost of Path from Cost of Solution: We will now assign a cost/weight to each edge of the tree in such a way that the total cost/weight of any path is exactly equal to the cost of the corresponding solution Sol for our instance I . Just as $S = \langle BirdAns, subS \rangle$, the cost $Cost(S)$ of solution S is computed from the bird's answer $BirdAns$ and the cost $Cost(subS)$ of the friend's solution. Typically, we can define some cost $Cost(BirdAns)$ of the bird's advice $BirdAns$, which is simply added to the friend's cost $Cost(subS)$. Unwinding this gives

$$\begin{aligned}
 Cost(S) &= Cost(BirdAns) + Cost(subS) \\
 &= Cost(BirdAns) + (Cost(subBirdAns) + Cost(subsubBirdAns)) \\
 &= \dots \\
 &= Cost(BirdAns) + Cost(subBirdAns) + Cost(subsubBirdAns) + \dots
 \end{aligned}$$

Graph G from Dynamic Programming Table: Dynamic programming reduces the total computation time using *memoization* to solve each subinstance only once. This changes the tree by merging the nodes that represent the same subinstance. These nodes/subinstances correspond to cells of the table constructed by the dynamic programming algorithm. Let's denote this new collapsed graph as G . It is layered, i.e., a DAG (directed acyclic graph) because all the edges still go down from sub to smaller sub-subinstance.

Path search algorithms are provided a start node s and destination node t . Recall that dynamic programming turns things upside-down by starting at the base cases and ending at the original instance I . Hence, let's create a new node s to be our start node and have edges from it to each of the base cases and let our destination node t be the root node, i.e., that corresponding to our instance I . Paths in the tree from the root to a base case, when turned around, become paths from s to t . Note that such paths still correspond to solutions Sol for our instance I .

26.2 The Graph of Life

Consider some problem that we want to design a dynamic programming algorithm for. Consider some instance I . Imagine a game in which you walk along making decisions about the solution you will produce. You have no idea about how to make these decision, but you ignore this problem. Form a graph whose nodes are the states that you might be in during this computation. These nodes also represent subinstances to the problem in question. For each such node/state/subproblem, the edges leaving it correspond to the action that you might take. Again there is a bijection between st -paths though G and valid solutions Sol for instance I . Our goal of finding an optimal solution is reduced to finding the optimal path in G .

Actions vs. Landmarks: A standard way to think of a computation is as a sequence of actions: Do this. Then do this. It is easy to get lost as to what is true. An alternative way of viewing a computation is as a sequence of landmarks: I am at home. Then at this subway stop. Then school.

Assertions: Preconditions, postconditions, and loop invariants are all assertions. They are NOT actions but static pictures of what is true at this instance in time. Then the code/action must take the computation from the current assertion to the next. For example, one assertion could be that the computation is at line 5 of the code and that i is negative. The action of line 5 could be $i = -i$. The next assertion would then be that the computation is at line 6 of the code and that i is positive.

TM and DFA: In 1936, Allen Turing designed a *Turing Machine* (TM) as the first general model of computation. It has an infinite memory, single-head tape. Without the tape, it is called a Deterministic Finite Automata (DFA). These are defined by a graph. The nodes of the graph represent *states* that a machine might be in at a given point in time, and the edges represent transitions between these states as the machine does actions. I recommend labeling each state with everything the machine knows to be true at this flash in time. I also recommend designing the DFA/TM by compiling it from pseudocode. For example, the machine may be in state $q_{(at\ line\ 5\ of\ code,\ i=5,\ j=4)}$. The action of line 5 is to read the value on the tape into a new variable x . If that value happens to be 8, then the next state the machine will be in will be $q_{(at\ line\ 6\ of\ code,\ i=5,\ j=4,\ x=8)}$. The computation follows a path through this graph of states/nodes.

The number of bits of memory the machine uses is the logarithm of the number of states. This is because if the machine remembers r bits, it must have 2^r states corresponding to the 2^r r -bit strings that it might be remembering. If the pseudocode has 100 lines of code and i, j , and x are variable in the range $[0, 9]$, then the number of states of the form $q_{(at\ line\ ?\ of\ code,\ i=?,\ j=?,\ x=?)}$ is $100 \times 10 \times 10 \times 10$. If variables are in the range $[1, n]$ where n is the size of the input, then the number is $100n^3$. This is not allowed in a DFA or in a TM because the number of states is not allowed to grow with the size of the input; however, we will do it here.

Path of Life: What is life but a path through this graph, i.e., a sequence of snapshot states and/or the sequence of actions/happening that move you from one state to

another? The goal, at each point in time, is to choose which action/edge to take from the current state/node to live an optimal life. This amounts to finding an optimal path in the game of life. Deciding requires not just a local but some global understanding of the graph. For example, locally it might feel that the best action the night before an exam is to party instead of study. But globally, this may not be optimal.

Cutting a Task into Two Independent Subinstances: The goal of the computation is to find an optimal path from s to t . Each node/state v , partitions this problem into two subinstances: that of finding an optimal path from s to v and that of finding an optimal path from v to t .

States and Possible Actions: Your job as the algorithm designer is to define the set of nodes/states that the machine might be in. You won't know which states the machine will follow in its life path so you will define many that it likely will never be in.

For each such node/state, your job is to define the set of actions that the machine might take when in that state. Each such action corresponds to an edge leaving the node/state. Your job is to determine for each such action which node/state the machine will be in next if it were to take that action.

Forgetting: The running time of your algorithm will depend on the number of edges of the graph G . Each node/state is labeled with everything you know when in that state. Knowing a “lot” means that there are lots of different possibilities of what you may have known. Hence, there needs to be lots of states. So the goal is to forget any information that you don't need to continue on. Suppose you are a front-line worker so you know every detail of everything that has ever happened. You need to tell your boss just enough information so that they can make an informed decision about what to do next. Each state $state[i,j]$ is labeled with this information $\langle i,j \rangle$. Specifically, $\langle i,j \rangle$ informs the boss of what they need to know. For example, if i and j are in the range $[1,n]$ and r is an n -bit string, then the number of states of the form $q_{\langle i=?,j=?,r=? \rangle}$ is $n \times n \times 2^n$. Forgetting the value of r reduces the form to $q_{\langle i=?,j=? \rangle}$ and the number to $n \times n$.

Reading the Input I : Because the input I to our problem P is known, fixed, and used to design the graph G , it can be thought as being stored on a tape that the machine always has access to. But the machine can't process it all at once. Hence, for each edge of G the machine traverses, it “reads” one part of the input. The state $state[i,j]$ the machine currently is in likely remembers an index i indicating which part of the input should be read next.

Resource: There is likely some limited resource that gets used up in producing the solution. The state $state[i,j]$ the machine currently is in likely remembers j , indicating how much of the resource has been used.

Produce the Solution Sol : Recall that there is a bijection between paths through G and solutions Sol . As such, there is a bijection between, say, the t^{th} edge in the path and the contents of the t^{th} part of the solution, i.e., for each value this part of the solution might have, there needs to be a different edge. The edges leaving a given state/node correspond to the possible actions the machine might take from there. As such, the action corresponds to outputting this part of the solution. The state $state[i,j]$ the

machine currently is in cannot remember the solution outputted so far because there are an exponential number of solutions and we do not want an exponential number of states.

Edge Cost: Each edge needs to be labeled with a weight. The cost/value of a path is defined to be the sum of the weights of its edges and the cost/value of a solution *Sol* is generally the sum of the costs of its parts. As such, the weight of an edge will be the cost of the part of the solution outputted by the edge. It can also be seen as the incremental cost/value between that of the subsolution for $state[i,j]$ and that for $state[i',j']$.

Little Bird: An interesting debate is *free will* vs. *determinism* vs. *divine intervention*. Deciding which action to take to follow an optimal path through the graph requires not just local but some global understanding of the graph. This decision is beyond the ability of the machine. Strangely, this is beyond you as the designer of the algorithm. (Remember this!) Instead, let's just assume that at each step, the machine can ask a magic little bird which edge to take next. Such a machine is called a **Nondeterministic Machine**.

Goal to Find Optimal Path: There must be a bijection between paths though G and solutions *Sol*. Moreover, the weight of a path must be equal to the value of the corresponding solution. Hence, we find a maximum valued solution by finding a maximum weighted path.

Finding Optimal Path: Finding the maximum weighted path in a graph turns out to require exponential time for a general graph. We, however, require our graph G to be leveled. Hence, our dynamic programming algorithm *LeveledGraph*(G, s, t) can find it.

Running Time: The running time of our leveled graph path algorithm is the number of subinstances times the number of edges $deg(u)$ coming into node u . To be more accurate the time is $\sum_u deg(u)$. This equals the number of edges in the graph.

Edge Length: Consider whether it is best to have the average length of the edges in G be short or be long. Short edges mean that your path from birth to death will traverse lots and lots of short actions. However, we don't care how many edges are in the optimal path. Running time is not the work of traversing this path, but the work of searching for it. The search time depends on the total number of possible edges that we need to search among, whether or not taken.

Long edges are scarier because you don't really know where in the future they are leaping to. The further you go, the more places you can get to, and hence the more edges there are to search.

Suppose, for example, all the edges go from the start node s all the way to the destination t . We would need a different such edge for each st -path and hence for each solution. Because there are an exponential number of solutions to I , this would be an exponential number of edges. Trying them all would amount to the brute force algorithm.

There should be an edge in your graph G for every "atomic" action that takes you from one state to another. Non-atomic actions are not needed because they can be broken into a sequence of atomic actions. An example of doing this is the stock problem in [Section 26.3](#).

26.3 Examples of the Graph of Life

Knapsack: The input consists of a bunch of objects $\langle obj_1, \dots, obj_n \rangle$ each of which has a volume and value. My knapsack has a capacity volume V . My life goal is to choose which objects to put into my knapsack without overfilling it while maximizing the value of the objects taken.

Let's describe this computation as a graph of life G . I walk down the aisle of objects. When I am in state $state[i, v]$, I have already decided (hopefully optimally) which of the first $i-1$ objects to take. I cannot remember what I have taken so far because that would require an exponential number of states. This depends on the path of actions I took to get here. I do remember the volume v of the knapsack remaining. This is the only thing I need to know to be able to make future decisions. At this point, I must decide whether or not to take object obj_i . I take one of the following two possible actions.

With action $k = \text{yes}$, I put it into my knapsack. Following this edge, I get to state $state[i+1, v-v_i]$, because I have now decided about the first i objects and have decreased the volume v remaining by the volume v_i of this object. Though I do not remember the sum value of the objects in my knapsack so far, adding this object adds the value of this object. Hence, we make the weight of this action/edge be this object's value.

In contrast, with action $k = \text{no}$, I do not put the i^{th} object in. Taking this edge, I move to state $state[i+1, v]$ because I have now decided about the first i objects and have not changed the volume v remaining. Because the value of our solution does not change, we make the weight of this action/edge be zero.

A solution does not need to totally fill the knapsack. Hence, there should be an edge with zero weight from each state $state[n+1, v]$ in which there is volume remaining to $state[n+1, 0]$ in which we consider the knapsack full.

Any path from state $s = state[1, V]$ to $t = state[n+1, 0]$ follows some *yes* edges and some *no* ones. In this way, the path indicates for each object whether to take it or not. Hence, as required, there is a bijection between paths through G and solutions Sol . When the path follows a *yes* edge, the value of the object in question is added to the total weight of the path. Hence, the value of the solution to our optimal path instance G , which is equal to the sum of the edge weights of the path, is equal to the sum of the values of the objects taken, which is the value of the solution for our knapsack instance. Hence, our goal is reduced to find the maximum weighted path.

The subinstance corresponding to state $state[i, v]$ could ask which of the first $i-1$ objects to take so that the volume of the knapsack remaining is v and the total value of objects taken is maximized. It could also ask to maximize the value of remaining objects put into the remaining space.

The number of states $state[i, v]$ in this graph is $n \cdot V$, as there are n objects i and V volumes v . Each such state has only two edges/actions. Hence, the running time of this algorithm is $O(nV)$. This looks like polynomial time, but polynomial in what? The size of the input is $O(n) + \log V$ to store the n objects and to write down the value V . The size of the input does not get much bigger when we set $\log V = n$. At this point, the running time is $O(n \cdot V) = O(n \cdot 2^n)$, which is exponential in the number bits to specify

the input. In fact, the knapsack is one of the classic NP-complete problems (see [Chapter 28](#)), meaning that it is not believed to have a polynomial time solution. This time can be decreased to $O(\frac{n^3}{\epsilon})$ if you only need a solution whose value is within a factor $1 + \epsilon$ of being optimal.

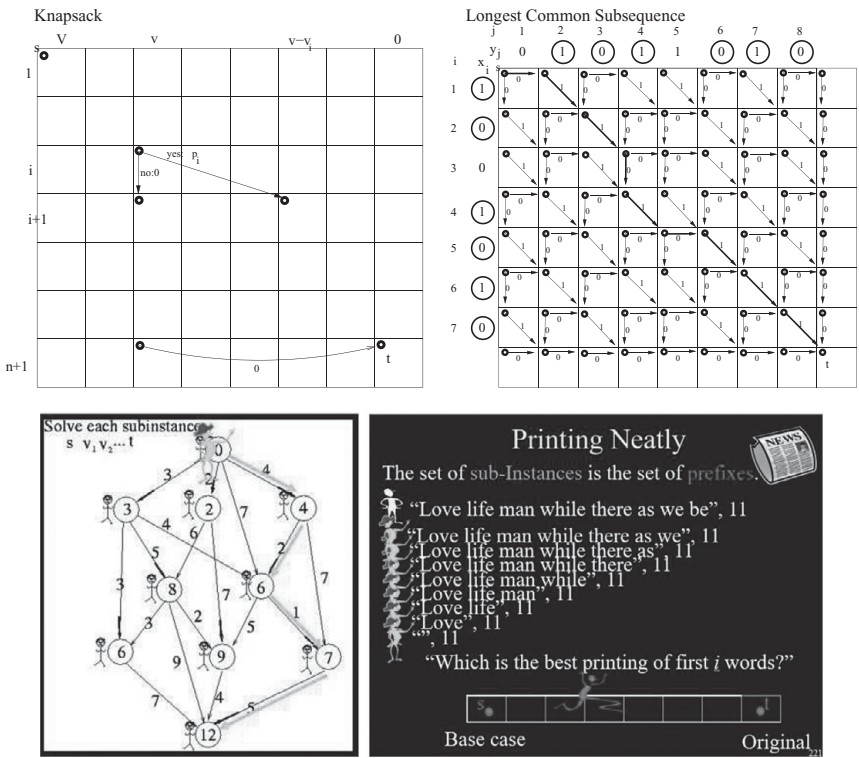


Fig. 26.1 The top left is the “table” for Shortest st -Path Problem. For each node there is a subinstance looking for the shortest path from s to this node. The top right table is that for the Knapsack Problem. The bottom left is for the Longest Common Subsequence Problem. In it, one LCS is circled and the corresponding path has its edges in bold. The bottom right is for Printing Neatly.

Longest Common Subsequence: My life goal is to build a longest common subsequence of the two strings $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_m \rangle$.

Let’s describe this computation as a graph of life G . When I am in state $state[i, j]$, I have already moved my first string’s cursor past $\langle x_1, \dots, x_{i-1} \rangle$ and my second past $\langle y_1, \dots, y_{j-1} \rangle$. Though I have formed a (hopefully longest) common subsequence LCS of what I have moved past, I don’t remember what I have done; this depends on the path of actions I took to get here.

When $x_i \neq y_j$, at least one of these characters cannot be in my LCS. Hence, my action either moves my first cursor past x_i without including it in the solution or my second past y_j . These actions are denoted $k = \downarrow$ and $\rightarrow$ because this can be viewed as the direction in which I am traveling through this matrix of states. The first moves me

to state $state[i+1, j]$ and the second to state $state[i, j+1]$. Though I do not know the length of the LCS formed so far, I do know that this action does not change this length. Hence, the weight of this action/edge is zero.

When $x_i = y_j$, I extend my LCS with this new character and move both cursors past it. This action, denoted $k = \searrow$, moves me to state $state[i+1, j+1]$. The weight of this action/edge is one because whatever my LCS is so far, this action makes it one longer.

Any path from state $s = state[1, 1]$ to $t = state[n+1, m+1]$ adds a character to my LCS every time action $k = \searrow$ is taken. Hence, as required, there is a bijection between paths through G and solutions Sol . The value of a path is equal to the number of $k = \searrow$ actions, which is equal to the length of the solution LCS. Hence, our goal is to find the maximum weighted path.

The subinstance corresponding to state $state[i, j]$ asks to find a longest common subsequence either of the two strings $\langle x_1, \dots, x_{i-1} \rangle$ and $\langle y_1, \dots, y_{j-1} \rangle$ or of the two strings $\langle x_i, \dots, x_n \rangle$ and $\langle y_j, \dots, y_m \rangle$.

The number of states $state[i, j]$ in this graph is nm and each only has two edges/actions. Hence, the running time of this algorithm is $O(nm)$.

Printing Neatly: My life goal is to print a sequence of words neatly in the way that minimizes the sum of the cubes of the number of blanks on the end of each line.

Let's describe this computation as a graph of life G . I print one line of text at a time. Though I don't remember how I did it, when I am in state $state[i]$, I have already printed the first i words completing some number of lines. How I printed them depends on the path of actions I took to get here. My next action to choose is the number of words k to print on the next line. Doing this moves me to state $state[i+k]$. The increase in the cost of the solution is the cube of the number of blanks on the end of this new line. Hence, this is the weight of this action/edge.

Any path from state $s = state[0]$ to $t = state[n]$ prints all n words deciding how many words to print on each line. Hence, as required, there is a bijection between paths through G and solutions Sol .

The subinstance corresponding to state $state[i]$ either asks how to neatly print the first i words or how to print the remaining $n-i$ words.

The number of states $state[i]$ in this graph is the number of words n and each has only ℓ edges/actions where this is the number of words that can fit on a line. Hence, the running time of this algorithm is $O(n\ell)$.

Stock Problem: You are very lucky to have a time machine bring you the value of a set of stocks for each day. The input instance gives the price $Price(t, s)$ of buying one share of stock s on day t . Selling a stock costs an overhead of 3%. Given one dollar on day 1, you will need to determine how you should buy and sell to maximize the amount of cash you have on day T . Because you know exactly what the stocks will do, there is no advantage in owning more than one stock at a time. If all the stocks are doing poorly, you can hold cash for a period. We will first give a $O(T^2S)$ algorithm using one set of nodes/states/subinstances, then a $O(TS^2)$ one using a different set, and then use both types and shorten the edges to represent atomic actions and show that the time is reduced to $O(TS)$.

Algorithm 1 – Maximize Cash on Day t : The computation bumbles along through time buying and selling stocks. Let's suppose that on days that you sell a stock, you do it at time 1:00 p.m. and on days that you buy, you do so at time 1:02 p.m. Let's describe this computation as a graph of life G . For every day t , $state[t]$ is the node/state in which it is 1:01 p.m. on day t and you are holding some amount of cash.

For each time stock s and day t' , a possible action is to buy at 1:02 p.m. as much of stock s as you can with the cash you have and then sell it on day t' at 1:00 p.m. This adds the edge $\langle state[t], state[t'] \rangle$ to graph G . You do not remember the amount of cash that you hold at time t , but let's denote this amount by p . The amount held at time t' would be $p \cdot (1 - 0.03) \cdot \frac{Price(t', s)}{Price(t, s)}$. This accounts for the change in price between these days and the 3% selling costs. Generally, each action/edge adds some fixed amount to the cost/value/weight of the solution, while here it multiplies it by some fixed amount. The effect is that the cost/value of the solution will correspond to the product, not the sum of the edge weights. This won't be a problem, because only a small change is required in the algorithm to return a path with optional product of edge weights. Hence, the cost/value of this edge will be $(1 - 0.03) \cdot \frac{Price(t', s)}{Price(t, s)}$.

Another possible action on day t is to do nothing. This gets you to node/state $state[t+1]$ with the same amount of cash. This adds the edge $\langle state[t], state[t+1] \rangle$ with multiplicative weight 1.

Any path from state $s = state[1]$ to $t = state[T]$ indicates which stocks are bought/sold when. Hence, as required, there is a bijection between paths through G and solutions Sol .

The subinstance corresponding to state $state[t]$ asks either to maximize the amount of cash accumulated from day one up until this this day or to maximize it from this day until time T .

The number of states $state[t]$ in this graph is the number T of days. Each such state has $TS + 1$ edges/actions. Hence, the running time of this algorithm is $O(T^2S)$.

Algorithm 2 – Maximize Stock s at Time t : For each time t and stock s , $state[t, s]$ is the node/state in which it is 1:03 p.m. on day t and you are holding some amount of dollars worth of stock s . To be consistent, we keep all our cost/values units in dollars. For each time stock s' , a possible action is to move to $state[t+1, s']$, i.e., to make sure that you own stock s' at 1:03 p.m. on day $t+1$. If this is a different stock that you hold at time t , then the action requires you to sell stock s and buy s' . The edge will have the resulting multiplicative weight. If they are the same, then you do nothing. The number of states $state[t, s]$ is TS , each with S edges, for a total running time of $O(TS^2)$.

Algorithm 3 – Two Types of States and Atomic Actions: We will now use both types of node/states and shorten the edges to represent atomic actions in order to reduce the time to $O(TS)$.

Suppose it is 1:01 p.m. on day t and you are sitting on node/state $state[t]$ holding cash. Our action in our first algorithm was to buy some stock and then sell it again. We now break this action into two atomic actions.

Sitting on Cash: A possible action is to sit on your cash for another day, moving you to node/state $state[t+1]$. This action does not increase your worth and hence action/edge $\langle state[t], state[t+1] \rangle$ has multiplicative ratio is one.

Buy Stock s : For each time stock s , a possible action is to buy stock s at time 1:02 p.m. so that at 1:03 p.m. you are in state $state[t, s]$. Your value at time 1:03 p.m. is measured in dollars. Hence, this action does not increase your worth and hence the multiplicative ratio of the action/edge $\langle state[t], state[t, s] \rangle$ is one.

Running Time: The number of action/edges leaving each of these T states $state[t]$ decreases from TS to S for a total of $O(TS)$ edge/time.

Suppose it is 1:03 p.m. on day t and you are sitting on node/state $state[t, s]$ holding stock s . A possible action in our second algorithm was to sell this stock and to buy another. We now break this action into two atomic actions.

Sitting on Stock s : A possible action is to do nothing for a day. The market, however, changes the value of your holdings by a factor of $\frac{Price(t+1, s)}{Price(t, s)}$. This corresponds to the action/edge $\langle state[t, s], state[t+1, s] \rangle$ with this weight.

Sell Stock s : Another possible action is to play with your kids until 1:00 p.m. on day $t+1$ and then sell your s stock. This moves you to the state of it being 1:01 p.m. on day $t+1$ and you are sitting on node/state $state[t+1]$ with money. Hence, there is an edge $\langle state[t, s], state[t+1] \rangle$ between these states with weight $(1-0.03) \cdot \frac{Price(t+1, s)}{Price(t, s)}$.

Running Time: The number of action/edges leaving each of these TS states $state[t, s]$ decreases from S to 2 for a total of (TS) edge/time.

As before, there is a bijection between paths from state $s = state[1]$ to $t = state[T]$ and solutions Sol .

27 Solution Is a Tree

When the solution search for is a binary tree, the problem can still be done by dynamic programming, but not by reducing the problem to the shortest path in a leveled graph. The problem is that when the little bird gives information about the root of the solution tree, then the algorithm recurses on two friends, one to find the solution tree's left subtree and one to find its right subtree.

27.1 The Solution Viewed as a Tree: Chains of Matrix Multiplications

We now look at an example in which the fields of information specifying a solution are organized into a tree instead of into a sequence (see [Section 24.3.2](#)). The algorithm asks the little bird to tell it the field at the root of one of the instance's optimal solutions, and then a separate friend will be asked for each of the solution's subtrees.

The optimization problem determines how to optimally multiply together a chain of matrices. Multiplying an $a_1 \times a_2$ matrix by a $a_2 \times a_3$ matrix requires $a_1 \cdot a_2 \cdot a_3$ scalar multiplications. Matrix multiplication is associative, meaning that $(M_1 \cdot M_2) \cdot M_3 = M_1 \cdot (M_2 \cdot M_3)$. Sometimes different bracketing of a sequence of matrix multiplications can lead to the total number of scalar multiplications being very different. For example,

$$\langle 5 \times 1,000 \rangle \cdot \langle 1,000 \times 2 \rangle \cdot \langle 2 \times 2,000 \rangle = \langle 5 \times 2 \rangle \cdot \langle 2 \times 2,000 \rangle = \langle 5 \times 2,000 \rangle$$

requires $5 \times 1,000 \times 2 + 5 \times 2 \times 2,000 = 10,000 + 20,000 = 30,000$ scalar multiplications. However,

$$\begin{aligned} \langle 5 \times 1,000 \rangle \cdot (\langle 1,000 \times 2 \rangle \cdot \langle 2 \times 2,000 \rangle) &= \langle 5 \times 1,000 \rangle \cdot \langle 1,000 \times 2,000 \rangle \\ &= \langle 5 \times 2,000 \rangle \end{aligned}$$

requires $1,000 \times 2 \times 2,000 + 5 \times 1,000 \times 2,000 = 4,000,000 + 10,000,000 = 14,000,000$ scalar multiplications. The problem considered here is to find how to bracket a sequence of matrix multiplications in order to minimize the number of scalar multiplications.

Chains of Matrix Multiplications:

Instances: An instance is a sequence of n matrices $\langle A_1, A_2, \dots, A_n \rangle$. (A precondition is that for each $k \in [1 \dots n-1]$, $\text{width}(A_k) = \text{height}(A_{k+1})$.)

Solutions: A solution is a way of bracketing the matrices, e.g., $((A_1 A_2)(A_3(A_4 A_5)))$. A solution can equivalently be viewed as a binary tree

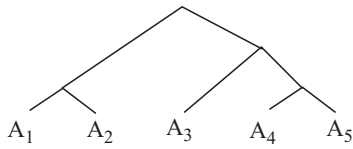


Fig. 27.1 A binary tree giving the order in which to multiply the matrices.

with the matrices $A_1, \dots, A_n$ at the leaves. The binary tree would give the order in which to multiply the matrices:

Measure of Success: The cost of a solution is the number of scalar multiplications needed to multiply the matrices according to the bracketing.

Goal: Given a sequence of matrices, the goal is to find a bracketing that requires the fewest multiplications.

A Failed Greedy Algorithm: An obvious greedy algorithm selects where the last multiplication will occur according to which is cheapest. We can prove that any such simple greedy algorithm will fail, even when the instance contains only three matrices. Let the matrices A_1, A_2 , and A_3 have height and width $\langle a_0, a_1 \rangle$, $\langle a_1, a_2 \rangle$, and $\langle a_2, a_3 \rangle$. There are two orders in which these can be multiplied. Their costs are as follows:

$$\begin{aligned} \text{cost}((A_1 \cdot A_2) \cdot A_3) &= a_0 a_1 a_2 + a_0 a_2 a_3, \\ \text{cost}(A_1 \cdot (A_2 \cdot A_3)) &= a_1 a_2 a_3 + a_0 a_1 a_3. \end{aligned}$$

Consider the algorithm that chooses so that the last multiplication is the cheapest. Let us assume that the algorithm uses the first order. This gives that $a_0 a_2 a_3 < a_0 a_1 a_3$, that is, $a_2 < a_1$. However, the second order will be cheaper if $a_0 a_1 a_2 + a_0 a_2 a_3 > a_1 a_2 a_3 + a_0 a_1 a_3$, that is, if $a_0 >> a_3$. Let us now assign simple values meeting $a_2 < a_1$ and $a_0 >> a_3$. Say $a_0 = 1000, a_1 = 2, a_2 = 1$, and $a_3 = 1$. Plugging these in gives

$$\begin{aligned} \text{cost}((A_1 \cdot A_2) \cdot A_3) &= 1000 \times 2 \times 1 + 1000 \times 1 \times 1 = 2000 + 1000 = 3000, \\ \text{cost}(A_1 \cdot (A_2 \cdot A_3)) &= 2 \times 1 \times 1 + 1000 \times 2 \times 1 = 2 + 2000 = 2002. \end{aligned}$$

This is an instance in which the algorithm gives the wrong answer. Because $1000 < 2000$, it uses the first order. However, the second order is cheaper.

A Failed Dynamic Programming Algorithm: An obvious question to ask the little bird would be which pair of consecutive matrices to multiply together last. Though this algorithm works, it has exponential running time. The problem is that there is an exponential number of different subinstances. Consider paths down the tree of stack frames in which for each pair A_{2i} and A_{2i+1} , the bird either gets us to multiply them together or does not. This results in $2^{n/2}$ different paths down the tree of stack frames in the recursive backtracking algorithm, each leading to a different subinstance.

The Question to Ask the Little Bird: A better question is to ask the little bird to give us the splitting k so that the last multiplication multiplies the product of $\langle A_1, A_2, \dots, A_k \rangle$ and of $\langle A_{k+1}, \dots, A_n \rangle$. This is equivalent to asking for the root of the binary tree. For

each of the possible answers, the best solution is found that is consistent with this answer, and then the best of these best solutions is returned.

Reduced to Subinstance: With this advice, our search for an optimal bracketing is simplified. We need only solve two subinstances: finding an optimal bracketing of $\langle A_1, A_2, \dots, A_k \rangle$ and of $\langle A_{k+1}, \dots, A_n \rangle$. In the example with $\langle A_1, A_2, \dots, A_5 \rangle$ above, the bird splits the problem into subinstances $\langle A_1, A_2 \rangle$ and $\langle A_3, A_4, A_5 \rangle$.

Recursive Structure: An optimal bracketing of the matrices $\langle A_1, A_2, \dots, A_n \rangle$ multiplies the sequence $\langle A_1, \dots, A_k \rangle$ with its optimal bracketing, and $\langle A_{k+1}, \dots, A_n \rangle$ with its optimal bracketing, and then multiplies these two resulting matrices together, that is, $optSol = (optLeft)(optRight)$.

The Cost of the Optimal Solution Derived from the Cost for Subinstances: The total number of scalar multiplications used in this optimal bracketing is the number used to multiply $\langle A_1, \dots, A_k \rangle$, plus the number for $\langle A_{k+1}, \dots, A_n \rangle$, plus the number to multiply the final two matrices. $\langle A_1, \dots, A_k \rangle$ evaluates to a matrix whose height is the same as that of A_1 and whose width is that of A_k . Similarly, $\langle A_{k+1}, \dots, A_n \rangle$ becomes a $height(A_{k+1}) \times width(A_n)$ matrix. Multiplying these requires a number $height(A_1) \times width(A_k) \times width(A_n)$ scalar multiplications. Hence, in total, $cost = costLeft + costRight + height(A_1) \times width(A_k) \times width(A_n)$.

The Set of Subinstances Called: The set of subinstances of the instance $\langle A_1, A_2, \dots, A_n \rangle$ is $\langle A_i, A_{i+1}, \dots, A_j \rangle$ for every choice of end points $1 \leq i \leq j \leq n$. This set of subinstances contains all the subinstances called, because it is closed under the sub operation. Applying the sub operator to an arbitrary subinstance $\langle A_i, A_{i+1}, \dots, A_j \rangle$ from this set constructs subinstances $\langle A_i, \dots, A_k \rangle$ and $\langle A_{k+1}, \dots, A_j \rangle$ for $i \leq k < j$, which are contained in the stated set of subinstances. Similarly, the set does not contain subinstances *not* called by the recursive program, because we easily can construct any arbitrary subinstance in the set with the sub operator. For example, $\langle A_1, \dots, A_n \rangle$ sets $k = j$ and calls $\langle A_1, \dots, A_j \rangle$, which sets $k = i + 1$ and calls $\langle A_i, \dots, A_j \rangle$.

Constructing a Table Indexed by Subinstances: The table indexed by the set of subinstances will have a dimension for each of the parameters i and j used to specify a particular subinstance. The tables will be $cost[1 \dots n, 1 \dots n]$ and $birdAdvice[1 \dots n, 1 \dots n]$. See [Figure 27.2](#).

Order in Which to Fill the Table: The size of a subinstance is the number of matrices in it. We will fill the table in this order.

algorithm *MatrixMultiplication*($\langle A_1, A_2, \dots, A_n \rangle$)

(pre-cond): An instance is a sequence of n matrices.

(post-cond): $optSol$ is a bracketing that requires the fewest multiplications, and $optCost$ is the resulting number of multiplications.

begin

% Table: $optSol[i, j]$ would store an optimal way of bracketing the matrices
 $\langle A_i, A_{i+1}, \dots, A_j \rangle$,

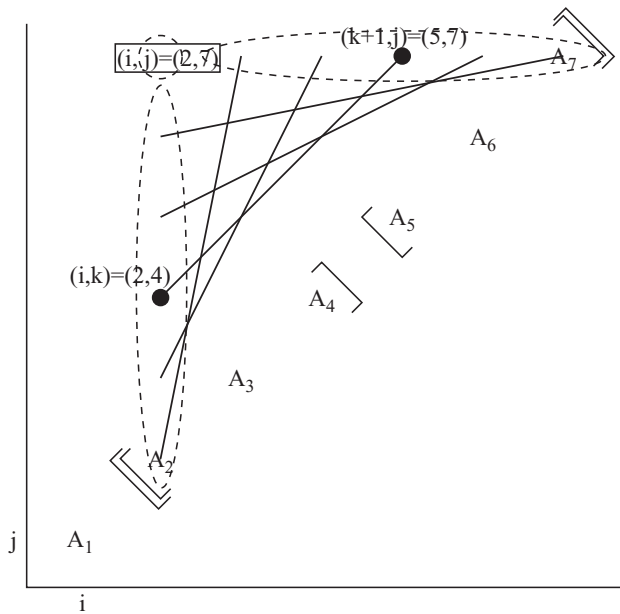


Fig. 27.2 The table produced by the dynamic programming solution for Matrix Multiplication. When searching for the optimal bracketing of $A_2, \dots, A_7$, one of the methods to consider is $[A_2, \dots, A_4][A_5, \dots, A_7]$.

but actually we store only the bird's advice for the subinstance and the cost of its solution.

table[1...*n*, 1...*n*] *birdAdvice*, *cost*

% Base cases: The base cases are when there is only one matrix, i.e., $\langle A_i \rangle$. For each, the solution is the empty bracketing with cost zero.

for $i = 1$ to n

 % *optSol*[*i*, *i*] = A_i

cost[*i*, *i*] = 0

birdAdvice[*i*, *i*] = $\emptyset$

end for

% General cases: Loop over subinstances in the table.

for $size = 2$ to n

 for $i = 1$ to $n - size + 1$

$j = i + size - 1$

 % Solve instance $\langle i, j \rangle$ and fill in the table.

 % Try each possible bird answers.

 for $k = i$ to $j - 1$

 % The bird-and-friend algorithm: The bird gives us the splitting k , so that the last multiplication multiplies the

product of $\langle A_i, A_{i+1}, \dots, A_k \rangle$ and that of $\langle A_{k+1}, \dots, A_j \rangle$. One friend gives us $optSol[i, k]$, an optimal bracketing $\langle A_i, \dots, A_k \rangle$, and another gives us $optSol[k + 1, j]$, an optimal bracketing of $\langle A_{k+1}, \dots, A_j \rangle$. We combine these friends' and the bird's information, obtaining $optSol_k$, which is a best bracketing for $\langle i, j \rangle$ from among those consistent with the bird's answer.

```

% Get help from friend
%  $optSol_k = (optLeft)(optRight)$ 
 $cost_k = cost[i, k] + cost[k + 1, j] + height(A_i)$ 
            $\times width(A_k) \times width(A_j)$ 
end for
% Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
we keep the best of these best.
 $k_{min}$  = a  $k$  that minimizes  $cost_k$ 
%  $optSol[i, j] = optSol_{k_{min}}$ 
 $cost[i, j] = cost_{k_{min}}$ 
 $birdAdvice[i, j] = k_{min}$ 
end for
end for
 $optSol = MatrixMultiplicationWithAdvice(\langle A_1, A_2, \dots, A_n \rangle, birdAdvice)$ 
return  $\langle optSol, cost[1, n] \rangle$ 
end algorithm

```

Constructing an Optimal Solution

algorithm *MatrixMultiplicationWithAdvice*($\langle A_i, A_2, \dots, A_j \rangle$, *birdAdvice*)

<pre- & post-cond>: Same as *MatrixMultiplication* except with advice.

```

begin
  if ( $i = j$ ) then
     $optSol = A_i$ 
    return  $optSol$ 
  end if
   $k_{min} = birdAdvice[i, j]$ 
   $optLeft = MatrixMultiplicationWithAdvice(\langle A_1, \dots, A_{k_{min}} \rangle, birdAdvice)$ 
   $optRight = MatrixMultiplicationWithAdvice(\langle A_{k_{min}+1}, \dots, A_j \rangle, birdAdvice)$ 
   $optSol = (optLeft)(optRight)$ 
  return  $optSol$ 
end algorithm

```

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers, and the space is the number of subinstances. The number of subinstances is $\Theta(n^2)$, and the bird chooses one of $\Theta(n)$ places to split the sequence of matrices. Hence, the running time is $\Theta(n^3)$, and the space requirements are $\Theta(n^2)$.

Exercise 27.1.1 Give the steps to find a counterexample for the greedy algorithm that multiplies the cheapest pair together first.

Exercise 27.1.2 Use a picture to make sure that when $\langle i, j \rangle$ is filled, $\langle i, k \rangle$ and $\langle k + 1, j \rangle$ are already filled for all $i \leq k < j$. Give two other orders that work.

27.2 Generalizing the Problem Solved: Best AVL Tree

As discussed in [Section 11.3](#), it is sometimes useful to generalize the problem solved so that you can either give or receive more information from your friend in a recursive algorithm. This was demonstrated in [Chapter 13](#) with a recursive algorithm for determining whether or not a tree is an AVL tree. This same idea is useful for dynamic programming. I will now demonstrate this by giving an algorithm for finding the best AVL tree. To begin, we will develop an algorithm for the best binary search tree.

The Best Binary Search Tree

Instances: An instance consists of n probabilities $p_1, \dots, p_n$ to be associated with the n keys $a_1 < a_2 < \dots < a_n$. The values of the keys themselves do not matter. Hence, we can assume that $a_i = i$.

Solutions: A solution for an instance is a binary search tree containing the keys. A binary search tree is a binary tree such that the nodes are labeled with the keys and for each node all the keys in its left subtree are smaller and all those in the right are larger.

Measure of Success: The cost of a solution is the expected depth of a key when choosing a key according to the given probabilities, namely $\sum_{i \in [1..n]} [p_i \cdot \text{depth of } a_i \text{ in tree}]$.

Goal: Given the keys and the probabilities, the goal is to find a binary search tree with minimum expected depth.

Expected Depth: The time required to search for a key is proportional to the depth of the key in the binary search tree. Finding the root is fast. Finding a deep leaf takes much longer. The goal is to design the search tree so that the keys that are searched for often are closer to the root. The probabilities $p_1, \dots, p_n$, given as part of the input, specify the frequency with which each key is searched for; e.g., $p_3 = \frac{1}{8}$ means that key a_3 is searched for on average one out of every eight times.

One minimizes the depth of a binary search tree by making it completely balanced. Having it balanced, however, dictates the location of each key. Although having the tree partially unbalanced increases its overall height, it may allow for the keys that are searched for often to be placed closer to the top.

We will manage to put some of the nodes close to the root, and others we will not. The standard mathematical way of measuring the overall success of putting more likely keys closer to the top is the expected depth of a key when the key is chosen randomly

according to the given probability distribution. It is calculated by $\sum_{i \in [1 \dots n]} p_i \cdot d_i$, where d_i is the depth of a_i in the search tree.

One way to understand this is to suppose that we needed to search for a billion keys. If $p_3 = \frac{1}{8}$, then a_3 is searched for on average one out of every eight times. Because we are searching for so many keys, it is almost certain that the number of times we search for this key is very close to $\frac{1}{8}$ billion. In general, the number of times we search for a_i is p_i billion. To compute the average depth of these billion searches, we sum their depths and divide by a billion, namely, $\frac{1}{10^9} \sum_{k \in [1 \dots 10^9]} [\text{depth of } k\text{th search}] = \frac{1}{10^9} \sum_{i \in [1 \dots n]} (p_i \times 10^9) \cdot d_i = \sum_{i \in [1 \dots n]} p_i \cdot d_i$.

Bird-and-Friend Algorithm: I am given an instance consisting of n probabilities $p_1, \dots, p_n$. I ask the bird which key to put at the root. They answer a_k . I ask one friend for the best binary search tree for the keys $a_1, \dots, a_{k-1}$ and its expected depth. I ask another friend for the best tree of the specified height for $a_{k+1}, \dots, a_n$ and its expected depth. I build the tree with a_k at the root and these as the left and right subtrees.

Generalizing the Problem Solved: A set of probabilities $p_1, \dots, p_n$ defining a probability distribution should have the property that $\sum_{i \in [1 \dots n]} p_i = 1$. However, we will generalize the problem by removing this restriction. This will allow us to ask our friend to solve subinstances that are not officially legal. Note that the probabilities given to the friends in the above algorithm do not sum to 1.

The Cost of an Optimal Solution Derived from the Costs for the Subinstances: The expected depth of my tree is computed from that given by my friend as follows.

$$\text{Cost} = \sum_{i \in [1 \dots n]} [p_i \cdot \text{depth of } a_i \text{ in tree}] \quad (27.1)$$

$$= \sum_{i \in [1 \dots k-1]} [p_i \cdot (\text{depth of } a_i \text{ in left subtree}) + 1] + [p_k \cdot 1] \quad (27.2)$$

$$+ \sum_{i \in [k+1 \dots n]} [p_i \cdot (\text{depth of } a_i \text{ in right subtree}) + 1] \quad (27.3)$$

$$= \text{Cost}_{\text{left}} + \left[\sum_{i \in [1 \dots k-1]} p_i \right] + p_k + \text{Cost}_{\text{right}} + \left[\sum_{i \in [k+1 \dots n]} p_i \right] \quad (27.4)$$

$$= \text{Cost}_{\text{left}} + \left[\sum_{i \in [1 \dots n]} p_i \right] + \text{Cost}_{\text{right}} \quad (27.5)$$

$$= \text{Cost}_{\text{left}} + \text{Cost}_{\text{right}} + 1. \quad (27.6)$$

The Complete Set of Subinstances That Will Get Called: The complete set of subinstances is $S = \{ \langle a_i, \dots, a_j; p_i, \dots, p_j \rangle \mid 1 \leq i \leq j \leq n \}$. The table is two-dimensional with size $\Theta(n \times n)$.

Running Time: The table has size $\Theta(n \times n)$. The bird can give n different answers. Hence, the time is $\Theta(n^3)$.

We now change the problem so that it is looking for the best AVL search tree.

The Best-AVL-Tree Problem

Instances: An instance consists of n probabilities $p_1, \dots, p_n$ to be associated with the n keys $a_1 < a_2 < \dots < a_n$.

Solutions: A solution for an instance is an AVL tree containing the keys. An AVL tree is a binary search tree with the property that every node has a balance factor of -1 , 0 , or 1 . Its *balance factor* is the difference between the heights of its left and its right subtrees.

Measure of Success: The cost of a solution is the expected depth of a key, $\sum_{i \in [1..n]} [p_i \cdot \text{depth of } a_i \text{ in } T]$.

Goal: Given the keys and the probabilities, the goal is to find an AVL tree with minimum expected depth.

Cannot Coordinate Friends: We could simply ask friends to build the left and right sub-AVL-trees, but then what would we do if the difference in their heights were greater than one? We cannot expect friends to coordinate their answers.

The New Generalized Problem: An instance consists of the keys, the probabilities, and a required height. The goal is to find the best AVL tree with the given height.

Exercise 27.2.1 (See solution in Exercise Solutions.)

1. What are the possible heights for the left and the right subtrees of an AVL tree of height h ?
2. What question would you ask the bird? It is OK. to ask two questions. What subinstance would you give your friend?
3. How would you ensure the balance between the heights of the left and right subtrees?
4. What is the complete set of subinstances that will get called?
5. What is the running time of your algorithm?
6. In the original problem, the height was not fixed. How would you use the table to solve this problem?

27.3 All Pairs Using Matrix Multiplication

In the following exercises, you will develop a dynamic programming algorithm referred to as the Bellman–Ford algorithm.

Exercise 27.3.1 Let $G = (V, E)$ be a (directed or undirected) graph, and $k \leq n$ some integer. Let M^k be a matrix with both a row and a column for each node in the graph, such that for each pair of nodes $u, v \in V$ the element $M^k[u, v]$ gives the number of distinct paths from u to v that contain at most k edges. Here a path may visit a node more than once. $M^1[u, v]$ is one if there is an edge $\langle u, v \rangle$ and zero otherwise, and $M^1[u, u] = 1$ because there is a path of length zero from u to u . Prove that $M^{i+j} = M^i M^j$, where $\times$ is standard matrix multiplication, i.e., $M^{i+j}[u, v] = \sum_w [M^i[u, w] \cdot M^j[w, v]]$.

Exercise 27.3.2 Now let the graph $G = (V, E)$ have weights $w_{u,v}$ (positive or negative) on each edge. Redefine $\hat{M}^k[u, v]$ to give the weight of the shortest path from u to v with the smallest total weight from amongst these paths that contains at most k edges. Note

that $\hat{M}^1[u, v]$ is the weight of the edge $w_{u,v}$ (or infinity), and $\hat{M}^1[u, u] = 0$ because there is a path of length zero and weight zero from u to u . Prove that $\hat{M}^{i+j} = \hat{M}^i \times \hat{M}^j$, where $\times$ is standard matrix multiplication except that scalar multiplication is changed to $+$ and that $+$ is changed to Min , i.e., $\hat{M}^{i+j}[u, v] = \text{Min}_w[\hat{M}^i[u, w] + \hat{M}^j[w, v]]$. Compare this exercise with [Exercise 4.4.3](#).

Exercise 27.3.3 Note that these exercises are in the tree solution dynamic programming chapter because that is what they are. Your job is to solve subinstance $\hat{M}^{i+j}[u, v]$. Your little bird knows a minimum weighted path from node u to v containing $i+j$ edges. The bird tells you a node w . What property should this node w have? You will have two recursive friends. Which subinstances do you give them? How do you combine your bird's and your friends' answers to get your answer? If you do not have a little bird, what do you do instead?

Exercise 27.3.4 In the above exercises, we go from paths with i and j edges to paths with $i+j$ edges. Suppose your goal is to form the matrix $\hat{M}^N[u, v]$ for some large N . How would you schedule these steps and how many of them would be needed as a function of N ?

Exercise 27.3.5 The standard algorithm for standard matrix multiplication takes $\Theta(n^3)$ time. Strassen's algorithm ([Section 12.2](#)) is able to do it in $\Theta(n^{2.8073})$ time. Does this same algorithm work for this strange multiplication? The equations $x \cdot (y + z) = x \cdot y + x \cdot z$ and $(x - y) + y = x$ are true for all real numbers. Are they true on replacing $\cdot$ with $+$ and replacing $+$ with Min ?

27.4 Parsing with Context-Free Grammars

In [Chapter 15](#) we developed an elegant recursive algorithm for parsing a string according to a given context-free grammar that works only for look ahead one grammars. We now develop a dynamic programming algorithm that works for any context-free grammar.

Given a grammar G and a string s , the first step in parsing is to convert the grammar into one in *Chomsky normal form*, which is defined below. Although a dynamic program could be written to work directly for any context-free grammar, it runs much faster if the grammar is converted first.

The Parsing Problem

Instance: An instance consists of $\langle G, T_{\text{start}}, s \rangle$, where G is a grammar in Chomsky normal form, T_{start} is the nonterminal of G designated as the start symbol, and s is the string $\langle a_1, \dots, a_n \rangle$ of terminal symbols to be generated. The grammar G consists of a set of nonterminal symbols $V = \langle T_1, \dots, T_{|V|} \rangle$ and a set of rules $\langle r_1, \dots, r_m \rangle$. The definition of Chomsky normal form is that each rule r_q has one of the following three forms:

- $A_q \Rightarrow B_q C_q$, where A_q, B_q , and C_q are nonterminal symbols.
- $A_q \Rightarrow b_q$, where b_q is a terminal symbol.

- $T_{start} \Rightarrow \epsilon$, where T_{start} is the start symbol and ϵ is the empty string. This rule may only be used to parse the string $s = \epsilon$. It may not be used within the parsing of a larger string.

Solution: A solution is partial parsing P , consisting of a tree. Each internal node of the tree is labeled with a nonterminal symbol; the root is labeled with the specified symbol T_{start} . Each internal node must correspond to a rule of the grammar G . For example, for the rule $A \Rightarrow BC$, the node is labeled A and its two children are labeled B and C . In a complete parsing, each leaf of the tree is labeled with a terminal symbol. In a partial parsing, some leaves may still be labeled with nonterminals.

Measure of Success: A parsing P is said to generate the string s if the leaves of the parsing in order form s . The cost of P will be one if it generates the string s , and will be zero otherwise.

Goal: The goal of the problem is, given an instance $\langle G, T_{start}, s \rangle$, to find a parsing P that generates s .

Not Look Ahead One: The grammar G might not be look-ahead-one. For example, in

$$A \Rightarrow BC$$

$$A \Rightarrow DE$$

you do not know whether to start parsing the string as a B or a D . If you make the wrong choice, you have to back up and repeat the process. However, this problem is a perfect candidate for a dynamic programming algorithm.

The Parsing Abstract Data Type: We will use the following abstract data type to represent parsings. Suppose that there is a rule $r_q = "A_q \Rightarrow B_q C_q"$ that generates B_q and C_q from A_q . Suppose as well that the string $s_1 = \langle a_1, \dots, a_k \rangle$ is generated starting with the symbol B_q using the parsing P_1 (B_q is the root of P_1) and that $s_2 = \langle a_{k+1}, \dots, a_n \rangle$ is generated from C_q using P_2 . Then we say that the string $s = s_1 \circ s_2 = \langle a_1, \dots, a_n \rangle$ is generated from A_q using the parsing $P = \langle A_q, P_1, P_2 \rangle$.

The Number of Parsings: Usually, the first algorithmic attempts at parsing are some form of brute force algorithm. The problem is that there is an exponential number of parsings to try. This number can be estimated roughly as follows. When parsing, the string of symbols needs to increase from being of size 1 (consisting only of the start symbols) to being of size n (consisting of s). Applying a rule adds only one more symbol to this string. Hence, rules need to be applied $n - 1$ times. Each time you apply a rule, you have to choose which of the m rules to apply. Hence, the total number of choices may be $\Theta(m^n)$.

The Question to Ask the Little Bird: Given an instance $\langle G, T_{start}, s \rangle$, we will ask the little bird a question that contains two subquestions about a parsing P that generates s from T_{start} .

The first subquestion is the index q of the rule $r_q = "T_{start} \Rightarrow B_q C_q"$ that is applied first to our start symbol T_{start} . Although this is useful information, I don't see how it alone could lead to a subinstance.

We don't know P , but we do know that P generates $s = \langle a_1, \dots, a_n \rangle$. It follows that, for some $k \in [1 \dots n]$, after P applies its first rule $r_q = "T_{start} \Rightarrow B_q C_q"$, it then generates the string $s_1 = \langle a_1, \dots, a_k \rangle$ from B_q and the string $s_2 = \langle a_{k+1}, \dots, a_n \rangle$ from C_q , so that overall it generates $s = s_1 \circ s_2 = \langle a_1, \dots, a_n \rangle$. Our second subquestion asked of the bird is to tell us this k that splits the string s .

Help from Friend: What we do not know about the parsing tree P is how B_q generates $s_1 = \langle a_1, \dots, a_k \rangle$ and how C_q generates $s_2 = \langle a_{k+1}, \dots, a_n \rangle$. Hence, we ask our friends for optimal parsings for the subinstances $\langle G, B_q, s_1 \rangle$ and $\langle G, C_q, s_2 \rangle$. They respond with the parsings P_1 and P_2 . We conclude that $P = \langle T_{start}, P_1, P_2 \rangle$ generates $s = s_1 \circ s_2 = \langle a_1, \dots, a_n \rangle$ from T_{start} . If either friend gives us a parsing with zero cost, then we know that no parsing consistent with the information provided by the bird is possible. The cost of our parsings in this case is zero as well. This can be achieved by setting the cost of the new parsing to be the minimum of those for P_1 and for P_2 . The line of code will be $cost_{(q,k)} = \min(cost[B_q, 1, k], cost[C_q k + 1, n])$.

The Set of Subinstances: The set of subinstances that gets called by the recursive program consisting of you, your friends, and their friends is $\{\langle G, T_h, a_i, \dots, a_j \rangle | h \in V, 1 \leq i \leq j \leq n\}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub operation. Consider an arbitrary subinstance $\langle G, T_h, a_i, \dots, a_j \rangle$ in the set. Its subinstances are $\langle G, B_q, a_i, \dots, a_k \rangle$ and $\langle G, C_q, a_{k+1}, \dots, a_j \rangle$, which are both in the set.

Generating: Some of these subinstances will not be generated. However, most of our instances will.

Constructing a Table Indexed by Subinstances: The table will be three-dimensional. The solution for the subinstance $\langle G, T_h, a_i, \dots, a_j \rangle$ will be stored in the entry $Table[h, i, j]$ for $h \in V$ and $1 \leq i \leq j \leq n$. See [Figure 27.3](#).

The Order in Which to Fill the Table: The size of the subinstance $\langle G, T_h, a_i, \dots, a_j \rangle$ is the length of the string to be generated, i.e., $j - i + 1$. We will start with smaller strings and then move to longer and longer strings.

Base Cases: One base case is the subinstance $\langle G, T_{start}, \epsilon \rangle$. This empty string ϵ is parsed with the rule $T_{start} \Rightarrow \epsilon$, assuming that this is a legal rule. The other base cases are the subinstances $\langle G, A_q, b_q \rangle$. This string, consisting of the single character b_q , is parsed with the rule $A_q \Rightarrow b_q$, assuming that this is a legal rule.

Constructing an Optimal Solution

algorithm *ParsingWithAdvice*($\langle G, T_h, a_i, \dots, a_j \rangle, birdAdvice$)

<pre- & post-cond>: Same as *Parsing* except with advice.

begin

$\langle q, k \rangle = birdAdvice[h, i, j]$

 if($i = j$) then

Rule r_q must have the form " $A_q \Rightarrow b_q$ ", where A_q is T_h and b_q is a_i
 Parsing $P = \langle T_h, a_i \rangle$
 else
 Rule r_q must have the form " $A_q \Rightarrow C_q B_q$ ", where A_q is T_h .
 $P_1 = \text{ParsingWithAdvice}(\langle G, B_q, a_i, \dots, a_k \rangle, \text{birdAdvice})$
 $P_2 = \text{ParsingWithAdvice}(\langle G, C_q, a_{k+1}, \dots, a_j \rangle, \text{birdAdvice})$
 Parsing $P = \langle T_h, P_1, P_2 \rangle$
 end if
 return (P)
 end algorithm

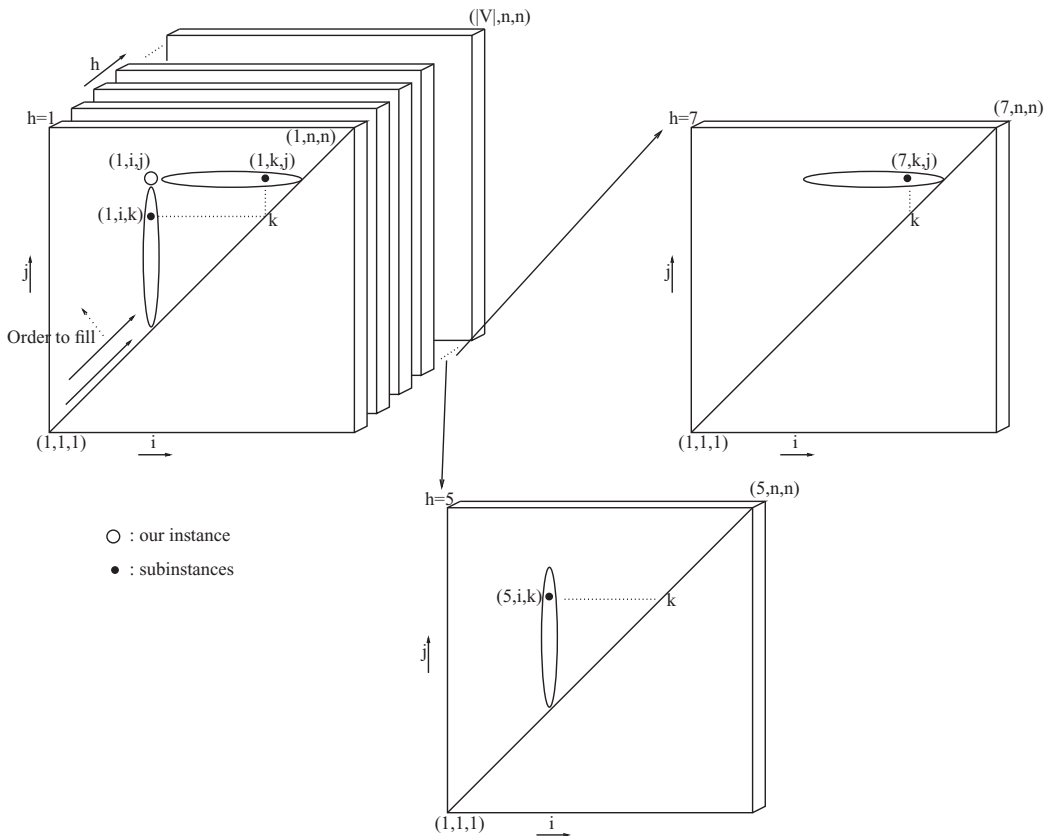


Fig. 27.3 The dynamic programming table for parsing. The table entry corresponding to the instance $\langle G, T_1, a_i, \dots, a_j \rangle$ is represented by the little circle. Using the rule $T_1 \Rightarrow T_1 T_1$, the subinstances $\langle G, T_1, a_i, \dots, a_k \rangle$ and $\langle G, T_1, a_{k+1}, \dots, a_j \rangle$ are formed. Using the rule $T_1 \Rightarrow T_5 T_7$, the subinstances $\langle G, T_5, a_i, \dots, a_k \rangle$ and $\langle G, T_7, a_{k+1}, \dots, a_j \rangle$ are formed. The table entries corresponding to these subinstances are represented by the dots within the ovals.

Exercise 27.4.1 (See solution in Exercise Solutions.) (a) Give the code for the parsing algorithm. (b) Give the running time for this algorithm.

28 Reductions and NP-Completeness

A giraffe with its long neck is a very different beast from a mouse, which is different from a snake. However, Darwin and other biologists observed that the first two have some key similarities, both being social, nursing their young, and having hair. The third is completely different in these ways. Studying similarities and differences between things can reveal subtle and deep understandings of their underlying nature that would not have been noticed by studying them one at a time. Sometimes things that at first appear to be completely different, when viewed in another way, turn out to be the same except for superficial, cosmetic differences. This section will teach how to use reductions to discover these similarities between different optimization problems.

Reduction $P_1 \leq_{poly} P_2$: We say that we can *reduce* problem P_1 to problem P_2 if we can write a polynomial-time ($n^{\Theta(1)}$) algorithm for P_1 using a supposed algorithm for P_2 as a subroutine. (Note we may or may not actually have an algorithm for P_2 .) The standard notation for this is $P_1 \leq_{poly} P_2$.

Why Reduce? A reduction let's us compare the time complexities and underlying structures of the two problems. Reduction is useful in providing algorithms for new problems (upper bounds), for giving evidence that there are no fast algorithms for certain problems (lower bounds), and for classifying problems according to their difficulty.

Upper Bounds: From the reduction $P_1 \leq_{poly} P_2$ alone, we cannot conclude that there is a polynomial-time algorithm for P_1 . But it does tell us that if there is a polynomial-time algorithm for P_2 , then there is one for P_1 . This is useful in two ways. First, it allows us to construct algorithms for new problems from known algorithms for other problems. Moreover, it tells us that P_1 is *at least as easy as* P_2 .

Hot Dogs $\leq_{poly}$ Linear Programming: [Section 21.4](#) describes how to solve the problem of making a cheap hot dog using an algorithm for linear programming.

Bipartite Matching $\leq_{poly}$ Network Flows: We will develop an algorithm for bipartite matching in [Section 28.4](#) that uses the network flow algorithm.

Lower Bounds: The contrapositive of the last statement is that if there is not a polynomial-time algorithm for P_1 , then there cannot be one for P_2 (otherwise there would be one for P_1). This tells us that P_2 is *at least as hard as* P_1 .

(Any Optimization Problem) $\leq_{poly}$ CIR-SAT: This small-looking statement, proved by Steve Cook in 1971, has become one of the foundations of theoretical computer science. There are many interesting optimization problems. Some people have worked hard on discovering fast algorithms for this one and others have done the same for that one. Cook's theorem shows that it is sufficient to focus on the optimization problem CIR-SAT, because if you can solve it quickly, then you can solve them all quickly. However, after many years of working hard, people have given up and strongly suspect that at least one optimization problem is hard. This gives strong evidence that CIR-SAT is hard. Cook's theorem is proved (and the problems defined) in [Section 28.1](#).

CIR-SAT $\leq_{poly}$ 3-COL: See [Section 28.3](#). This states that the optimization problem 3-COL is as hard as CIR-SAT, already known to be a hard problem. This gives evidence that 3-COL is also hard. Moreover, reductions are transitive, meaning that $P_1 \leq_{poly} P_2$ and $P_2 \leq_{poly} P_3$ automatically gives that $P_1 \leq_{poly} P_3$. Hence, together these last two statements give that (any optimization problem) $\leq_{poly}$ 3-COL.

3-COL $\leq_{poly}$ Course Scheduling, 3-COL $\leq_{poly}$ Independent Set, 3-COL $\leq_{poly}$ 3-SAT: These give evidence that course scheduling, independent set, and 3-SAT are hard. See [Sections 28.2](#) and [28.3](#).

(Halting Problem) $\leq_{poly}$ (What Does This Turing Machine Do)? In 1936, Alan Turing designed a *Turing Machine* as the first general model of computation. It has a finite set of states that it transitions between and an infinite memory, single-head tape. Read [Section 26.2](#) Assertions, TM and DFA, for further explanation. It can be proved that the *halting problem* (given a Turing machine M and an input I , does the M halt on I ?) is undecidable (no algorithm can always answer it correctly in finite time). Given this, reductions can be used to prove that almost any problem asking what the computation of a given Turing machine does is also undecidable.

Reverse Reductions: Knowing $P_1 \leq_{poly} P_2$ and knowing that there is not a polynomial-time algorithm for P_2 does not tell us anything about whether there is a polynomial-time algorithm for P_1 . Though it does tell us that the algorithm for P_1 given in the reduction does not work, there well may be another, completely different algorithm for P_1 . Similarly, knowing that there is a polynomial-time algorithm for P_1 does not tell us anything about whether there is one for P_2 . To reach these two conclusions, you must prove the reverse reduction $P_2 \leq_{poly} P_1$.

Classifying Problems: Reductions are used to classify problems.

The Same Problem Except for Superficial Differences: More than just being able to compare their time complexities, knowing $P_1 \leq_{poly} P_2$ and $P_2 \leq_{poly} P_1$ reveals that the two problems are somehow fundamentally the same problem, asking the same types of questions. Sometimes this similarity is quite superficial.

They simply use different vocabulary. However, at other times this connection between the problems is quite surprising, providing a deeper understanding of each of the problems. One way in which we can make a reduction even more striking is by restricting the algorithm for the one to call the algorithm for the other only once. Then the mapping between them is even more direct.

NP-Completeness: We have shown that the optimization problems CIR-SAT, 3-COL, course scheduling, independent set, and 3-SAT are all reducible to each other and in that sense are all fundamentally the same problem. In fact, there are thousands of very different problems that are equivalent to these. These problems are said to be *NP-complete*. We discuss this further in [Section 28.2](#).

Halting-Problem Completeness: Another important class defined in this way consists of all problems that are equivalent to the halting problem.

28.1 Satisfiability Is at Least as Hard as Any Optimization Problem

In [Chapter 19](#) we saw that *optimization problems* involve searching through the exponential set of solutions for an instance to find one with optimal cost. Though there are quick (i.e., polynomial) algorithms for some of these problems, for most of them the best known algorithms require $2^{\Theta(n)}$ time on the worst case input instances, and it is strongly believed that there are no polynomial-time algorithms for them. The main reason for this belief is that many smart people have devoted many years of research to looking for fast algorithms and have not found them. This section uses reductions to prove that some of these optimization problems are universally hard, or *complete*, among the class of optimization problems, because if you could design an algorithm to solve such a problem quickly, then you could translate this algorithm into one that solves any optimization problem quickly. Conversely (and more likely), if there is even one optimization problem that cannot be solved quickly, then none of these complete problems can be either. Proving in this way that a problem that your boss wants you to solve is hard is useful, because you will know not to spend much time trying to design an all-purpose algorithm for it.

(Any Optimization Problem) $\leq_{poly}$ CIR-SAT: This reduction will prove the satisfiability problem is complete for the class of optimization problems, meaning that it is universally hard for this class.

The Circuit Satisfiability Problem: This famous computational problem, CIR-SAT, requires one to find a satisfying assignment for a given circuit. [Section 23.4](#) gives a recursive backtracking algorithm for the satisfiability problem, but in the worst case its running time is $2^{\Theta(n)}$.

Circuit: A *circuit* can be either a useful notation for describing an algorithm in detail or a practical thing built in silicon in your computer.

Construction: It is built with *AND*, *OR*, and *NOT* gates. At the top are n wires labeled with the binary variables $x_1, x_2, \dots, x_n$. To specify the circuit's input, each of these will take on either 1 or 0, *true* or *false*, 5 volts

or 0 volts. Each *AND* gate has two wires coming into it, either from an input x_i or from the output of another gate. An *AND* gate outputs *true* if both of its inputs are *true*. Similarly, each *OR* gate outputs *true* if at least one of its inputs is *true*, and each *NOT* gate outputs *true* if its single input is *false*. We will only consider circuits that have no cycles, so these *true* and *false* values percolate down to the output wires. There will be a single output wire if the circuit computes a *true*–*false* function of its input $x_1, x_2, \dots, x_n$ and will have m output wires if it outputs an m -bit string, which can be used to encode some required information.

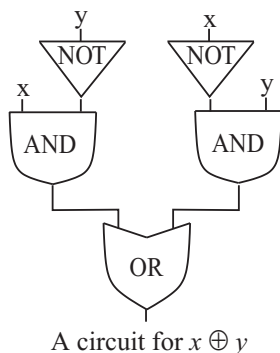


Fig. 28.1 A circuit for $x \oplus y$.

Binary Endoding: The function f that you want to compute may take as input some abstract object like a graph G and return another abstract object, like a path through this graph; however, whether the computation of f is done by a Java program, a Turing machine, or a circuit, these abstract objects first need to be encoded into strings of zeros and ones.

Compute Any Function: Given any function $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$, a circuit can compute it with at most $O(nm \cdot 2^n)$ gates as follows. For any fixed input instance $\langle x_1, x_2, \dots, x_n \rangle = \langle 1, 0, \dots, 1 \rangle$, a circuit can say “the input is this fixed instance” simply by computing $[(x_1 = 1) \text{ AND } \text{NOT}(x_2 = 1) \text{ AND } \dots \text{AND } (x_n = 1)]$. Then the circuit computes the i th bit of the function’s output by outputting 1 if [“the input is this fixed instance” OR “this instance” OR $\dots$ OR “this instance”], where each instance is listed for which the i th bit of the function’s output is 1.

Polynomial Size for Polynomial Time: More importantly, given any algorithm whose Turing machine’s running time is $T(n)$ and given any fixed integer n , there is an easily constructed circuit with at most $\Theta(T(n)^2)$ gates that computes the output of the algorithm given any n -bit input instance. Change the definition of a Turing machine slightly so that each cell is big enough so that the cell currently being pointed to by the head can store not only its contents but also the current state of the machine. This cell’s contents can be encoded with $\Theta(1)$ bits. Because the Turing machine uses only $T(n)$ time, it can use at most the first $T(n)$ cells of memory.

For each of the $T(n)$ steps of the algorithm, the circuit will have a row of $\Theta(1) \cdot T(n)$ wires whose values encode the contents of memory of these $T(n)$ cells during this time step. The gates of the circuit between these rows of wires compute the next contents of memory from the current contents. Because the contents of cell i at time t depend only on the contents of cells $i - 1, i$, and $i + 1$ at time $t - 1$ and each of these is only a $\Theta(1)$ number of bits, this dependence can be computed using a circuit with $\Theta(1)$ gates. This is repeated in a matrix of $T(n)$ time steps and $T(n)$ cells, for a total of $\Theta(T(n)^2)$ gates. At the bottom, the circuit computes the output of the function from the contents of memory of the Turing machine at time $T(n)$.

Circuit Satisfiability Specification: The CIR-SAT problem takes as input a circuit with a single *true-false* output and returns an assignment to the variables $x_1, x_2, \dots, x_n$ for which the circuit gives *true*, if such an assignment exists.

Optimization Problems: This reduction will select a generic optimization problem and show that CIR-SAT is at least as hard as it is. To do this, we need to have a clear definition of what a generic optimization problem looks like.

Definition: Each such problem has a set of instances that might be given as input; each instance has a set of potential solutions, some of which are valid; and each solution has a cost. The goal, given an instance, is to find one of its valid solutions with optimal cost. An important feature of an optimization problem is that there are polynomial-time algorithms for the following:

Valid(I, S): Given an instance I and a potential solution S , there is an algorithm $Valid(I, S)$ running in time $|I|^{O(1)}$ that determines whether I is a valid instance for the optimization problem and that S is a valid solution for I .

Cost(S): Given a valid solution S , there is an algorithm $Cost(S)$ running in time $|I|^{O(1)}$ that computes the cost of the solution S .

Example: Course Scheduling: Given the set of courses requested by each student and the set of time slots available, find a schedule that minimizes the number of conflicts.

I : The set of courses requested by each student and the set of time slots available

S : A schedule specifies at which time start each course will be taught.

Valid(I, S): An algorithm that returns whether the schedule S allocates each course requested in I to exactly one time slot provided in I .

Cost(S): A conflict occurs in the schedule when two courses requested by the same student are scheduled at the same time. Cost (S) is an algorithm that returns the number of conflicts in the schedule S .

Alg for the Optimization Problem: Given a fast algorithm $Alg_{CIR-SAT}$ for CIR-SAT and the descriptions $Valid(I, S)$ and $Cost(S)$ of an optimization problem P , we will now design a fast algorithm Alg_P for the optimization problem and use it to prove that the problem CIR-SAT is at least as hard as the optimization problem, i.e., that $P \leq_{poly}$ CIR-SAT.

Binary Search for Cost: Given some instance I_P of the optimization problem, Alg_P 's first task is to determine the cost c_{opt} of the optimal solution for I . Alg_P starts by determining whether or not there is a valid solution for I that has cost at least $c = 1$. If it does, Alg_P repeats this with $c = 2, 4, 8, 16, \dots$. If it does not, Alg_P tries $c = 0, -1, -2, -4, \dots$, until it finds c_1 and c_2 between which it knows the cost of an optimal solution lies. Then it does a binary search to find c_{opt} . The last step is to find a solution for I that has this optimal cost.

Finding a Solution with Given Cost: Alg_P determines whether I has a solution S with cost at least c or finds a solution with cost c_{opt} , as follows. Alg_P will construct a circuit C and calls the algorithm $Alg_{CIR-SAT}$, which provides a satisfying assignment to C . Alg_P wants the satisfying assignment that $Alg_{CIR-SAT}$ provides to be the solution S that it needs. Hence, Alg_P designs C to be satisfied by the assignment S only if S is a solution S for I with a cost as required, i.e., $C(S) \equiv [Valid(I, S) \text{ and } Cost(S) \geq c]$. Because there are polynomial-time algorithms for $Valid(I, S)$, for $Cost(S)$, and for $\geq$, the algorithm Alg_P can easily construct such a circuit $C(S)$. If such a solution S satisfying C exists, then $Alg_{CIR-SAT}$ kindly provides one.

This completes the reduction $P \leq_{poly}$ CIR-SAT of any Optimization problem to CIR-SAT.

Exercise 28.1.1 For each of the following problems, define I , S , $Valid(I, S)$, and $Cost(S)$.

1. **Graph coloring:** Given a graph, color its nodes so that two nodes do not have the same color if they have an edge between them. Use as few colors as possible.
2. **Independent set:** Given a graph, find a largest subset of the nodes for which there are no edges between any pair in the set.
3. **Airplane:** Given the requirements of an airplane, design it, optimizing its performance.
4. **Business:** Given a description of a business, make a business plan to maximize its profits.
5. **Factoring:** Given an integer, factor it, e.g., $6 = 2 \times 3$.
6. **Cryptography:** Given an encrypted message, decode it.

28.2 Steps to Prove NP-Completeness

In this section, we define the class NP and give steps for proving that a computational problem is NP-complete.

Completeness for Nondeterministic Polynomial-Time Decision Problems: The set of computational problems that are complete (universally hard) for optimization problems is extremely rich and varied. Studying them has become a fascinating field of research.

NP Decision Problems: Theoretical computer scientists generally only consider a subclass of the optimization problems, referred to as the class of *nondeterministic polynomial-time* problems (NP).

One Level of Cost: Instead of worrying about whether one solution has a better cost than another, we will completely drop the notion of the cost of a solution. S will only be considered a *valid solution* for the instance I if it is a solution with a sufficiently good cost. This is not a big restriction, because if you want to consider solutions with different costs, you can always do a binary search, as has already been done for the cost of the optimal solution.

Decision Problem: Given an instance to the problem, the goal is to determine either *yes* I does have a valid solution or *no* it does not.

Witness: A solution for an instance is often referred to as a *witness*, because, though it may take exponential time to find it, if it were provided by a (nondeterministic) fairy godmother, then it could be used in polynomial time to witness the fact that the answer for this instance is yes. In this respect, NP problems are asymmetrical in that there does not seem to be a witness that quickly proves that an instance does not have a solution.

Formal Definition: We say that such a computational problem P is in the class of nondeterministic polynomial-time problems (NP) if there is a polynomial-time algorithm $Valid(I, S)$ that specifies *Yes* when S is a (sufficiently good) solution for the instance I , and *No* if not. More formally, P can be defined as follows:

$$P(I) \equiv [\exists S, Valid(I, S)].$$

Examples:

Circuit Satisfiability (CIR-SAT): Circuit satisfiability could be defined as a decision problem: Given a circuit, determine whether there is an assignment that satisfies it.

Graph 3-Coloring (3-COL): Given a graph, determine whether its nodes can be colored with three colors so that two nodes do not have the same color if they have an edge between them.

Course Scheduling: Given the set of courses requested by each student, the set of time slots available, and an integer K , determine whether there is schedule with at most K conflicts.

Cook vs. Karp Reductions: Stephen Cook first proved that CIR-SAT is complete for the class of NP problems. His definition of a reduction $P_1 \leq_{poly} P_2$ is

that one can write an algorithm Alg_1 for the problem P_1 using an algorithm Alg_2 for the problem P_2 as a subroutine. In general, this algorithm Alg_1 may call Alg_2 as many times as it likes and do anything it likes with the answers that it receives. Richard Karp later observed that when the problems P_1 and P_2 are sufficiently similar, the algorithm Alg_1 used in the reduction need only call Alg_2 once and answers *Yes* if and only if Alg_2 answers *Yes*. These two definitions of reductions are referred to as *Cook* and *Karp* reductions. Though we have defined Cook reductions because they are more natural, we will consider only Karp reductions from here on.

NP-Completeness: For the problem P to be NP-complete, it has to be hard, but not too hard. We say that a computational problem P is *NP-complete* if

Sufficiently Hard, $P' \leq_{poly} P$: We say a problem P is *NP-Hard* if it is as hard as every other problem P' in the class *NP*. Intuitively, this means that if one did find a quick algorithm for P , then this algorithm could be translated into quick algorithms for each NP problem P' . More formally, it means that every language in NP can be polynomially reduced to it using a Karp reduction.

$$\forall \text{ optimization problems } P', P' \leq_{poly} P.$$

To prove this, it is sufficient to prove that our computational problem is at least as hard as some problem already known to be NP-complete. For example, because we now know that CIR-SAT is NP-complete, it is sufficient to prove that $\text{CIR-SAT} \leq_{poly} P$.

Not Too Hard, $P \in \text{NP}$: On the other hand, for the problem P to be *complete* for the class NP, it has to be sufficiently easy that it is itself in the class. For example, the Halting problem is sufficiently hard to be NP-hard, but is far too hard to be in NP.

The Steps to Prove NP-Completeness: Proving that a problem P is NP-complete can be a bit of an art, but once you get the hang of it, it can be fun. I will now carefully lay out the steps needed. (After step 3, we will use P_{oracle} to denote the problem instead of P , because at that point we will be assuming that we have an oracle for it.)

Running Example: Course Scheduling Is NP-Complete: The problem P that we will prove is NP-complete will be the course scheduling problem.

(0) $P \in \text{NP}$: As said, in order for the problem P to be NP-complete, it needs to be sufficiently easy to be in NP. To prove this we effectively need to provide a nondeterministic polynomial time for it. This is accomplished by providing polynomial-time algorithm $\text{Valid}(I, H)$ that specifies whether S is a valid solution for instance I .

Course Scheduling: It is not hard to determine in polynomial time whether the instance I and the solution S are properly defined and to check that within this

schedule S , the number of times that a student wants to take two courses that are offered at the same time is at most K .

1) What to Reduce to It: An important and challenging step in proving that a problem is NP-complete is deciding which NP-complete problem to reduce to it. We will denote this problem with P_{alg} because later we will be designing an algorithm for it.

3-COL $\leq_{poly}$ Course Scheduling: We will reduce 3-COL to course scheduling, that is, we will prove the reduction $3\text{-COL} \leq_{poly} \text{course scheduling}$. I will save the proof that 3-COL is NP-complete for our next example, because it is much harder.

Hint: You want to choose a problem that is “similar” in nature to yours. In order to have more to choose from, it helps to know a large collection of problems that are NP-complete. There are entire books devoted to this. When in doubt, 3-SAT and 3-COL are good problems to use.

2) What Is What: It is important to remember what everything is.

3-COL $\leq_{poly}$ Course Scheduling

- $P_{alg} = 3\text{COL}$ is the graph 3-coloring problem.
- $I_{alg} = I_{graph}$, an instance of it, is an undirected graph.
- $S_{alg} = S_{coloring}$, a potential solution, is a coloring of each of its nodes with either red, blue, or green. It is a valid solution if no edge has two nodes with the same color.
- Alg_{alg} is an algorithm that takes the graph I_{graph} as input and determines whether it has a valid coloring.
- $P_{oracle} = \text{course scheduling}$.
- $I_{oracle} = I_{courses}$, an instance of it, is the set of courses requested by each student, the set of time slots available, and the integer K .
- $S_{oracle} = S_{schedule}$, a potential solution, is a schedule assigning courses to time slots. It is a valid solution if it has at most K conflicts.
- $Algoracle$ is an algorithm that takes $I_{courses}$ as input and determines whether it has a valid schedule.

Such instances may or may not be satisfiable and such potential solutions may or may not be valid.

Warning: Be especially careful about what is an instance and what is a solution for each of the two problems.

3) Direction of Reduction and Code: Another common source of mistakes is doing the reduction in the wrong direction. I recommend not memorizing this direction, but working it out each time. Our goal is to prove that the problem P is sufficiently hard to be NP-complete. Hence, you must put P on the hard side of the inequality $3\text{-COL} \leq_{poly} P$ with the problem, say 3-COL, chosen in step 1 on the easy side. At this point,

turn your thinking around. Instead of proving P is relatively hard, we will prove that the problem 3-COL is relatively easy. To do this, we must design a fast algorithm Alg_{alg} for it. Because this is our goal from here on, we will denote the problem 3-COL with P_{alg} . Our belief is that there is not a fast algorithm for this problem. Hence, to help us we will use a supposed fast algorithm, Alg_{oracle} , for P as a subroutine. Typically for reductions, people assume that Alg_{oracle} is an *oracle*, meaning that it solves its problem in one time step. Hence, from here on we will use P_{oracle} to denote the problem instead of P . The code for our algorithm for P_{alg} will be as follows.

algorithm $Alg_{alg}(I_{alg})$

⟨**pre-cond**⟩: I_{alg} is an instance of P_{alg} .

⟨**post-cond**⟩: Determine whether I_{alg} has a solution S_{alg} , and if so, return it.

begin

$I_{oracle} = InstanceMap(I_{alg})$

% I_{oracle} is an instance of P_{oracle}

$\langle ans_{oracle}, S_{oracle} \rangle = Alg_{oracle}(I_{oracle})$

% If there is one, S_{oracle} is solution for I_{oracle}

if ($ans_{oracle} = Yes$) then

$ans_{alg} = Yes$

$S_{alg} = SolutionMap(S_{oracle})$

else

$ans_{alg} = No$

$S_{alg} = nil$

end if

return ($\langle ans_{alg}, S_{alg} \rangle$)

end algorithm

4) Look for Similarities: Though the problems P_{alg} and P_{oracle} may appear to be very different, the goal in this step is to look for underlying similarities. Compare how their solutions, S_{alg} and S_{oracle} , are formed out of their instances, I_{alg} and I_{oracle} . Generally, the instances can be thought of as sets of elements with a set of constraints between them. Can you view their solutions as subsets of these elements or as labelings of them? What allows a solution to form, and what constrains how it is formed? Can you talk about the two problems using the same language? For example, a subset of elements can be viewed as a labeling of the elements with zero and one. Similarly, a labeling of each element e with $\ell_e \in [1..L]$ can be viewed as a subset of the pairs $\langle e, \ell_e \rangle$.

3-COL $\leq_{poly}$ Course Scheduling: A solution $S_{coloring}$ is a coloring that assigns a color to each node. A solution $S_{schedule}$ is a schedule that assigns a time slot to each course. This similarity makes it clear that there is a similarity between the roles of the nodes of I_{graph} and of the courses of $I_{courses}$ and between the colors of $S_{coloring}$ and the time slots of $S_{schedule}$. Each coloring conflict arises from an edge between nodes, and each scheduling conflict arises from a student wanting two courses. This similarity makes it clear that there is a similarity between the roles of the edges of I_{graph} and of the course requests of $I_{courses}$.

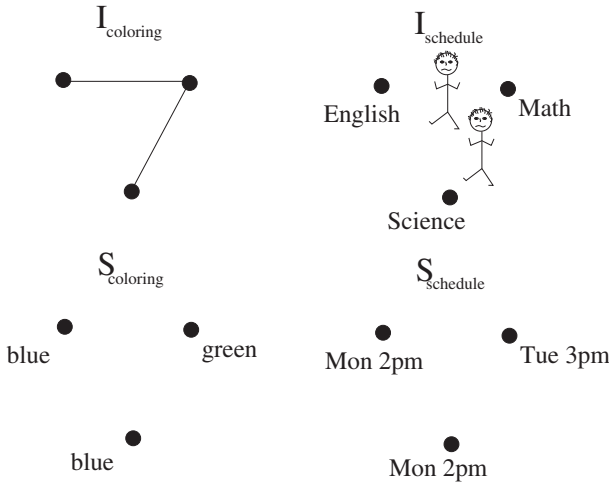


Fig. 28.2 The input and the solution when doing the reducing between node coloring and scheduling.

5) InstanceMap: You must define a polynomial-time algorithm $InstanceMap(I_{alg})$ that, given an instance I_{alg} of P_{alg} , constructs an instance I_{oracle} of P_{oracle} that has similar sorts of solutions. The main issue is that the constructed instance I_{oracle} has a solution if and only if the given instance I_{alg} has a solution, that is, *Yes* instances get mapped to *Yes* instances and *No* to *No*.

3-COL $\leq_{poly}$ Course Scheduling: Given a graph I_{graph} to be colored, we design an instance $I_{courses} = InstanceMap(I_{graph})$ to be scheduled. Using the similarities observed in step 4, in our mapping we will have one course for each node of the graph, and one time slot for each of the three colors green, red, and blue. For each edge between nodes u and v in the graph, we will have a student who requests both course u and course v . The coloring problem does not allow any conflicts. Hence, we set $K = 0$.

Not Onto or 1-1: It is important that each instance I_{alg} be mapped to some instance I_{oracle} , but it is not important whether an instance P_{oracle} is mapped to more than one or none at all. In our example, we never mention instances to be scheduled that have more than three time slots or that allow $K > 0$ conflicts.

Warning: Be sure to do this mapping in the correct direction. The first step in designing an algorithm Alg_{alg} is to suppose that you have been given an input I_{alg} for it. Before your algorithm can call the algorithm Alg_{oracle} as a subroutine, you must construct an instance I_{oracle} to give to it.

Warning: Do not define the mapping only for the *Yes* instances or use a solution S_{alg} for I_{alg} for determining the instance I_{oracle} mapped to. The algorithm Alg_{alg} that you are designing is given an instance I_{alg} , but it does not know whether or not the instance has a solution. The whole point is to give an argument

that finding a solution may take exponential time. It is safer, when defining the mapping $InstanceMap(I_{alg})$, not to even mention whether the instance I_{alg} has a solution or what that solution might be.

6) SolutionMap: You must also define a polynomial-time algorithm $SolutionMap$ (S_{oracle}) mapping each valid solution S_{oracle} for the instance $I_{oracle} = InstanceMap(I_{alg})$ you just constructed to a valid solution S_{alg} for the instance I_{alg} that was given as input. Valid solutions can be subtle, and the instance I_{oracle} may have some solutions that you had not intended when you constructed it. One way to help avoid missing some is to throw a much wider net by considering all *potential* solutions. In this step, for each potential solution S_{oracle} for I_{oracle} , you must either give a reason why it is not a valid solution or map it to a solution $S_{alg} = SolutionMap(S_{oracle})$ for I_{alg} . It is fine if some of the solutions that you map happen not to be valid.

3-COL $\leq_{poly}$ Course Scheduling: Given a schedule $S_{schedule}$ assigning course u to time slots c , we define $S_{coloring} = SolutionMap(S_{schedule})$ to be the coloring that colors node u with color c .

Warning: When the instance I_{oracle} you constructed has solutions that you did not expect, there are two problems. First, the unknown algorithm Alg_{oracle} may give you one of these unexpected solutions. Second, there is a danger that I_{oracle} has solutions but your given instance I_{alg} does not. For example, if, in step 5, our I_{oracle} allowed more than three time slots or more than $K=0$ conflicts, then the instance might have many unexpected solutions. In such cases, you may have to redo step 5, adding extra constraints to the instance I_{oracle} so that it no longer has these solutions.

7) Valid to Valid: In order to prove that the algorithm $Alg_{alg}(I_{alg})$ works, you must prove that if S_{oracle} is a valid solution for $I_{oracle} = InstanceMap(I_{alg})$, then $S_{alg} = SolutionMap(S_{oracle})$ is a valid solution for I_{alg} .

3-COL $\leq_{poly}$ Course Scheduling: Supposing that the schedule is valid, we prove that the coloring is valid as follows. The instance to be scheduled is constructed so that, for each edge of the given graph, there is a student who requests the courses u and v associated with the nodes of this edge. Because the schedule is valid, there are $K=0$ course conflicts, and hence these courses are all scheduled at different time slots. The constructed coloring therefore allocates different colors to these nodes.

8) ReverseSolutionMap: Though we do not need it for the code, for the proof you must define an algorithm $ReverseSolutionMap(S'_{alg})$ mapping in the reverse direction from each potential solution S'_{alg} for the instance I_{alg} to a potential solution S'_{oracle} for the instance I_{oracle} .

3-COL $\leq_{poly}$ Course Scheduling: Given a coloring $S'_{coloring}$ coloring node u with color c , we define $S'_{schedule} = ReverseSolutionMap(S'_{coloring})$ to be the schedule assigning course u to time slots c .

Warning: $\text{ReverseSolutionMap}(S'_{alg})$ does not need to be the inverse map of $\text{SolutionMap}(S_{oracle})$. You must define the mapping $\text{ReverseSolutionMap}(S_{alg})$ for every possible solution S'_{alg} , not just those mapped to by $\text{SolutionMap}(S_{oracle})$. Otherwise, there is the danger that I_{alg} has solutions but your constructed instance I_{oracle} does not.

9) Reverse Valid to Valid: You must also prove the reverse direction: that if S'_{alg} is a valid solution for I_{alg} , then $S'_{oracle} = \text{ReverseSolutionMap}(S'_{alg})$ is a valid solution for $I_{oracle} = \text{InstanceMap}(I_{alg})$.

3-COL $\leq_{poly}$ Course Scheduling: Supposing that the coloring is valid, we prove that the schedule is valid as follows. The instance to be scheduled is constructed so that each student requests the courses u and v associated with nodes of some edge. Because the coloring is valid, these nodes have been allocated different colors and hence the courses are all scheduled in different time slots. Hence, there will be $K = 0$ course conflicts.

10) Working Algorithm: Given the above steps, it is now possible to prove that if the supposed algorithm Alg_{oracle} correctly solves P_{oracle} , then our algorithm Alg_{alg} correctly solves P_{alg} .

Yes to Yes: We start by proving that Alg_{alg} answers *Yes* when given an instance for which the answer is *Yes*. If I_{alg} is a *Yes* instance, then by the definition of the problem P_{alg} , it must have a valid solution. Let us denote by S'_{alg} one such valid solution. Then by step 9, it follows that $S'_{oracle} = \text{ReverseSolutionMap}(S'_{alg})$ is a valid solution for $I_{oracle} = \text{InstanceMap}(I_{alg})$. This witnesses the fact that I_{oracle} has a valid solution and hence I_{oracle} is an instance for which the answer is *Yes*. If Alg_{oracle} works correctly as supposed, then it returns *Yes* and a valid solution S_{oracle} . Our code for Alg_{alg} will then return the correct answer *Yes* and $S_{alg} = \text{SolutionMap}(S_{oracle})$, which by step 7 is a valid solution for I_{alg} .

No to No: We must now prove the reverse, that if the instance I_{alg} given to Alg_{alg} is a *No* instance, then Alg_{alg} answers *No*. The problem with *No* instances is that they have no witness to prove that they are *No* instances. Luckily, to prove something, it is sufficient to prove the contrapositive. Instead of proving $A \Rightarrow B$, where $A = "I_{alg} \text{ is a No instance}"$ and $B = "\text{Alg}_{alg} \text{ answers No}"$, we will prove that $\neg B \Rightarrow \neg A$, where $\neg B = "\text{Alg}_{alg} \text{ answers Yes}"$ and $\neg A = "I_{alg} \text{ is a Yes instance}"$. Convince yourself that this is equivalent.

If Alg_{alg} is given the instance I_{alg} and answers *Yes*, our code is such that Alg_{oracle} must have returned *Yes*. If Alg_{oracle} works correctly as supposed, the instance $I_{oracle} = \text{InstanceMap}(I_{alg})$ that it was given must be a *Yes* instance. Hence, I_{oracle} must have a valid solution. Let us denote by S_{oracle} one such valid solution. Then by step 7, $S_{alg} = \text{SolutionMap}(S_{oracle})$ is a valid solution for I_{alg} , witnessing I_{alg} being a *Yes* instance. This is the required conclusion $\neg A$.

This completes the proof that if the supposed algorithm Alg_{oracle} correctly solves P_{oracle} , then our algorithm Alg_{alg} correctly solves P_{alg} .

11) Running Time: The remaining step is to prove that the constructed algorithm Alg_{alg} runs in polynomial time ($|I_{alg}|^{\Theta(1)}$). Steps 5 and 6 require that both $InstanceMap(I_{alg})$ and $SolutionMap(S_{oracle})$ work in polynomial time. Hence, if P_{oracle} can be solved quickly, then Alg_{alg} runs in polynomial time. Typically, for reductions people assume that Alg_{oracle} is an *oracle*, meaning that it solves its problem in one time step. [Exercise 28.2.5](#) explores the issue of running time further.

This concludes the proof that P_{oracle} = course scheduling is NP-complete (assuming, of course, that $P_{alg} = 3\text{-COL}$ has already been proven to be NP-complete).

Exercise 28.2.1 We began this section by proving (any optimization problem) $\leq_{poly}$ CIR-SAT. To make this proof more concrete, redo it, completing each of the above steps specifically for $3\text{-COL} \leq_{poly}$ CIR-SAT. (Hint: The circuit $I_{oracle} = InstanceMap(I_{alg})$ should have a variable $x_{\langle u, c \rangle}$ for each pair $\langle u, c \rangle$.)

Exercise 28.2.2 3-SAT is a subset of the CIR-SAT problem in which the input circuit must be a big AND of clauses, each clause must be the OR of at most three literals, and each literal is either a variable or its negation. Prove that 3-SAT is NP-complete by proving that $3\text{-COL} \leq_{poly}$ 3-SAT. (Hint: The answer is almost identical to that for [Exercise 28.2.1](#).)

Exercise 28.2.3 Let $\overline{CIR\text{-}SAT}$ be the complement of the CIR-SAT problem, namely, the answer is Yes if and only if the input circuit is not satisfiable. Can you prove $CIR\text{-}SAT \leq_{poly} \overline{CIR\text{-}SAT}$ using Cook reductions? Can you prove it using Karp reductions?

Exercise 28.2.4 (See solution in Exercise Solutions.) Suppose problem P_1 is a restricted version of P_2 , in that they are the same except P_1 is defined on a subset $I_1 \subseteq I_2$ of the instances that P_2 is defined on. For example, 3-SAT is a restricted version of CIR-SAT, because both determine whether a given circuit has a satisfying assignment; however, 3-SAT only considers special types of circuits with clauses of three literals. How hard is it to prove $P_1 \leq_{poly} P_2$? How hard is it to prove $P_2 \leq_{poly} P_1$?

Exercise 28.2.5 (See solution in Exercise Solutions.) Suppose that when proving $P_{alg} \leq_{poly} P_{oracle}$, the routines $InstanceMap(I_{alg})$ and $SolutionMap(S_{oracle})$ each run in $O(|I_{alg}|^3)$ time, and that the mapping $InstanceMap(I_{alg})$ constructs from the instance I_{alg} an instance I_{oracle} that is much bigger, namely, $|I_{oracle}| = |I_{alg}|^2$. Given the following two running times of the algorithm Alg_{oracle} , determine the running time of the algorithm Alg_{alg} . (Careful!)

1. $Time(Alg_{oracle}) = \Theta(2^{n^{\frac{1}{3}}})$.
2. $Time(Alg_{oracle}) = \Theta(n^c)$ for some constant c .

28.3 Example: 3-Coloring Is NP-Complete

We will now use the steps again to prove that 3-coloring is NP-complete.

0) In NP: The problem 3-COL is in NP because, given an instance graph I_{graph} and a solution coloring $S_{coloring}$, it is easy to have an algorithm $Valid(I_{graph}, S_{coloring})$ check

that each node is colored with one of three colors and that the nodes of each edge have different colors.

1) What to Reduce to It: We will reduce CIR-SAT to 3-COL by proving CIR-SAT $\leq_{poly}$ 3-COL. In [Section 28.1](#) we proved that (any optimization problem) $\leq_{poly}$ CIR-SAT and that 3-COL $\leq_{poly}$ course scheduling. By transitivity, this gives us that CIR-SAT, 3-COL, and course scheduling are each NP-complete problems.

2) What Is What

- P_{alg} is the circuit satisfiability problem (CIR-SAT).
- $I_{circuit}$, an instance of it, is a circuit.
- $S_{assignment}$, a potential solution, is an assignment to the circuit variables $x_1, x_2, \dots, x_n$.
- P_{oracle} is the graph 3-coloring problem (3-COL).
- I_{graph} , an instance to it, is a graph.
- $S_{coloring}$, a potential solution, is a coloring of the nodes of the graph with three colors.

3) Direction of Reduction and Code: To prove 3-COL is at least as hard, we must prove that CIR-SAT is at least as easy, i.e., CIR-SAT $\leq_{poly}$ 3-COL. To do this, we must design an algorithm for CIR-SAT given an algorithm for 3-COL. The code will be identical to that in [Section 28.2](#).

4) Look for Similarities: An assignment allocates *true* or *false* values to each variable, which in turn induces *true* or *false* values to the output of each gate. A coloring allocates one of three colors to each node. This similarity hints at mapping the variables and outputs of each gate to nodes in the graph and mapping *true* to one color and *false* to another. With these ideas in mind, Steven Rudich made a computer search for the smallest graph that behaves like an *OR* gate when colored with three colors. The graph found is shown in [Figure 28.3](#). He calls it an *OR gadget*.

Translating between Colors and True/False: The three nodes v_T , v_F , and v_R in the *OR* gadget are referred to as the *pallet*. Because of the edges between them, when the gadget is properly colored, these nodes need to be assigned different colors. We will call whatever color is assigned to the node v_T the color indicating *true*; that assigned to v_F , the color indicating *false*; and that assigned to v_R , the *remaining* color. For example, in all the colorings in [Figure 28.3](#), green (g) indicates *true*, red (r) indicates *false*, and blue (b) is the remaining color.

Input and Output Values: The nodes v_x and v_y in the *OR* gadget act as the gadget's inputs, and the node v_{out} as its output. Because each of these nodes has an edge to node v_R , they cannot be colored with the remaining color. The node will be said to have the value *true* if it is assigned the same color as v_T , and *false* if the same as v_F . The coloring in the second figure in [Figure 28.3](#) sets $x = \text{false}$, $y = \text{true}$, and the output $= \text{true}$. The coloring in the third figure sets $x = \text{false}$, $y = \text{false}$, and the output $= \text{false}$.

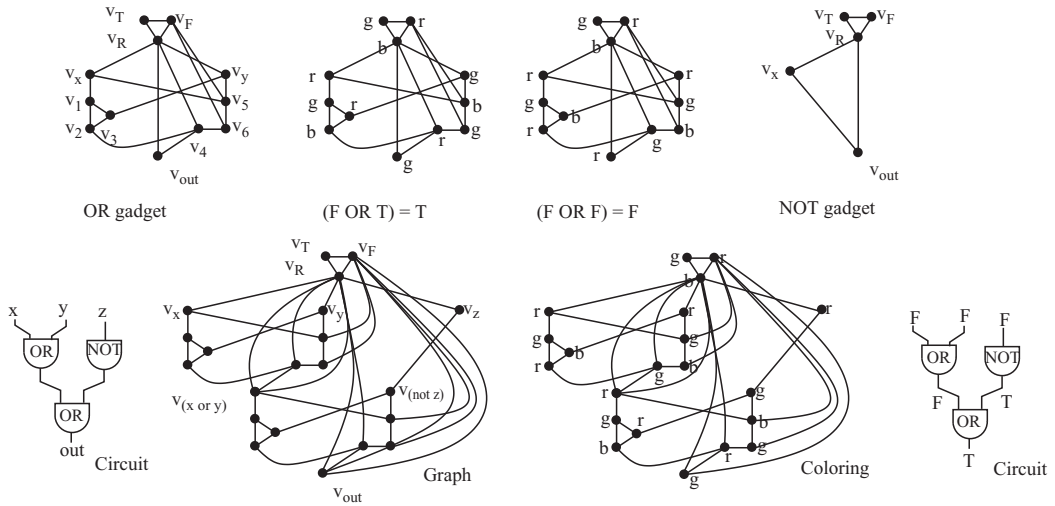


Fig. 28.3 On the top, the first diagram is the *OR* gadget. The next two are colorings of this gadget demonstrating *(false or true) = true* and *(false or false) = false*. The top right diagram is the *NOT* gadget. On the bottom, the first diagram is the circuit given as an instance to SAT. The next is the graph that it is translated into. The next is a 3-coloring of this graph. The last is the assignment for the circuit obtained from the coloring.

Theorem 28.3.1 Rudich's *OR* gadget acts like an *OR* gate, in that it always can be and always must be colored so that the value of its output node v_{out} is the *OR* of the values of its two input nodes v_x and v_y . Similarly for the *NOT* gate.

Proof: There are four input instances to the gate to consider.

(false OR true) = true: If node v_x is colored false and v_y is colored true, then because v_5 has an edge to each, it must be colored the remaining color: v_6 , with edges to v_F and v_5 , must be colored true. v_4 , with edges to v_R and v_6 , must be colored false. v_{out} , with edges to v_R and v_4 , must be colored true. The coloring in the second diagram in Figure 28.3 proves that such a coloring is possible.

(false OR false) = false: If nodes v_x and v_y are both colored false, then neither nodes v_1 nor v_3 can be colored false. Because of the edge between them, one of them must be true and the other the remaining color. Because v_2 has an edge to each of them, it must be colored false. v_4 , with edges to v_R and v_2 , must be colored true. v_{out} , with edges to v_R and v_4 , must be colored false. The coloring in the third diagram in Figure 28.3 proves that such a coloring is possible.

(true OR true) = true, and (true OR false) = true: See Exercise 28.3.1 for these cases and for the *NOT* gate.

5) InstanceMap, Translating the Circuit into a Graph: Our algorithm for CIR-SAT takes as input a circuit $I_{circuit}$ to be satisfied and, in order to receive help from the 3-COL

algorithm, constructs from it a graph $I_{\text{graph}} = \text{InstanceMap}(I_{\text{circuit}})$ to be colored. See the first two diagrams on the bottom of Figure 28.3. The graph will have one pallet of nodes v_T , v_F , and v_R with which to define the *true* and the *false* color. For each variable x_i of the circuit, it will have one node labeled x_i . It will also have one node labeled x_{out} . For each *OR* gate and *NOT* gate in the circuit, the graph will have one copy of the *OR* gadget or the *NOT* gadget. The *AND* gates could be translated into a similar *AND* gadget or translated to $[x \text{ AND } y] = [\text{NOT}(\text{NOT}(x) \text{ OR } \text{NOT}(y))]$. All of these gadgets share the same three pallet nodes. If in the circuit the output of one gate is the input of another, then the corresponding nodes in the graph are the same. Finally, one extra edge is added to the graph from the v_F node to the v_{out} node.

6) SolutionMap, Translating a Coloring into an Assignment: When the supposed algorithm finds a coloring S_{coloring} for the graph $I_{\text{graph}} = \text{InstanceMap}(I_{\text{circuit}})$, our algorithm must translate this coloring into an assignment $S_{\text{assignment}} = \text{SolutionMap}(S_{\text{coloring}})$ of the variables $x_1, x_2, \dots, x_n$ for the circuit. See the last two diagrams on the bottom of Figure 28.3. The translation is accomplished by setting x_i to *true* if node v_{x_i} is colored the same color as node v_T , and *false* if the same as v_F . If node v_{x_i} has the same color as node v_R , then this is not a valid coloring (because there is an edge in the graph from node v_{x_i} to node v_R) and hence need not be considered.

Warning: Suppose that the graph constructed had a separate node for each time that the circuit used the variable x_i . The statement “set x_i to *true* when the node v_{x_i} has some color” would then be ambiguous, because the different nodes representing x_i might be given different colors.

7) Valid to Valid: Here we must prove that if the supposed algorithm gives us a valid coloring S_{coloring} for the graph $I_{\text{graph}} = \text{InstanceMap}(I_{\text{circuit}})$, then $S_{\text{assignment}} = \text{SolutionMap}(S_{\text{coloring}})$ is an assignment that satisfies the circuit. By the gadget theorem, each gadget in the graph must be colored in a way that acts like the corresponding gate. Hence, when we apply the assignment to the circuit, the output of each gate will have the value corresponding to the color of the corresponding node. It is as if the coloring of the graph were performing the computation of the circuit. It follows that the output of the circuit will have the value corresponding to the color of node v_{out} . Because node v_{out} has an edge to v_R and an extra edge to v_F , v_{out} must be colored *true*. Hence, the assignment is one for which the output of the circuit is true.

8) ReverseSolutionMap: For the proof we must also define the reverse mapping from each assignment $S_{\text{assignment}}$ to a coloring $S_{\text{coloring}} = \text{ReverseSolutionMap}(S_{\text{assignment}})$. Start by coloring the pallet nodes *true*, *false*, and the remaining color. Color each node v_{x_i} *true* or *false* according to the assignment. Then Theorem 28.3.1 states that no matter how the input nodes to a gadget are colored, the entire gadget can be colored with the output node having the color indicated by the output of the corresponding gate.

9) Reverse Valid to Valid: Now we prove that if the assignment $S_{\text{assignment}}$ satisfies the circuit, then the coloring $S_{\text{coloring}} = \text{ReverseSolutionMap}(S_{\text{assignment}})$ is valid. Theorem 28.3.1 ensured that each edge in each gadget has two different colors. The only edge

remaining to consider is the extra edge. As the colors percolate down the graph, node v_{out} must have color corresponding to the output of the circuit, which must be the true color, because the assignment satisfies the circuit. This ensures that even the extra edge from v_F to v_{out} is colored with two different colors.

10) and 11): These steps are always the same. $InstanceMap(I_{circuit})$ maps *Yes* circuit instances to *Yes* 3-COL instances and *No* to *No*. Hence, if the supposed algorithm 3-COL works correctly in polynomial time, then our designed algorithm correctly solves CIR-SAT in polynomial time. It follows that CIR-SAT $\leq_{poly}$ 3-COL. In conclusion, 3-coloring is NP-complete.

Exercise 28.3.1 (a) Complete the proof of [Theorem 28.3.1](#) by proving the cases (*true OR true*) = *true* and (*true OR false*) = *true*. (b) Prove a similar theorem for the NOT gadget. See the top right diagram in [Figure 28.3](#).

Exercise 28.3.2 Verify that each edge in the graph $I_{graph} = InstanceMap(I_{circuit})$ is needed by showing that if it were not there, then it would be possible for the graph to have a valid coloring even when the circuit is not satisfied.

Exercise 28.3.3 (See solution in Exercise Solutions.) Prove that independent set is NP-complete by proving that 3-COL $\leq_{poly}$ Independent set. (Hint: A 3-coloring for the graph G_{3-COL} can be thought of as a subset of the pairs $\langle u, c \rangle$ where u is a node of G_{3-COL} and c is a color. An independent set of the graph G_{Ind} selects a subset of its nodes. Hence, a way to construct the graph G_{Ind} in the instance $\langle G_{Ind}, N_{Ind} \rangle = InstanceMap(G_{3-COL})$ would be to have a node for each pair $\langle u, c \rangle$. Be careful when defining the edges for the graph $G_{Ind} = InstanceMap(G_{3-COL})$ so that each valid independent set of size n in the constructed graph corresponds to a valid 3-coloring of the original graph. If the constructed graph has unexpected independent sets, you may need to add more edges to it.)

28.4 An Algorithm for Bipartite Matching Using the Network Flow Algorithm

Up to now, we have been justifying our belief that certain computational problems are difficult by reducing them to other problems believed to be difficult. Here, we will give an example of the reverse, by proving that the problem of bipartite matching can be solved easily by reducing it to the network flow problem, which we already know is easy because we gave a polynomial-time algorithm for it in [Chapter 21](#).

Bipartite Matching: Bipartite matching is a classic optimization problem. As always, we define the problem by giving a set of instances, a set of solutions for each instance, and a cost for each solution.

Instances: An input instance to the problem is a bipartite graph. A bipartite graph is a graph whose nodes are partitioned into two sets, U and V , and all edges in the graph go between U and V . See the first diagram in [Figure 28.4](#).

Solutions for an Instance: Given an instance, a solution is a matching. A matching is a subset M of the edges such that no node appears more than once in M . See the last diagram in [Figure 28.4](#).

Cost of a Solution: The cost (or success) of a matching is the number of pairs matched. It is said to be a perfect matching if every node is matched.

Goal: Given a bipartite graph, the goal of the problem is to find a matching that matches as many pairs as possible.

Network Flow: Network flow is another example of an optimization problem that involves searching for a best solution from some large set of solutions.

Instances: An instance $\langle G, s, t \rangle$ consists of a directed graph G and specific nodes s and t . Each edge $\langle u, v \rangle$ is associated with a positive capacity $c_{\langle u, v \rangle}$.

Solutions for the Instance: A solution for the instance is a *flow* F , which specifies a flow $F_{\langle u, v \rangle} \leq c_{\langle u, v \rangle}$ through each edges of the network with no leaking or additional flow at any node.

Measure of Success: The cost (or success) of a flow is the amount of flow out of node s .

Goal: Given an instance $\langle G, s, t \rangle$, the goal is to find an optimal solution, that is, a maximum flow.

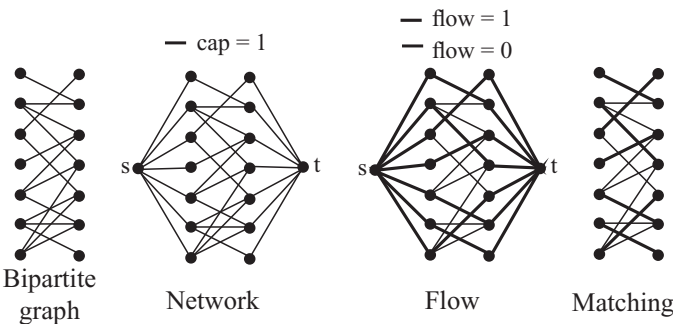


Fig. 28.4 The first diagram is the bipartite graph given as an instance to bipartite matching. The next is the network that it is translated into. The next is a flow through this network. The last is the matching obtained from the flow.

Bipartite Matching $\leq_{poly}$ Network Flows: We go through the same steps as before.

3) Direction of Reduction and Code: We will now design an algorithm for bipartite matching given an algorithm for network flows.

4) Look for Similarities: A matching decides which edges to keep, and a flow decides which edges to put flow through. This similarity suggests keeping the edges that have flow through them.

5) *InstanceMap*, Translating the Bipartite Graphs into a Network: Our algorithm for bipartite matching takes as input a bipartite graph $G_{bipartite}$. The first step is to translate this into a network $G_{network} = \text{InstanceMap}(G_{bipartite})$. See the first two diagrams in Figure 28.4. The network will have the nodes U and V from the bipartite graph, and for each edge $\langle u, v \rangle$ in the bipartite graph, the network has a directed edge $\langle u, v \rangle$. In addition, the network will have a source node s with a directed edge from s to each node $u \in U$. It will also have a sink node t with a directed edge from each node $v \in V$ to t . Every edge out of s and every edge into t will have capacity one. The edges $\langle u, v \rangle$ across the bipartite graph could be given capacity one as well, but they could just as well be given capacity ∞ .

6) *SolutionMap*, Translating a Flow into a Matching: When the network flow algorithm finds a flow S_{flow} through the network, our algorithm must translate this flow into a matching $S_{matching} = \text{SolutionMap}(S_{flow})$. See the last two diagrams in Figure 28.4.

***SolutionMap*:** The translation puts the edge $\langle u, v \rangle$ in the matching if there is a flow of one through the corresponding edge in the network, and not if there is no flow in the edge.

Warning: Be careful to map every possible flow to a matching. The above mapping is ill defined when there is a flow of $\frac{1}{2}$ through an edge. This needs to be fixed and could be quite problematic.

Integer Flow: Luckily, Exercise 21.2.4 proves that if all the capacities in the given network are integers, then the algorithm always returns a solution in which the flow through each edge is an integer. Given that our capacities are all one, each edge will have a flow either of zero or of one. Hence, in our translation, it is well defined whether to include the edge $\langle u, v \rangle$ in the matching or not.

7) Valid to Valid: Here we must prove that if the flow S_{flow} is valid, then the matching $S_{matching}$ is also valid.

Each u Matched at Most Once: Consider a node $u \in U$. The flow into u can be at most one, because there is only one edge into it and it has capacity one. For the flow to be valid, the flow out of this node must equal that into it. Hence, it too can be at most one. Because each edge out of u either has flow zero or one, it follows that at most one edge out of u has flow. We can conclude that u is matched to at most one node $v \in V$.

Each v Matched at Most Once: See Exercise 28.4.1.

Cost to Cost: To be sure that the matching we obtain contains the maximum number of edges, it is important that the cost of the matching $S_{matching} = \text{SolutionMap}(S_{flow})$ equal the cost of the flow. The cost of the flow is the amount of flow out of node s , which equals the flow across the

cut $\langle U, V \rangle$, which equals the number of edges $\langle u, v \rangle$ with flow of one, which equals the number of edges in the matching, which equals the cost of the matching.

8) ReverseSolutionMap: The reverse mapping from each matching S_{matching} to a valid flow $S_{\text{flow}} = \text{ReverseSolutionMap}(S_{\text{matching}})$ is straightforward. If the edge $\langle u, v \rangle$ is in the matching, then put a flow of one from the source s , along the edge $\langle s, u \rangle$ to node u , across the corresponding edge $\langle u, v \rangle$, and then on through the edge $\langle v, t \rangle$ to t .

9) Reverse Valid to Valid: We must also prove that if the matching S_{matching} is valid, then the flow $S_{\text{flow}} = \text{ReverseSolutionMap}(S_{\text{matching}})$ is also valid.

Flow In Equals Flow Out: Because the flow is a sum of paths, we can be assured that the flow in equals the flow out of every node except for the source and the sink. Because the matching is valid, each u and each v is matched at most once. Hence the flows through the edges $\langle s, u \rangle$, $\langle u, v \rangle$, and $\langle v, t \rangle$ will be at most their capacity one.

Cost to Cost: Again, we need to prove that the cost of the flow $S_{\text{flow}} = \text{ReverseSolutionMap}(S_{\text{matching}})$ is the same as the cost of the matching. See [Exercise 28.4.2](#).

10) and 11): These steps are always the same. $\text{InstanceMap}(G_{\text{bipartite}})$ maps bipartite graph instances to network flow instances G_{flow} with the same cost. Hence, because algorithm Alg_{flow} correctly solves network flows quickly, our designed algorithm correctly solves bipartite matching quickly.

In conclusion, bipartite matching can be solved in the same time that network flow is solved.

Exercise 28.4.1 Give a proof for the case where each v is matched at most once.

Exercise 28.4.2 Give a proof that the cost of the flow $S_{\text{flow}} = \text{ReverseSolutionMap}(S_{\text{matching}})$ is the same as the cost of the matching.

Exercise 28.4.3 [Chapter 25](#) constructs three dynamic programming algorithms using reductions. For each of these, carry out the formal steps required for a reduction.

Exercise 28.4.4 There is a collection of software packages $S_1, \dots, S_n$ that you are considering buying. These are partitioned into two groups. For those $i \in N \subseteq [n]$, the costs of buying it outweighs the benefits and hence it effectively costs you a given amount $b_i \leq 0$ to buy it. For those $j \in P \subseteq [n]$, the benefits outweigh the costs of buying it and hence it effectively costs you a given amount $b_j \leq 0$ to not buy it. Some of these packages rely on each other; if S_i relies on S_j , then you will incur an additional cost of $a_{(i,j)} \leq 0$ if you buy S_i but not S_j . Provide a polynomial-time algorithm to decide the subset $S \subseteq [n]$ of $S_2, \dots, S_n$ that you should buy. The cost of your solution is $\text{cost}(S) = \sum_{i \in S \cap N} b_i + \sum_{j \in \bar{S} \cap P} b_j + \sum_{i \in S, j \in \bar{S}} a_{(i,j)}$. (Hint: Do not design a new algorithm but do a reduction to min cut similar to that done for matching the boys and girls.)

29 Randomized Algorithms

For some computational problems, allowing the algorithm to flip coins (i.e., use a random number generator) makes for a simpler, faster, easier-to-analyze algorithm. The following are the three main reasons.

Hiding the Worst Cases from the Adversary: The running time of a randomized algorithm is analyzed in a different way than that of a deterministic algorithm. At times, this way is fairer and more in line with how the algorithm actually performs in practice. Suppose, for example, that a deterministic algorithm quickly gives the correct answer on most input instances, yet is very slow or gives the wrong answer on a few instances. Its running time and its correctness are generally measured to be those on these worst case instances. A randomized algorithm might also sometimes be very slow or give the wrong answer. (See the discussion of quick sort, [Section 12.1](#).) However, we accept this, as long as on every input instance, the probability of doing so (over the choice of random coins) is small.

Probabilistic Tools: The field of probabilistic analysis offers many useful techniques and lemmas that can make the analysis of the algorithm simple and elegant.

Solution Has a Random Structure: When the solution that we are attempting to construct has a random structure, a good way to construct it is to simply flip coins to decide how to build each part. Sometimes we are then able to prove that with some probability the solution obtained this way has better properties than any solution we know how to construct deterministically. For example, if we can prove that the solution constructed randomly has extremely good properties with some very small but nonzero probability (for example, $prob = 10^{-100}$), then this proves the existence of such a solution even though we have no reasonably quick way of finding one. Another interesting situation is when the randomly constructed solution very likely has the desired properties, for example, with probability 0.999999, but there is no quick way of testing whether what we have produced has the desired properties.

This chapter considers these ideas further.

29.1 Using Randomness to Hide the Worst Cases

The standard way of measuring the running time and correctness of a deterministic algorithm is based on the worst case input instance chosen by some nasty adversary

who has studied the algorithm in detail. This is not fair if the algorithm does very well on all but a small number of very strange and unlikely input instances. On the other hand, knowing that the algorithm works well on most instances is not always satisfactory, because for some applications it is just those hard instances that you want to solve. In such cases, it might be more comforting to use a randomized algorithm that guarantees that on every input instance, the correct answer will be obtained quickly with high probability.

A randomized algorithm is able to flip coins as it proceeds to decide what actions to take next. Equivalently, a randomized algorithm A can be thought of as a set of deterministic algorithms $A_1, A_2, A_3, \dots$, where A_r is what algorithm A does when the outcome of the coin flips is $r = \langle \text{heads}, \text{tails}, \text{heads}, \text{heads}, \dots, \text{tails} \rangle$. Each such deterministic algorithm A_r will have a small set of worst case input instances on which it either gives the wrong answer or runs too slow. The idea is that these algorithms $A_1, A_2, A_3, \dots$ have different sets of worst case instances. This randomized algorithm is good if for each input instance, the fraction of the deterministic algorithms $A_1, A_2, A_3, \dots$ for which it is not a worst case instance is at least p . Then when one of these A_r 's is chosen randomly, it solves this instance quickly with probability at least p .

I sometimes find it useful to consider the analysis of randomized algorithms as a game between an algorithm designer and an adversary who tries to construct input instances that will be bad for the algorithm. In the game, it is not always fair for the adversarial input chooser to know the algorithm first, because then they can choose the instance that is the worst case for this algorithm. Similarly, it is not always fair for the algorithm designer to know the input instance first or even which instances are likely, because then they can design the algorithm to work well on these. The way we analyze the running time of randomized algorithms compromises between these two. In this game, the algorithm designer, without knowing the input instance, must first fix what their algorithm will do given the outcome of the coins. Knowing this, but not knowing the outcomes of the coins, the instance chooser chooses the worst case instance. We then flip coins, run the algorithm, and see how well it does.

Three Models: The following are formal definitions of three models.

Deterministic Worst Case: In a worst case analysis, a deterministic algorithm A for a computational problem P must always give the correct answer quickly:

$$\forall I, [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq T_{\text{upper}}(|I|)].$$

Las Vegas: The algorithm is said to be *Las Vegas* if it is always guaranteed to give the correct answer, but its running time depends on the outcomes of the random coin flips. The goal is to prove that on every input instance, the expected running time is small:

$$\forall I, [\forall r, A_r(I) = P(I) \text{ and } \text{Exp}_r[\text{Time}(A_r, I)] \leq T_{\text{upper}}(|I|)].$$

Monte Carlo: The algorithm is said to be *Monte Carlo* if the algorithm is guaranteed to stop quickly, but it can sometimes, depending on the outcomes

of the random coin flips, give the wrong answer. The goal is to prove that on every input, the probability of it giving the wrong answer is small:

$$\forall I, [Pr_r[A_r(I) \neq P(I)] \leq p_{\text{fails}} \text{ and } \forall r, \text{Time}(A_r, I) \leq T_{\text{upper}}(|I|)].$$

The following examples demonstrate these ideas.

Quick Sort: Recall the quick sort algorithm from [Section 12.1](#). The algorithm chooses a pivot element and partitions the list of numbers to be sorted into those that are smaller than the pivot and those that are larger than it. Then it recurses on each of these two parts. The running time varies from $\Theta(n \log n)$ to $\Theta(n^2)$, depending on the choices of pivots.

Deterministic Worst Case: A reasonable choice for the pivot is to always use the element that happens to be located in the middle of the array to be sorted. For all practical purposes, this would likely work well. It would work exceptionally well when the list is already sorted. However, there are some strange inputs, cooked up for the sole purpose of being nasty to this particular implementation of the algorithm, on which the algorithm runs in $\Theta(n^2)$ time. The adversary will provide such an input, giving a worst case time complexity of $\Theta(n^2)$.

Las Vegas: In practice, what is often done is to choose the pivot element randomly from the input elements. This makes it irrelevant in which order the adversary puts the elements in the input instance. The expected computation time is $\Theta(n \log n)$.

The Game Show Problem: The input I to the game show problem specifies which of N doors have prizes behind them. At least half the doors are promised to have prizes. An algorithm A is able to look behind the doors in any order that it likes, but nothing else. It solves the problem correctly when it finds a prize. The running time is the number of doors opened.

Deterministic Worst Case: Any deterministic algorithm fixes the order in which it looks behind the doors. Knowing this order, the adversary places no prizes behind the first $\frac{N}{2}$ doors looked behind.

Las Vegas: In contrast, a random algorithm will look behind doors in random order. It does not matter where the adversary puts the prizes; the probability that one is not found after t doors is $1/2^t$, and the expected time until a prize is found is $\text{Exp}[T] = \sum_t Pr[T = t] \cdot t = 2$.

Monte Carlo: If the promise is that either at least half the doors have prizes or none of them do and if the algorithm stops after 10 empty doors and claims that there are no prizes, then this algorithm is always fast, but gives the wrong answer with probability $1/2^{10}$.

Randomized Primality Testing: An integer x is said to be *composite* if it has factors other than one and itself. Otherwise, it is said to be *prime*. For example, $6 = 2 \times 3$ is

composite and 2, 3, 5, 7, 11, 13, 17, ... are prime. See [Section 35.1](#), Example 35.2, for explanations of why it takes $2^{\Theta(n)}$ time to factor an n -bit number.¹ Here I give an easy randomized algorithm, due to Rabin and Miller, for this problem.

Fermat's Little Theorem: Don't worry about the math, but Fermat's little theorem says that if x is prime, then for every $a \in [1, x-1]$, it is the case that $a^{x-1} \equiv_{(\text{mod } x)} 1$.

If we want to test if x is prime, then we can pick random a s in the interval and see if the equality holds. If the equality does not hold for a value of a , then x is composite. If the equality does hold for many values of a , then we can say that x is probably prime, or what we call a *pseudoprime*.

The Game Show Problem: Finding an a for which $a^{x-1} \not\equiv_{(\text{mod } x)} 1$ is like finding a prize behind door a . See [Exercise 29.1.1](#).

Randomized Counting: In many applications, one wants to count the number of occurrences of something. This problem can often be expressed as follows: Given the input instance x , count the number of y for which $f(x, y) = 1$. It is likely very difficult to determine the exact number. However, a good way to approximate this number is to randomly choose some large number of values y . For each, test whether $f(x, y) = 1$. Then the fraction of y for which $f(x, y) = 1$ can be approximated by [the number you found]/[the number you tried]. The number of y for which $f(x, y) = 1$ can be approximated by [the fraction you found] $\times$ [the total number of y].

For example, suppose you had some strange shape and you wanted to find its area. Then x would specify the shape, y would specify some point within a surrounding box, and $f(x, y) = 1$ if the point is within the shape. Then the number of y for which $f(x, y) = 1$ gives you the area of your shape.

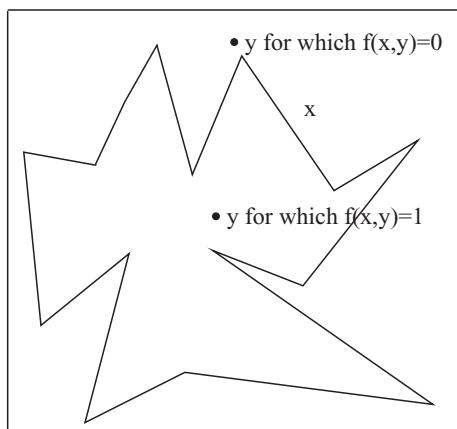


Fig. 29.1 A strange shape of which to find the area by randomized counting.

¹ A major breakthrough by Agrawal et al. in 2002 was to find a polynomial-time deterministic algorithm for determining whether an n -bit number is prime.

Exercise 29.1.1 Given an integer x , suppose that you have one door for each $a \in [1, x - 1]$. We will say that there is a prize behind this door if $a^{x-1} \not\equiv_{(\text{mod } x)} 1$. Fermat's little theorem says that if x is pseudoprime, then none of the doors have prizes behind them, and if it is composite, then at least half the doors have prizes. The algorithm attempts to determine which is the case by opening t randomly chosen doors for some integer t .

1. If the algorithm finds a prize, what do you know about the integer? If it does not find a prize, what do you know?
2. If the algorithm must always give the correct answer, how many doors need to be opened, as a function of the number n of digits in the instance x ?
3. If t doors are open and the input instance x is a pseudoprime, what is the probability that the algorithm gives the correct answer? If the instance is composite, what is this probability?

Exercise 29.1.2 Section 4.3 designed an iterative algorithm for separating n VLSI chips into those that are good and those that are bad by testing two chips at a time and learning either that they are the same or that they are different. To help, at least half of the chips are promised to be good. Now design (much easier) a randomized algorithm for this problem. Here are some hints.

- Randomly select one of the chips. What is the probability that the chip is good?
- How can you learn whether or not the selected chip is good?
- If it is good, how can you easily partition the chips into good and bad chips?
- If the chip is not good, what should your algorithm do?
- When should the algorithm stop?
- What is the expected running time of this algorithm?

29.2 Solutions of Optimization Problems with a Random Structure

Optimization problems involve looking for the best solution for an instance. Sometimes good solutions have a random structure. In such cases, a good way to construct one is to simply flip coins to decide how to build each part. I give two examples. The first one, *max cut*, being NP-complete, likely requires exponential time to find the best solution. However, in $O(n)$ time, we can find a solution that is likely to be at least half as good as optimal. The second example, *expander graphs*, is even more extreme. Though there are deterministic algorithms for constructing graphs with fairly good expansion properties, a random graph almost for sure has much better expansion properties (with probability $p \geq 0.999999$). A complication, however, is that there is no polynomial-time algorithm that tests whether this randomly constructed graph has the desired properties. Pushing the limits further, it can be proved that the same random graph has extremely good properties with some very small but nonzero probability (e.g., $p \geq 10^{-100}$). Though we have no quick way to construct such a graph, this does prove that such a graph exists.

The Max Cut Problem: The input to the max cut problem is an undirected graph. The output is a partition of the nodes into two sets, U and V , such that the number of edges that cross over from one side to the other is as large as possible. This problem is NP-complete, and hence the best known algorithm for finding an optimal solution requires $2^{\Theta(n)}$ time. The following randomized algorithm runs in time $\Theta(n)$ and is expected to obtain a solution for which half the edges cross over. This algorithm is incredibly simple. It simply flips a coin for each node to decide whether to put it into U or into V . Each edge will cross over with probability $\frac{1}{2}$. Hence, the expected number of edges to cross over is $\frac{|E|}{2}$. The optimal solution cannot have more than all the edges cross over, so the randomized algorithm is expected to perform at least half as well as the optimal solution can do.

Expander Graphs: An n -node degree- d graph is said to be an *expander graph* if moving from a set of its nodes across its edges expands us out to an even larger set of nodes. More formally, for $0 < \alpha < 1$ and $1 < \beta < d$, a graph $G = \langle V, E \rangle$ is an $\langle \alpha, \beta \rangle$ -expander if for every subset $S \subseteq V$ of its nodes, if $|S| \leq \alpha n$, then $|N(S)| \geq \beta |S|$. Here, $N(S)$ is the neighborhood of S , that is, the set of all nodes with an edge from some node in S .

Nonoverlapping Sets of d Neighbors: Because each node $v \in V$ has d neighbors $N(v)$, a set S has $d|S|$ edges leaving these nodes. However, if these sets $N(v)$ of neighbors overlap a lot, then the total number of neighbors $N(S) = \bigcup_{v \in S} N(v)$ of S might be very small. We can't expect $N(S)$ to be bigger than $d|S|$, but we do want it to have size at least $\beta|S|$ where $1 < \beta < d$. If S is too big, we can't expect it to expand further. Hence, we only require this expansion property for sets S of size at most αn . Because we do expect sets of size αn to expand to a neighborhood of size $\beta \alpha n$, we require that $\alpha \beta < 1$.

Connected with Short Paths: If $\alpha \beta > \frac{1}{2}$, then every pair of nodes in G is connected with a path of length at most $\frac{2 \log(n/2)}{\log \beta}$.

Proof: Consider two nodes u and v . The node u has d neighbors, $N(u)$. These neighbors $N(u)$ must have at least $\beta|N(u)| = \beta d$ neighbors $N(N(u))$. These neighbors $N(N(u))$ must have at least $\beta^2 d$ neighbors. It follows that there are at least $\beta^{i-1} d$ nodes with distance i from u . The last time we are allowed to do this expands the neighbor set of size $|S| = \alpha n$ to $|N(S)| \geq \beta|S| = \beta \alpha n$. By the requirement that $\alpha \beta > \frac{1}{2}$, this new neighbor set has size greater than $\frac{n}{2}$ nodes. The distance of these nodes from u is at most $i = \log_{\beta} \frac{n}{2}$. This set might not contain v . However, starting from v there is another set of more than half the nodes that are distant $i = \log_{\beta} \frac{n}{2}$ from v . These two sets must overlap at some node w . Hence, there is a path from u to w to v of length at most $\frac{2 \log(n/2)}{\log \beta}$.

Uses: Expander graphs are very useful both in practice and for proving theorems.

Fault-Tolerant Networks: As we have seen, every pair of nodes in an expander graph is connected. This is still true if a large number of nodes

or edges fail. Hence, this is a good pattern for wiring a communications network.

Pseudorandom Generators: Taking a short random walk in an expander graph quickly gets you to a random node. This is useful for generating long random-looking strings from a short seed string.

Concentrating and Recycling Random Bits: If we have a source that has some randomness in it (say, n coin tosses with an unknown probability and with unknown dependences between the coins), we can use expander graphs to produce a string of m bits appearing to be the result of m fair and independent coins.

Error-Correcting Codes: Expander graphs are also useful in designing ways of encoding a message into a longer code so that if any reasonable fraction of the longer code is corrupted, the original message can still be recovered. The faulty bits are connected by short paths to correct bits.

If $\alpha\beta < 1$, Then Expander Graphs Exist: We will now prove that for any constants α and β for which $\alpha\beta < 1$ there exists an (α, β) -expander graph with n nodes and degree d for some sufficiently big constant d . For example, if $\alpha = \frac{1}{2}$, $\beta = \frac{3}{2}$, then $d = 5$ is sufficient. To make the analysis easier, we will consider directed graphs where each node u is connected to d nodes chosen independently at random. (If we ignore the directions of the edges, then each node has average degree $2d$ and neighborhood sets are only bigger.) We prove that the probability we do not get such an expander graph is strictly less than one. Hence, one must exist.

Event $E_{S,T}$: The graph G will not be a (α, β) -expander if there is some set S for which $|S| \leq \alpha n$ and $N(S) < \beta|S|$. Hence, for each pair of sets S and T , with $|S| \leq \alpha n$ and $|T| < \beta|S|$, let $E_{S,T}$ denote the bad event that $N(S) \subseteq T$. Let us bound the probability of $E_{S,T}$ when we choose G randomly. Each node in S needs d neighbors, for a total of $d|S|$ randomly chosen neighbors. The probability of a particular one of these landing in T is $|T|/n$. Because these edges are chosen independently, the probability of them all landing in T is $(|T|/n)^{d|S|}$.

Probability of Some Bad Event: The probability that G is not an expander is the probability that at least one of these bad events $E_{S,T}$ happens, which is at most the sum of the probabilities of these individual events:

$$\begin{aligned} \Pr[G \text{ not an expander}] &= \Pr[\text{At least one of the events } E_{S,T} \text{ occurs}] \\ &\leq \sum_{S,T} \Pr[E_{S,T}] = \sum_{(s \leq \alpha n)} \sum_{(|S|=s)} \sum_{(|T|=\beta s)} \Pr[E_{S,T}] \\ &= \sum_{s \leq \alpha n} \binom{n}{s} \binom{n}{\beta s} \left(\frac{|T|}{n}\right)^{d|S|}. \end{aligned}$$

We now use the result that $\binom{n}{a} \leq \left(\frac{en}{a}\right)^a$:

$$\begin{aligned} \Pr[G \text{ not an expander}] &\leq \sum_{s \leq \alpha n} \left(\frac{en}{s}\right)^s \left(\frac{en}{\beta s}\right)^{\beta s} \left(\frac{\beta s}{n}\right)^{ds} = \sum_{s \leq \alpha n} \left[\left(\frac{en}{s}\right) \left(\frac{en}{\beta s}\right)^{\beta} \left(\frac{\beta s}{n}\right)^d \right]^s \\ &\leq \sum_{s \leq \alpha n} \left[\left(\frac{en}{\alpha n}\right) \left(\frac{en}{\beta \alpha n}\right)^{\beta} \left(\frac{\beta \alpha n}{n}\right)^d \right]^s = \sum_{s \leq \alpha n} \left[\frac{e^{\beta+1}}{\alpha} \cdot (\alpha \beta)^{d-\beta} \right]^s. \end{aligned}$$

The requirement is that $\alpha\beta < 1$. Hence, if d is sufficiently big ($d \geq \log(2e^{\beta+1}/\alpha)/\log(1/\alpha\beta + \beta)$), then the bracketed summand is at most $\frac{1}{2}$:

$$\Pr[G \text{ not an expander}] \leq \sum_{s \leq \alpha n} \left[\frac{1}{2} \right]^s < 1.$$

It follows that $\Pr[G \text{ is an expander}] > 0$, meaning that there exists at least one such G that is an expander.

30 Machine Learning

Computers can now drive cars and find cancer using X-rays. For better or worse, this will change the world (and the job market). Strangely, designing these algorithms is not done by telling the computer what to do or even by understanding what the computer does. The computers learn themselves from lots and lots of data and lots of trial and error. This learning process is more analogous to how brains evolved over billions of years of learning. The machine itself is a neural network that models both the brain and silicon and-or-not circuits, both of which are great for computing. The only difference with neural networks is that what they compute is determined by weights, and small changes in these weights give you small changes in the result of the computation. The process for finding an optimal setting of these weights is analogous to finding the bottom of a valley. “Gradient descent” achieves this by using the local slope of the hill (derivatives) to direct the travel down the hill, i.e., small changes to the weights. There is some theory. If a machine is found that gives the correct answers on the randomly chosen training data without simply memorizing, then we can prove with high probability this same machine will also work well on never-seen-before instances.

Coding: When writing computer code, the instructions are painstakingly written by humans. This works great for simple repetitive tasks. Expert systems of the 1980s, however, failed. Though we are able to walk, it is hard to explain how to do it!

Machine Learning: When designing neural networks, the instructions are 100% dictated by weights $\langle w_1, \dots, w_m \rangle$. Like the brain, these are way too hard to understand. They are learned using a simple algorithm, finding patterns in lots of data. Walking, for example, has enough of a pattern that machine learning can copy it.

Evolution: Add just a little random change to a system that is encouraged by some feedback to go in the right direction and the *Emergent Complexity* that arises is awe inspiring.

Hopeful Applications: Self-driving cars, image processing, speech processing, robots, art, medical, financial, and legal experts. Hopefully, these will make the world a better place.

Scary Applications: People losing their jobs, the system watching you, ads customized to you, cybersecurity picking out “criminals” in a crowd, killing machines, machines deciding your fate. These might make the world a worse place.

Rich and Powerful: I do not believe that AI is the threat. For me, the question is whether the people who own the AI will be benevolent to the rest of us when they become even richer, more powerful, and control robots with guns.

Consciousness: There is a lot of talk about whether the large language models are conscious. They take inputs and compute outputs that sound like a person would sound. Some people believe that humans too are just computing machines. The following is a possible example and evolutionary purpose of *consciousness*: As an early cave person, it would have been important for me to be able to learn from direct experience, from understanding the experiences and emotions of others, and from thought experiments. In my mind, for example, I can imagine running into a lion. How do I know whether it wants to eat me and whether it knows I am here? Does the lion know that I know that it is there? Does the lion know that I know that it knows that I am here? All of these require a model of me and of the lion as separate knowing beings. A crow can do this. Surely an AI can. Might this be consciousness?

Abstract Thinking: We can talk about many aspects of machine learning without needing to know the nature of the machine or the nature of the data. We talk of knowing if an image is a face without mentioning noses. This simplicity lets us focus on what is important and ensures that what we say works for any model of machine and for any computational problem.

Supervised Training Data: The input that we receive is a set of input/output pairs $\{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots, (\vec{x}_D, y_D)\}$. In order to be able to draw a simple graph for intuitive purposes, one might pretend each input $\vec{x}_d$ is a real number. But more likely it is something complex like an image. For each such input, y_d is the answer provided by a *Supervisor*. When y_d is a single real number, we call the process *regression*. When y_d is a label like cat or dog, we call the process *categorizing*. Another way of visualizing the data is to assume that each input $\vec{x}_d$ is some point in some high dimensional space and the supervisor's answer y_s is indicated by the color of the point.

Cat: If the input $\vec{x}_d$ is an image of a cat, then the computer just sees a big matrix of numbers. From this, meaning must be extracted. This seems to me to be magic. On the other hand, the brain manages to do it.

Machine: Our goal is to build a machine $M_{\vec{w}}(\vec{x})$ that takes an input $\vec{x}$ and returns an answer y . It is parametrized by a vector of m real valued weights $\vec{w} = \langle w_1, \dots, w_m \rangle$. This includes anything that the learning process learns and remembers about the data and uses later to make predictions.

Linear and Non-Linear Regression: For example, y_d might be the likelihood of rain on day d , $x_{(d,1)}$ whether there are clouds, $x_{(d,2)}$ which color of coat the man is wearing, and $\langle w_1, w_2 \rangle$ their level of effect on rain. Our machine might then approximate y_d with $M_{\vec{w}}(\vec{x}_d) = w_0 + w_1 x_{(d,1)} + w_2 x_{(d,2)}$. Who knows, maybe if people are wearing yellow coats, then these might be raincoats and hence it is more likely to rain. The model designer could make the types of machines learned more powerful by adding additional terms like $+w_3 x_{(d,2)}^2$. Here $x_{(d,2)}^2$ is a hand-picked precomputed feature of the input

$\vec{x}_d$, and w_3 is its weight that is learned during the learning process. Though this machine is now non-linear in the input $\vec{x}_d$, we still call this linear regression, because the answer is linear in the learned weights w_3 . By contrast, learning $M_{\vec{w}}(\vec{x}_d) = x_{(d,1)}^{w_1} + x_{(d,2)}^{w_2}$, being non-linear in the weights, would be much harder.

Neural Networks: Our machine might also be something much more complex like a *neural network*. All you need to know now about it is that how it computes $y = M_{\vec{w}}(\vec{x})$ is dictated by its weights $\vec{w} = \langle w_1, \dots, w_m \rangle$.

Error: Given some setting $\vec{w} = \langle w_1, \dots, w_m \rangle$ of the weights, we want to measure how well the machine $M_{\vec{w}}(\vec{x})$ does at getting the correct answers y on the training data $\{\langle \vec{x}_1, y_1 \rangle, \langle \vec{x}_2, y_2 \rangle, \dots, \langle \vec{x}_D, y_D \rangle\}$. The easiest thing to do would be to count the fraction of the data set for which the machine gets the correct answer, namely, $y_d = M_{\vec{w}}(\vec{x}_d)$. However, this count does not vary continually with small changes in the machine. If answer y is a real number, the difference $y_d - M_{\vec{w}}(\vec{x}_d)$ gives us a measure of how close our machine is to the correct answer. We could take the absolute value of this $|y_d - M_{\vec{w}}(\vec{x}_d)|$ because we do not care which direction the error is in, but $|z|$ is too pointy at $z = 0$, making the math nearly impossible. Instead, we square it and sum over all data items, namely, $E(\vec{w}) = \sum_d (y_d - M_{\vec{w}}(\vec{x}_d))^2$. When the answer y is a category like “cat,” we can have the machine instead produce a real number telling us the “probability” that it thinks it is a “cat.” The error would then be based on how close the probability of the correct answer is to one.

Machine Learning: Given the training data, our goal is to find the weights $\vec{w}_{opt}$ that minimize the error $E(\vec{w})$. This is all that machine learning is.

Error Surface: We think of the error as a function of the weights $\vec{w}$ because they are the parameters that we are able to tweak. If we pretend that the weights consist of one real number $\vec{w} = \langle w_1 \rangle$, then we can graph the error function on paper with w_1 being the x -axis and the error $E(\vec{w})$ being the y -axis. We are looking for a minimum. With two real numbers $\vec{w} = \langle w_1, w_2 \rangle$, we get an error surface. The first weight w_1 can tell you your East–West location, the second w_2 your North–South location, and the error $E(\vec{w})$ your altitude. This gives you a topological map of your error surface. We are looking for a location $\vec{w}$ in a valley.

Blind: In practice, the weights $\vec{w} = \langle w_1, \dots, w_m \rangle$ consist of hundreds of billions of real numbers making for a very high dimensional space to search. With 10,000 weights with 0 to 9 integer values, there are $10^{10,000}$ settings of the weights. A supercomputer on each atom of the universe working for the age of the universe couldn’t dent this list!

Linear Regression: If the machines are linear in the weights as in $M_{\vec{w}}(\vec{x}_d) = w_0 + w_1 x_{(d,1)} + w_2 x_{(d,2)} + w_3 x_{(d,2)}^2$, then there are formulas that will give you the weights $\vec{w}_{opt}$ that minimize the error $E(\vec{w})$. Not so for complicated machines.

Gradient Descent: When I asked my young son how to find the top of the hill, he answered, “Just keep going up.” When asked in which direction to head, he answered, “In the direction that is steepest.” When asked how you know when you are at the top, he answered, “When you can’t go up anymore.” This is how we will find the bottom of

a valley. It might not find the *global* minimum, but we hope that it finds a machine that is good enough for our purposes. From this we get self-driving cars!

Smooth/Differentiable: This error surface is such that an infinitesimally small change in any one of the weights w_k causes an infinitesimally small change in the output $M_{\vec{w}}(\vec{x}_d)$ of the machine and in the error $E(\vec{w}) = \sum_d (y_d - M_{\vec{w}}(\vec{x}_d))^2$. Dividing the change in $E(\vec{w})$ by the change in w_k gives the derivative $\frac{\delta E}{\delta w_k}$. The vector of these derivatives $\Delta(\vec{w}) = \left\langle \frac{\delta E}{\delta w_1}, \frac{\delta E}{\delta w_2}, \dots, \frac{\delta E}{\delta w_m} \right\rangle$ is called the *gradient*. This is computed efficiently by a process called *back-propagation* using the calculus learned in high school. Intuitively, the weights w_k that influence correct answers are increased and those that influence wrong answers are decreased.

Algorithm: Method for finding weights $\vec{w}_{opt}$ that minimizes the error $E(\vec{w})$ is as follows:

Start with random $\vec{w} = \langle w_1, \dots, w_m \rangle$

Repeat:

 (loop-invariant): Given current weights, $\vec{w} = \langle w_1, \dots, w_m \rangle$.

 Calculate our height $E(\vec{w})$.

 Calculate the direction and slope of steepest descent.

 Change $\vec{w}$ by a small step in this direction and distance proportional to the slope.

 Stop when every direction is up.

Complexity of Machines: One measure of the complexity of the machines $\{M_{\vec{w}}\}$ that machine learning considers is the number of weights m . Generally, if you increase this, the machine $M_{\vec{w}}$ learned gets the answer wrong for a smaller percentage of your training data. See [Figure 30.1](#).

Generalizing: An even more important question, however, is how well the learned machine generalizes to other inputs that it has never seen before. There is a “no free lunch” theorem that says that no one method works in all cases. But there are a few principles. Consider the following three examples, A, B, C. Each dot gives the $\langle x_d, y_d \rangle$ of one training data item. The curve gives the answer that the machine gives for each input x . Ask yourself which of these three gives the best answers for values of x that it has never seen before.

Underfitting: A is an example of *Underfitting*, in which the class of machines just doesn’t have the capacity to learn the material.

Overfitting: C is an example of *Overfitting*, in which the class of machines is so powerful that it can memorize the answers for any function. The problem is that if you cheat on the exam by making a table of all the answers you were taught, then you learn nothing and are stuck when a new question comes up.

Compression: B seems like a good balance of under- and overfitting. On a test, if you summarize the learned material, compressing it all to fit on one “cheat sheet,” then hopefully you *understand* it and do well when a new question comes up. Compression

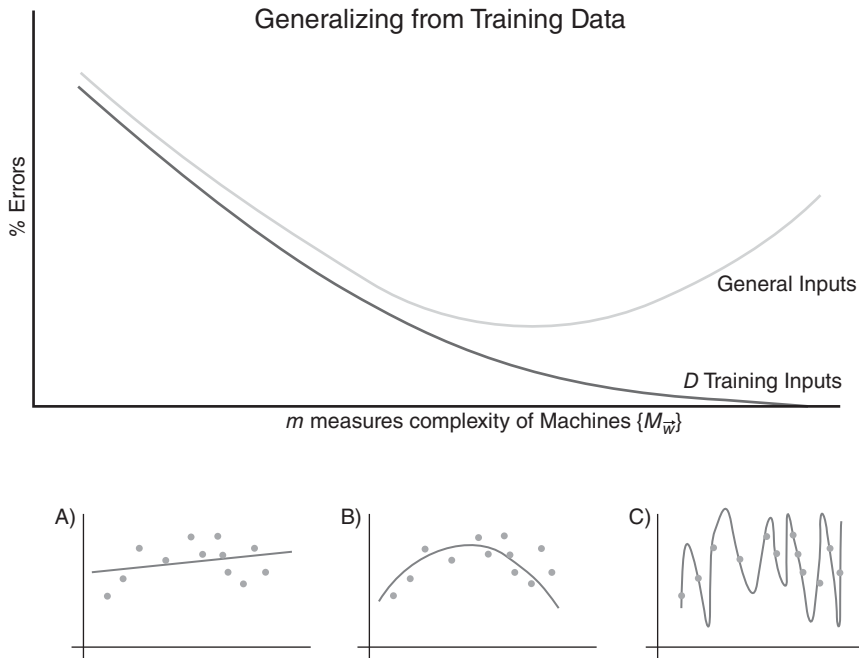


Fig. 30.1 The top graph shows that as the architecture of the machine gets more complex (i.e., described by more weights), the fraction of the training data on which the machine gives the wrong answer goes down. The dots in graphs A, B, C plot the training data with the input x being the horizontal axis and the supervisor's answer being the vertical axis. In contrast, the curve is the answer given by the learned machine. Graph A gives an example where the machine is not complex enough to model the data and hence under-fits, B models it well, and C gets the training data perfectly, but because it over-fits, it likely does not do well on data never seen before.

occurs when the weights $\vec{w} = \langle w_1, \dots, w_m \rangle$ with which the machine learns does not contain enough bits of information to store the training data answers $\langle y_1, \dots, y_D \rangle$.

Regularization: A common way to balance computing the right answers on the training data and not overfitting i is to minimize not just the error but a sum of the error and of some measure of how complex your machine is. William of Ockham said, "All things being equal, the simplest solution tends to be the best one."

Theory: Learnable Probably Approximately Correct (PAC): If on the randomly chosen training data, a machine is found that gives good answers, even though compression is required, then we can prove that with high probability this same machine will also work well on never-seen-before instances. The proof does not measure the quality of the machine but of the training data. If the set of data is chosen randomly, then with high probability it is *representative* of the entire universe of possible data.

Neural Networks: Let's now understand neural networks in more detail. A neural network is made of layers of artificial neurons. The input layer has an input wire for

each pixel of the input image receiving a value between 0 and 1, indicating the pixel's brightness. The output layer has a wire for each category, indicating the "probability" that the input image is in fact a cat or dog. In between, the layers are said to be *hidden*. These have no human designated meaning. Meaning is "learned" by choosing weights $\vec{w} = \langle w_1, \dots, w_m \rangle$.

When a *neuron* in the brain fires, it sends a signal to its neighbors across its synapses. These synapses have different weights in their influence. When a neuron's incoming signal reaches a threshold, it fires. Similarly, an artificial neuron takes a real valued signal, x_i , along each of its incoming edges, multiplies this by the edge's weight, w_i , and fires if the weighted sum $z = \sum_{i=0 \dots n} x_i w_i$ reaches its threshold. In this way, a neuron is able to compute an *And*, *Or*, or *Not* gate and hence is powerful enough to be able to compute anything a digital computer can compute. So we can later do gradient descent, we change the threshold activation $y = \sigma(z)$ so that it transitions gracefully. Instead of YES/NO we allow maybe so, i.e., if the weighted sum is close to its threshold, then it outputs a half instead of 0 or 1.

Vectors: Let's try to understand how to best think about all of these numbers. A data input $\vec{x} = \langle x_1, \dots, x_n \rangle$, e.g., an image of a cat, is just a large tuple of real values. As such, it can be thought of as a point in some high dimensional vector space. Whether the image is of a cat or a dog partitions this vector space into regions. Classifying your image amounts to knowing which region the corresponding point is in. A *linear separator* separates two regions with a plane. It is unlikely that this will work to separate cats from dogs. Instead, each layer of the neural network transforms the input $\vec{x}$ until it can categorize it.

Correlation of Vectors: Machine learning is all about finding complex correlations. Suppose I am starting a dating service. For each client I will collect properties. For each pair of clients, we measure how compatible they are. The first two clients in [Table 30.1](#) below each like movies a lot. This adds $15 \times 20 = 300$ to their compatibility. They are both neutral on nature, adding only $3 \times 2 = 6$. They differ on being extroverted, making them $(-20) \times 30 = -600$ incompatible. Summing over all properties gives a total compatibility of -874 , i.e., incompatible. The first and third client with 910 are much more compatible.

Denote the vector/tuple of values for the first client with $\vec{w} = \langle w_1, \dots, w_n \rangle$ and for the second with $\vec{x} = \langle x_1, \dots, x_n \rangle$; their compatibility $(\vec{w}, \vec{x})$ is computed as $\sum_i w_i x_i$. This is called the *dot product* $\vec{w} \cdot \vec{x}$ between these vectors. If these vectors are thought of as arrows in high dimensional space, then the angle between them is given by $\cos(\Theta) = \frac{\vec{w} \cdot \vec{x}}{|\vec{w}| \cdot |\vec{x}|}$. If $\vec{w} \cdot \vec{x}$ is positive, the angle Θ is less than 90 degrees, and if it is negative, then it is greater. If you shine a light perpendicular down onto $\vec{w}$, the length of $\vec{x}$'s projection will be $|\vec{x}| \cdot \cos(\Theta) = \frac{\vec{w} \cdot \vec{x}}{|\vec{w}|}$. For some given distance d , the set of all vectors $\vec{x}$ with this projection length equaling d are those within the hyperplane perpendicular to $\vec{w}$ a distance d from the origin. When $\frac{\vec{w} \cdot \vec{x}}{|\vec{w}|}$ is less than this d , then $\vec{x}$ is on the near side of this plane and when greater, then it is on the far side.

Linear Layer = Matrix Multiplication: The simplest layer of a neural network is linear. A novice reading a machine learning paper might not understand that many

Table 30.1

	Likes animals	Extrovert	Partier	Likes movies	Likes nature	Likes music
Person 1	-10	-20	-20	15	3	-9
Person 2	-9	30	20	20	2	30
Person 3	-8	-10	-15	10	30	-10
Person 1 $\times$ Person 2	$(-10) \times$ $(-9) = +90$	$(-20) \times$ $30 = -600$	$(-20) \times$ $20 = -400$	$15 \times$ $20 = 300$	$3 \times 2 = 6$	$(-9) \times$ $30 = -270$
Total correlation: -874						
Person 1 $\times$ Person 3	$(-10) \times$ $(-8) = 80$	$(-20) \times$ $(-10) = 200$	$(-20) \times$ $(-15) =$ 300	$15 \times$ $10 = 150$	$3 \times 30 =$ 90	$(-9) \times$ $(-10) = 90$
Total correlations: 910						

of the symbols are not real numbers but are matrices. Hence, the product of two such symbols is matrix multiplication.

Consider feeding the d^{th} training data input $\vec{x}_d = \langle x_{\langle d,1 \rangle}, \dots, x_{\langle d,m \rangle} \rangle$ into the neural network. Its i^{th} value $x_{\langle d,i \rangle}$ is fed into the network's i^{th} input wire. The set of all of these values $x_{\langle d,i \rangle}$ can be organized into a matrix $X = [x_{\langle d,i \rangle}]_{\langle d,i \rangle}$ whose d^{th} row corresponds this d^{th} training data input $\vec{x}_d$.

Let $w_{\langle i,j \rangle}$ denote the weight of the edge from the i^{th} input wire to the j^{th} artificial neuron in the first layer of the network. These values can be also be organized into a matrix $W = [w_{\langle i,j \rangle}]_{\langle i,j \rangle}$.

The total incoming signal into the j^{th} hidden neuron on the d^{th} input $\vec{x}_d$ is denoted $z_{\langle d,j \rangle}$. Again, these values form a matrix $Z = [z_{\langle d,j \rangle}]_{\langle d,j \rangle}$.

This j^{th} hidden neuron has many incoming edges, the i^{th} of which is labeled with weight $w_{\langle i,j \rangle}$ and receives the input value $x_{\langle d,i \rangle}$. These are multiplied and these products for the different incoming edges are added together, i.e., $z_{\langle d,j \rangle} = x_{\langle d,1 \rangle} \times w_{\langle 1,j \rangle} + \dots + x_{\langle d,n \rangle} \times w_{\langle n,j \rangle} = \sum_i x_{\langle d,i \rangle} \times w_{\langle i,j \rangle}$. Note that this is the exact calculation computed in the matrix multiplication $Z = X \times W$ as the $\langle d,j \rangle$ entry of matrix Z is given by the dot product of the d^{th} row of matrix X and the j^{th} column of matrix W , summing over index i .

Non-Linear Layer: Given signal $z_{\langle d,j \rangle}$, the j^{th} hidden neuron ideally either fires or it doesn't. The picture is either a cat or not. One might model this by having the hidden neuron fire if this signal $z_{\langle d,j \rangle}$ passes a threshold. Typically, a constant $w_{\langle 0,j \rangle}$ is added into the sum so that the threshold is shifted to zero. To be more graceful, a typical activation function used is the sigmoid function $y_{\langle d,j \rangle} = \sigma(z_{\langle d,j \rangle})$ that has the hidden neuron output close to one when input sum $z_{\langle d,j \rangle}$ is very positive, output close to zero when it is very negative, and in between it changes gracefully. The problem with the sigmoid activation function is that its derivative and hence the gradient of steepest descent is close to zero for large activation values z , making learning slow. To solve this problem, the rectilinear activation function does not change the signal, i.e., $y = z$

when it is positive and zeros it, i.e., $y = 0$, when negative. Note for positive z , the slope here is always one, making learning faster. Recent work has shown that this activation function works surprisingly well for many applications.

Convolution and Recurrent Layer: Suppose the input is an image and the neural network is trained to find a cat. This problem is invariant over the location of the cat, namely, the network should answer yes or no matter where in the image the cat is located. This invariant can be built into the design of the neural network by having a set of weights be learned to find a feature in some small part of the image and to use those same weights translated all over the image. Similarly, if the input is a stream of speech, then the same set of weights can detect the word “cat” at the beginning and at the end of the stream. These changes decrease the number of weights that the neural network has, making learning easier, decreasing computation time, and decreasing overfitting.

The Singularity: If technology grows just slightly faster than exponentially, then at some point in time, the expected amount of technology will be infinite. What will life look like after that?

31 Key Concepts Summary: Greedy Algorithms and Dynamic Programming

31.1 Greedy Algorithms

When providing a greedy algorithm and its proof of correctness, the following are all the paragraphs that should be included, their headings, and what they should contain.

Carefully read the full description of the steps in [Section 22.1](#).

Specifications: This is likely part of the question and hence does not need to be written. However, before writing anything, consider: What are the objects in the instance? What decisions need to be made about each such object in order to produce a solution? When does such a sequence of decisions make a valid/invalid solution? What is the measure according to which the solution is optimal? When writing your answer, be sure to use the details of the question at hand.

The Greedy Choice: Each iteration, the algorithm grabs the object that, according to some simple criteria, seems to be the “best” (or “worst”) from among the unconsidered objects in the instance. **Describe these criteria. Is it a fixed or an adaptive decision, i.e., does it depend on what objects have been seen already?** The algorithm then makes an irrevocable decision about this object. **How is this decision made?**

Hint: Usually the decision made is simply to commit to the next object unless it creates conflicts with the objects that the algorithm has already been committed to.

Example: With MST, the algorithm chooses the edge with the smallest weight. It decides to commit to this edge if it does not create a cycle with the previously committed to edges.

A_t : Let A_t denote the choices made by the algorithm during the first t time steps. See [Figure 31.1](#).

Here A_4 denotes the decisions made by the greedy algorithm during the first four time steps. Based on its greedy criteria, it chose to look at objects 5, 7, 9, and 3 in the first 4 time steps and to irrevocably commit to all of these except for object 7. (Note that unlike in the text, these numbers index the objects, not the time step that they were chosen.) Irrevocably committing to object 3 means that it has “burnt the bridge” of having any optimal solutions in that, object 3 is rejected. The loop invariant assures us that the algorithm has not burned in this way all of the optimal solutions. Here S_4 , held

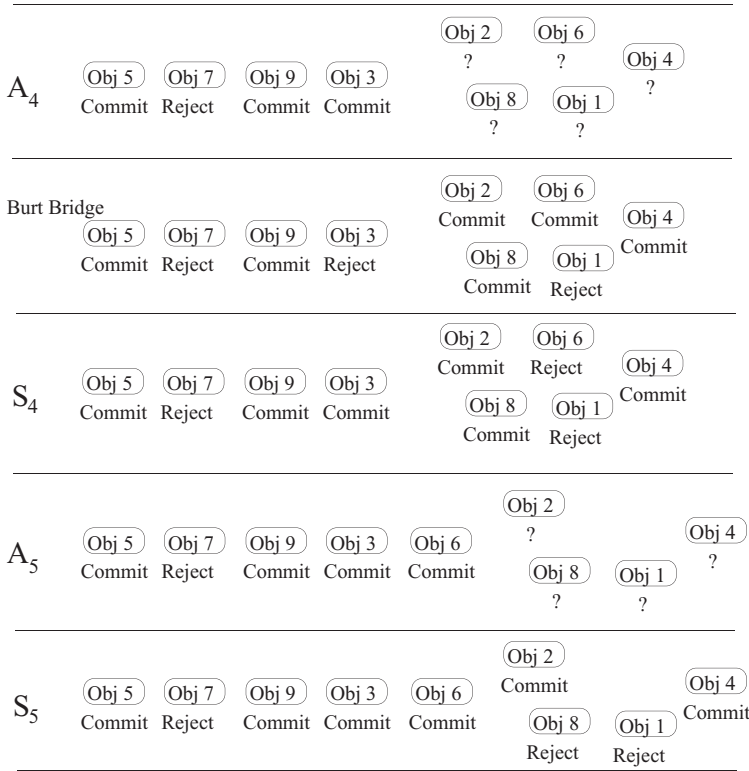


Fig. 31.1 Maintaining the loop invariant.

by the fairy godmother, is a valid optimal solution that extends the decisions A_4 that the algorithm has made so far. Being a full solution, it makes a decision about every object in the input instance, but being consistent with the algorithm it also commits to objects 4, 9, and 3 and rejects object 7. What other decisions S_4 makes is unknown to the algorithm and to the prover. In the 5th time step, the algorithm commits to object 6. The prover gives instructions to the fairy godmother to commit to object 6 as well, if she has not already done so. Doing this alone makes the solution she is holding either invalid or of less worth. Hence, the prover also instructions to the fairy godmother to change her commitment to object 8 to a reject. Let S_5 denote what she holds after the changes. The prover then proceeds to prove that S_5 is a valid solution, that it is an extension of A_5 that the algorithm has done so far and is optimal.

Example: With MST, A_t specifies for each of the t edge with the smallest weight whether it was committed to or rejected.

S_t Extends A_t : For S_t to be solution for our instance, it must make a decision for every object in our instance. We say that S_t is a solution that *extends* A_t , if for the objects that algorithm has already made a decision about, S_t makes the same decision, i.e. $\forall$ object $Obj \in \text{instance } I$, if $A_t(Obj) \neq \text{undefined}$, then $S_t(Obj) = A_t(Obj)$.

Example: With MST, S_t specifies a full spanning tree of the graph.

The Loop Invariant: We have not gone wrong. There is at least one valid optimal solution S_t that extends the choices A_t made so far by the algorithm.”

(If the problem is such that an instance may have no valid solutions, then the loop invariant needs to be modified to: “If our instance does have a valid solution, then there is at least one valid optimal solution S_t that extends the choices A_t made so far by the algorithm.”)

Initially ($\langle pre \rangle \rightarrow \langle LI \rangle$): Initially no choices have been made (i.e., $A_0 = \emptyset$), and hence all optimal solutions S_0 extend these choices.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI_t \rangle$):

Consider an arbitrary iteration.

Fly In from Mars: The algorithm has iterated $t-1$ times making decisions A_{t-1} about the first $t-1$ objects and is now at the top of the loop. **If useful, state or at least think about the types of decisions the algorithm makes.** All that we know is that the loop invariant is true and the exit condition is not.

S_{t-1} : The loop invariant states that there is at least one optimal solution that extends the choices made by the algorithm before this iteration. Let S_{t-1} denote one such solution. (Note that this is a full solution and as such specifies a decision about each object in the instance.) If you like, have a fairy god-mother hold it.

Taking a Step: Let Obj_t denote the next “best” object chosen by the algorithm during the t^{th} iteration. Denote $A_t = A_{t-1} + \text{decision made about } Obj_t$.

Example: With MST, there are two cases. First suppose that the algorithm commits to the next best edge $Obj_t = \langle u, v \rangle$ because it does not create a cycle with the previously committed to edges.

Instructions for Modifying S_{t-1} : Give the prover’s detailed instructions on how the fairy godmother should modify S_{t-1} . We will use S_t to denote what she constructs.

Warning: Be sure you give a *full* description of how S_t is constructed *before* you try proving anything about it. When marking, this is a first thing I look at.

Perhaps S_t Is Already Consistent: S_{t-1} is already consistent with the first $t-1$ decisions made by the algorithm. If S_{t-1} happens to be consistent with the decision A_t , the algorithm made about Obj_t , then we are done. Sometimes because of the decisions already made by the algorithm, the algorithm is forced to make this decision about Obj_t . Usually, this occurs when the algorithm’s instructions are to commit to taking Obj_t unless it can’t because Obj_t “conflicts” with the previous decisions made by the algorithm. In such a case, if the algorithm rejected Obj_t , then

we know it was forced to. Such forced decisions are good for the prover, because we know that because the fairy godmother has made the same first $t-1$ decisions, that she too is forced to make this same decision about Obj_t .

Making S_t Consistent: If the decision the fairy godmother has made about Obj_t is different from A_t , then the first modification the prover must tell the fairy godmother is to change this decision, namely, $S'_t = S_{t-1} + A_t$'s decision made about Obj_t . For example, if the algorithm committed to adding Obj_t and she did not, then get her to add it.

Check For and Fix Any Conflict: We know that S_{t-1} is a valid solution as it is. Changing the decision made about Obj_t most likely makes it no longer valid. Find and describe some object Obj'_t in S_{t-1} (but not in A_{t-1}) that “conflicts” with this changed decision about Obj_t and describe a way of changing the decision made by the fairy godmother about Obj'_t so that the solution S_t is made to be again valid. Define S_t to be the new solution $S_t = S_{t-1} + A_t$'s decision made about Obj_t + the prover's new decision about Obj'_t . Maybe the decision about more than one object needs to be changed.

The Fairy Godmother's Decisions Are not Ordered: Do not say “Add object Obj_t that the algorithm took next and delete the object Obj'_t that the fairy godmother took next.” Unlike the algorithm, the fairy godmother does not make decisions about the objects in any particular order. Hence, it is meaningless to say, “Let Obj'_t be the object that the fairy godmother considered at time t .” Because the algorithm makes a decision about an object each time step, we can define Obj_t to be the one it makes a decision about at time t . Yes, the fairy godmother has made a decision about the objects $Obj_1, Obj_2, \dots, Obj_{t-1}$ and in fact made the same decision about these that the algorithm did. Yes, she made a decision about Obj_t and in fact about every object in the input. This does not mean that there is an object she decided about “next.” Note, we might use the notation Obj'_t instead of Obj'_t , but this is because this is the object that **you** as the prover choose to change at time t .

Example: For MST, if S_{t-1} already contains the edge $Obj_t = \langle u, v \rangle$ added by the algorithm, then we are done. Otherwise, because S_{t-1} spans the graph without cycles, it must contain some other unique path from u to v . The algorithm can't have already committed to all the edges in this path, or adding $\langle u, v \rangle$ would create a cycle. The algorithm can't have already rejected an edge in this path, or else for consistency S_{t-1} would have rejected it too. Let $Obj'_t = \langle u', v' \rangle$ be some edge in this path from u to v in S_{t-1} for which no decision has been yet by the algorithm. Let $S_t = S_{t-1} + \langle u, v \rangle - \langle u', v' \rangle$.

Proving S_t Is a Valid Solution: By the loop invariant, S_{t-1} is a valid solution. (i.e., **repeat what it means to be valid**). The prover made sure that their modifications did not make it invalid. The changes may temporarily have made it invalid, but all of these problem were fixed. Hence, S_t is a valid solution.

Warning: Do not use the word “valid” without defining it. What does it mean for this problem for a solution to be valid? Careful. What needs to be proved to prove that *everything* about S_t is valid?

Example: With MST, we must prove that S_t is acyclic and spans the graph. Even if you can’t get the full proof, be sure to minimally state what it is that needs to be proved.

Adding $\langle u, v \rangle$ created one cycle with the unique path in S_{t-1} from u to v , but we broke this cycle by removing $Obj'_t = \langle u', v' \rangle$. Hence, S_t is acyclic as was S_{t-1} . Because S_{t-1} spans the graph, it has a path from node x to node y . The path from x to y in S_t may have to go the other way around the cycle that was changed. A picture would help. Hence, S_t is acyclic.

Proving S_t Extends A_t : By the loop invariant S_{t-1} extends A_{t-1} . The prover made sure that his modifications did not change any of these decisions and changed S_{t-1} ’s decision about Obj_t . Hence, S_t extends both with earlier decisions made by algorithm and this most recent decision, i.e., extends A_t .

Warning: Do not use the word “extends” without defining it. What decisions did the algorithm make before, that we know S_{t-1} extends, and how are you going to prove that S_t is still consistent with these? What decision did the algorithm just make and how are you going to prove that S_t is now consistent with it?

Example: For MST, we were careful that the only edge $Obj'_t = \langle u', v' \rangle$ removed from S_{t-1} had not previously been committed to by the algorithm and we were careful to add the edge $Obj_t = \langle u, v \rangle$ just committed to by the algorithm.

Proving S_t is an Optimal Solution: By the loop invariant, S_{t-1} is one of the optimal solutions for this instance. The prover made sure that his modifications did not make S_t worse than S_{t-1} . Hence, S_t is an optimal solution.

Warning: What does it mean for this problem for a solution to be optimal? i.e., what measure of success is minimized or maximums. What do you need to prove that S_t is at least as good as S_{t-1} ? No need to waste your efforts or insult the fairy godmother by mentioning that in some cases S_t is in fact better than S_{t-1} .

Example: For MST, $Obj'_t = \langle u', v' \rangle$ is defined to be some edge for which no decision has been made yet by the algorithm. The algorithm sorted the edges by weight and is currently deciding about edge $Obj_t = \langle u, v \rangle$. Hence, the weight of $\langle u, v \rangle$ is less than or equal to that of $\langle u', v' \rangle$ and hence the weight of S_t is less than or equal to that of S_{t-1} .

→ **$\langle LI_t \rangle$:** Because S_t witnesses the fact that there is at least one optimal solution that extends A_t , we know that the loop invariant has been maintained.

Second Case: With MST, now suppose the the algorithm rejects the next best edge $Obj_t = \langle u, v \rangle$ because it creates a cycle with the previously committed to edges. In

this case, the fairy godmother must have the same the previously committed to edges and hence also can't take $\langle u, v \rangle$. Let $S_t = S_{t-1}$.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): By the exit condition, the algorithm has made a decision about every object in the instance. Hence, the algorithm has a full solution. By the loop invariant, this extends an optimal valid solution. Hence, the algorithm must have an optimal valid solution.

31.2 Dynamic Programming

31.2.1 Quick Thoughts

- Given an instance, go through the game of life.
 What does Pooh Bear remember about his past so that he can continue?
 How do you index the states that he might be in?
 What actions might he take from where he is? (The bird tells him which to take.)
 Describe the nodes, the source node s , the sink node t , the edges, and the edge weights of the resulting graph.
 Give special thought to the edges into t .
 Explain why.
 Hint: Be sure that there is a 1-1 mapping between the valid solutions of the problem and the st -paths of your graph. (You do not need to discuss this.)
 Hint: Be sure to have as few edges as possible.
- In order to make the code more aesthetic, we turn everything around. Give the bird-friend algorithm.
 - What is the set of subinstances solved? What is the input/output of each?
 - What question do I ask the bird?
 - What answers do they give?
 - For each such answer k , what subinstance do you give your friend and what do they give you?
 - And how do you form your answer $optSol_{hi(i,k)}$?
- Which subinstances are base cases?
- Complete the following lines of the code:

```

    loop over the subinstances ... (give order)
      loop over the bird answers ...
        %  $optSol_{<i,k>}$  =
           $optCost_{<i,k>}$  =

```

31.2.2 Code from Bird & Friend Algorithm

- What is the running time of this algorithm?

When providing a dynamic programming algorithm and its proof of correctness, the following are all the paragraphs that should be included, their headings, and what they should contain.

1) Specifications: This is likely part of the question and hence does not need to be written. However, before writing anything, consider: For each instance, what is its set of valid solutions, and for each solution, what is its cost/value?

Algorithm Using Trusted Bird and Friend: I have my instance I . The little bird knows an optimal solution $optS$ to it.

2) Question for Bird: I ask the little bird a little question about it. Choose something like one of the following:

- Is the last object from the instance in the solution?
- What is the last object in the solution?
- The solution forms a tree. What is the object at the root?

3) Possible Answers from Bird: Define the list of possible answers that they may give. (Enumerate them with $k \in [K]$.) We want the number K of these answers to be small.

4) Trust the Bird: Assume that the little bird gives me the answer indexed by k . Trust them. What does this tell you about the solution $optS$ to your instance? More importantly, what does it tell you about the parts of the solution that they did not tell you?

5) Constructing Subinstances: The bird gave me some of the solution $optS$ I am looking for. I want my recursive friend to give me the rest of it. However, I can only ask them a smaller instance to my same computational problem. What instance $subI$ should I give my friend so that they will give me what I want? Again, we don't micromanage them, but trust them. They give me an optimal solution $optSubSol$ for their instance $subI$ and its cost.

6) Constructing a Solution for My Instance: I produce an optimal solution $optSol[k]$ for my instance I from the bird's answer k and the friend's solution $optSubSol$. If my little bird happens to be trustworthy, then this is an optimal solution. But even if it is not, this will be the best solution for my instance from among those consistent with the k^{th} bird answer.

7) Costs of Solution: Similarly, I compute the cost $optCost[k]$ of our solution $optSol[k]$.

Recursive Backtracking Algorithm

8) Best of the Best: I can trust the friend because they are a recursive version of myself. Not actually having a little bird, I try all their answers and take best of best.

9) Base Cases: An instance that is so small that there are no smaller instances that can be given to a friend is considered to be a base case. The base case instances and their solutions need to be considered.

Dynamic Programming Algorithm

10) The Set of Subinstances: I imagine running the above recursive backtracking algorithm on my instance I . Determine the complete set S of subinstances $subI$ ever

given to me, my friends, their friends. Note that this set S needs to: 1) contain my instance I ; 2) be closed under this “sub”-operator; and 3) all (or at least most) of these subinstances should be needed.

11) Construct a Table Indexed by Subinstances: I index these subinstances with i (and maybe j) so that $subI[i,j]$ denotes a particular subinstance. Which subinstance $subI[i,j]$ denotes needs to be carefully described. I build a table indexed by these subinstances so that $optS[i,j]$ stores an optimal solution, for instance, $subI[i,j]$, $optCost[i,j]$, the cost of this solution, and $birdAdvice[i,j]$, the advice given by the bird on this subinstance. (Actually we don’t store the solution because it is too big.)

12) The Order in Which to Fill the Table: The order in which the friends must solve their subinstances must be determined. It must be an order so that nobody has to wait, i.e., from smaller to larger instances.

13) Loop Over Subinstances: A key line in the dynamic programming algorithm is the iteration over these instances in this order. Let $subI[i,j]$ denote the one currently being worked on. Be clear what this instance is. The task now is to find an optimal solution for it and to store it and its cost in the table at $optS[i,j]$ and $optCost[i,j]$. This is done in the exact same way that it is above. The only difference is how the friends communicate.

14) Question for the Little Bird: I have my instance $subI[i,j]$ and the little bird knows an optimal solution $optS[i,j]$ to it. I ask them a little question about it. A key line in the dynamic programming algorithm is the iteration over these bird answers indexed by $k \in [K]$. Be clear what the current answer is.

15) Constructing Subinstances: I have my instance $subI[i,j]$ and the little bird has given me their k^{th} answer. Trusting the bird, I consider what their answer tells me about the parts of the solution that they did not tell me. I design a subinstance $subI[i',j']$ to give my friend so that their solution will give me the parts of the solution that the little bird did not give me.

16) Communication between Friends: The key difference between recursive backtracking and dynamic programming is how the friends communicate. When you as the recursive backtracking friend on instance I want help from a friend on instance $subI$, you recurse and wait until they compute and return a solution. In contrast, when you as the dynamic programming friend on instance $subI[i,j]$ want help from a friend on instance $subI[i',j']$, you assume that because their instance is smaller, they have already found an optimal solution for their instance and have stored it and its cost in the table at $optS[i',j']$ and $optCost[i',j']$. All you need to do is look it up. Similarly, when you have finally completed solving your instance, you will store the solution and its cost in the table at $optS[i,j]$ and $optCost[i,j]$.

17) Constructing a Solution for My Instance: I produce a solution $optSol_{((i,j),k)}$ for my instance $subI[i,j]$ from the bird’s answer k and the friend’s solution $optSubSol$. This will be the best solution for my instance $subI[i,j]$ from among those consistent with the k^{th} bird answer.

18) Costs of Solution: Similarly, I compute the cost $optCost_{\langle(i,j),k\rangle}$ of my solution $optSol_{\langle(i,j),k\rangle}$.

19) Best of the Best: Once I have for each bird's answer k the best solution $optSol_{\langle(i,j),k\rangle}$ for my instance $optS[i,j]$ from among those consistent with the k^{th} bird's answer, I take the best of the best. This will be an overall best solution. I store it and its cost in the table at $optS[i,j]$ and $optCost[i,j]$.

20) Solution Too Big: Actually, the solution $optS[i,j]$ is too big and hence it takes too much time and space storing and copying this from friend to friend. Hence, we comment out every line of code involving solutions. Instead, we store the cost of this solution in $optCost[i,j]$ and the advice given by the bird on this subinstance in $birdAdvice[i,j]$.

21) Base Cases: A recursive backtracking algorithm says, "If the instance I that I am personally given is so small that there are no smaller instances that can be given to a friend, then I must solve it myself and return its answer." **Do not** do this in a dynamic programming algorithm. Note that even if the end user never calls the algorithm on such a base case instance, a recursive program needs to handle these because it calls itself on these smaller and smaller instances, stopping at the base cases. In contrast, dynamic programming algorithms do not recurse. Hence, my instance is a base case only if the end user gives it. Despite this, a dynamic programming algorithm must always solve a collection of base cases. Recall that S is defined to be the complete set of subinstances $subI$ ever given to me, my friends, their friends The table is indexed by these subinstances and each needs to be solved. In fact, the base cases, being the smallest of these, are solved first. Start the algorithm by storing for each base case instance $subI[i,j] \in S$, its (commented out) optimal solution $optS[i,j]$, cost $optCost[i,j]$ of this solution, and the bird's advice $birdAdvice[i,j]$ into the table.

22) Code: Your code **must** have the following structure.

algorithm *DynamicProgrammingAlg* (I)

<pre-cond>: I is my instance.

<post-cond>: $optSol$ is an optimal solution for I and $optCost$ is its cost.

begin

 % Table: $subI[i,j]$ denotes the subinstance indexed by $\langle i,j \rangle$ (Describe).

$optSol[i,j]$ would store an optimal solution for it, but it is too big.

 Hence, we store only the bird's advice $birdAdvice[i,j]$ given for the subinstance and the cost $optCost[i,j]$ of an optimal solution.

$table[range_i, range_j] \text{ } optCost, birdAdvice$

 % Base Cases: Describe the base cases and their solutions.

 loop over base cases

 % $optSol[basecases]$ = its solution

$optCost[basecases]$ = its cost

$birdAdvice[basecases]$ = ?

 end loop

```

% General Cases: Loop over subinstances in the table.
for  $i \in [range_i]$ 
    for  $j \in [range_j]$ 
        % Solve instance  $subI[i,j]$  and fill in table entry  $\langle i,j \rangle$ .
        % Try each possible bird answer.
        for  $k \in [K]$ 
            % The bird and Friend Alg: see above
            %  $optSol_{\langle i,j \rangle, k}$  = Describe how to construct the solution to
            % our instance  $subI[i,j]$  from the bird's advice
            %  $k$  and the solution  $optSol[friend]$  to our
            % friend's instance  $subI[friend]$ . This will be
            % the best solution for our instance from
            % among those consistent with the  $k^{th}$  bird
            % answer.
             $optCost_{\langle i,j \rangle, k}$  = Describe how to construct the cost of this
            % solution from the bird's advice  $k$  and the cost
            %  $optCost[friend]$  of our friend's solution.

        end for
        % Having the best,  $optSol_{\langle i,j \rangle, k}$ , for each bird's answer  $k$ ,
        % we keep the best of these best.
         $k_{min}$  = "a  $k$  that minimizes  $optCost_{\langle i,j \rangle, k}$ "
        %  $optSol[i,j] = optSol_{\langle i,j \rangle, k_{min}}$ 
        %  $optCost[i,j] = optCost_{\langle i,j \rangle, k_{min}}$ 
         $birdAdvice[i,j] = k_{min}$ 
    end for
     $optSol = AlgWithAdvice(I, birdAdvice)$ 
    return  $\langle optSol, optCost[initial\ instance] \rangle$ 
end algorithm

```

23) Constructing the Solution: We do all of this work, but because we commented out the lines of code having to do with solutions, we do not in the end have the optimal solution for our initial instance I . However, all this work was not in vain. We now have the advice the little bird would give for every subinstance in S . We then can run the recursive backtracking algorithm to obtain the optimal solution for I . This computation is now very fast because we do not need to try all the bird's answers.

24) Running Time: Clearly state the number of subinstances in the table. Clearly state the number of bird answers per subinstance. The running time is the product of these.

31.2.3 The Game of Life Algorithm

Reduction of Any Dynamic Programming Algorithm to *LeveledGraph* (G, s, t):

Most dynamic programming algorithms can be *reduced* to an optimal s - t path in a leveled graph.

LeveledGraph (G, s, t) : We covered the problem *LeveledGraph* (G, s, t) . Its input is $I_{LG} = \langle G, s, t \rangle$ where G is a weighted leveled graph and s and t are nodes. A solution S_{LG} is a path from s to t through G . The cost of a solution $cost_{LG}(S_{LG})$ is the sum of the weights of the edges in the given path. If we climb the mountain, the oracle $Oracle_{LG}$ will give us an optimal solution $Oracle_{LG}(I_{LG}) = S_{LG}$, for which we can compute its cost. This optimal solution could be that of minimum or of maximum cost, depending on what is needed.

Algorithm for Problem $P(I_P)$: Consider some problem $P(I_P)$. Its input is I_P and its solution is S_P with cost $cost_P(S_P)$. Our goal is to construct an algorithm Alg_P for problem P . Clearly, it must take I_P as input. It maps this given instance I_P to instance I_{LG} using our program $I_{LG} = InstanceMap(I_P)$. Our algorithm then gives I_{LG} to $Oracle_{LG}(I_{LG})$ who gives us S_{LG} . Our algorithm then maps solution S_{LG} to solution S_P using our program $S_P = SolutionMap(S_{LG})$.

Correct: All you need to do to prove that your algorithm Alg_P works is to show that the solution map is a bijection $SolutionMap^{-1}(S_P) = S_{LG}$ and show that this bijection keeps the solutions ordered with respect to cost, namely, $cost_P(S_P) \geq cost_P(S'_P)$ if and only if $cost_{LG}(S_{LG}) \geq cost_{LG}(S'_{LG})$. From this, we can conclude that Alg_P returns the optimal solution if and only if the oracle does.

Recursion is like a reduction to the same problem except with smaller instances.

A dynamic programming algorithm with a nonlinear global cost can suffer from the fact that the solution bijection did not keeps the solutions ordered with respect to cost.

Graph G : Algorithm Alg_P takes its input I_P and maps it using $InstanceMap(I_P)$ to $I_{LG} = \langle G, s, t \rangle$. The oracle give us the optimal s - t path through this graph G . Let's try to understand this graph G in three different ways (Figure 26.1).

1. The nodes of G are the cells of the dynamic programming table. See the five examples below.
2. Take the tree of stack frames produced by a recursive backtracking algorithm and merge some of its nodes. This gives you the graph G .
3. Look at life as following a path between the possible states that one might be in, i.e., a DFA automaton.

Nodes of G Are Subinstances

Recursive Backtracking: Recursion given an original instance I_P recurses on smaller subinstances, which in turn recurses on smaller subinstances and so on. This is implemented by a tree of stack frames. In this tree, there is a node for every subinstance that is ever solved when starting at our original instance I_P . Recursive backtracking sometimes solves a given subinstance many, many times. When this happens, dynamic programming reduces the total computation time by solving each subinstance only once. Imagine what happens to the recursion tree when you merge a set of its nodes into one. It become a DAG, i.e., a directed acyclic graph, i.e., a leveled graph, i.e., an instance to *LeveledGraph* (G, s, t) . Here

node t , being our destination, will be our original instance I_P , i.e., the root of the tree of stack frames. We might have to add a new node s to this collapsed graph G to be our starting point and put an edge from it to every base case, i.e., to every node with no children. The oracle gives us the optimal s - t path that winds back up through this newly formed graph G .

Subinstances/Nodes: Instead of constructing Alg_P for problem P , suppose instead we formed a dynamic programming algorithm for P . In this case, we imagine running the recursive backtracking algorithm and collecting this set of subinstances. Then we “index” a table with these subinstances. This is done by specifying each subinstance $subI[i,j]$ with one or two integers $\langle i,j \rangle$ and then indexing the table by these.

Edges: When constructing this dynamic programming algorithm for P , focusing your attention on solving one of its subinstances $subI[i,j]$, imagine setting the corresponding node of this graph. You ask your little bird to tell you some information about the solution. The recursive backtracking stack frame $subI[i,j]$ iteratively tries all the possible answers $k \in [K]$ that they might tell you. For each k tried, the stack frame recurses. Each time it recurses, the tree of stack frames has an edge from $subI[i,j]$ to the subinstance $subI[i',j']$ that you ask your friend. Hence, the bird’s answer k effectively tells you which of the outgoing edges you should take. Following this edges gets you to your friend’s subinstance $subI[i',j']$. In this way, the bird step by step provides you with the oracle’s optimal t - s path that starts at the root t and follows the bird’s path down to the base case nodes and on to node s .

Connection to the *LeveledGraph* Algorithm: Recall that when solving *LeveledGraph* using dynamic programming, there is a subinstance for every node $subI[i,j]$ asking for the best path from s to this node $subI[i,j]$. The bird then tells you the last edge to take in this path, i.e., the edge from some $k = subI[i',j']$ to our last node $subI[i,j]$.

***G* as the Graph of Life**

Loop Invariants: Remember what you have learned about loop invariants. They are NOT actions but static pictures of what is true at this instance in time. Then the code/action within the loop takes you from one such assertion/invariant to another.

DFA Automata: Each state in an automata (DFA) also specifies one such point in time. Label this state with every thing the automata knows to be true at this flash in time. If the number of states grows with the size of the input, then it is not a “finite” automata and if you have a little bird then it might not be a “deterministic” one, but the idea of a state will be the same. The edges between these states in this automata represent actions that take it from one such state to another.

Path of Life: What is life but a path through this automata graph, i.e., a sequence of snapshot states and/or the sequence of actions/happening that move you from one state to another? Some but not all of these states will happen to you. What “cause/decides” which edge to follow next is the whole debate of free will vs. determinism vs. divine intervention. But in an optimal world, for each state that you might be in, the little bird’s advice tells you the last action k you took to get to this state.

Read the Input I_P and Produce the Solution S_P : As you follow a path from start state s to the destination state t , your overall task will be to read the input I_P and produce the solution S_P . As such, each edge $\langle state[i, j], state[i', j'] \rangle$ of our graph G needs to be associated/labeled with following information.

States: Clearly, the edge is identified with the states $\langle state[i, j]$ and $state[i', j'] \rangle$ that this edge goes between.

Action: The action needed to take place for this change of state to occur.

Part of Input I_P : The part of the input that is being read during this action.

Part of Solution S_P : There needs to be a bijection $S_P = \text{SolutionMap}(S_{LG})$ between the solution to these two problems. Here, S_{LG} is a path through the leveled graph G . Each such path needs to be associated with a solution S_P (whatever that looks like) to the problem P . This solution S_P needs to be read off the edges of this path S_{LG} . Hence, each edge must be labeled with a small part of this solution S_P .

Incremental Cost: Each edge needs to be labeled with a weight. The cost/-value of a solution/path S_{LG} will be the sum of the weights along this path. So the weight of the edge will be the incremental cost/value of the subsolution for $state[i, j]$ and that for $state[i', j']$.

Forgetting: The running time of your algorithm will depend on the number of edges (and nodes) of the graph G . Each node/state is labeled with everything you know when in that state. Knowing a “lot” means that there are lots of different possibilities of what you may have known. Hence, there needs to be lots of states. So the goal is to forget any information that you don’t need to continue on. Suppose you are a front-line worker and know every detail of everything that has ever happened. You need to tell your boss just enough information so that they can make an informed decision about what to do next. Each state $state[i, j]$ is labeled with this information. Specifically, $\langle i, j \rangle$ informs the boss of what they need to know.

Cutting Task into Two Independent Tasks

Examples

Knapsack: I am walking down an aisle of objects $\langle obj_1, \dots, obj_n \rangle$ and my life goal is put an optimal subset of the objects into my knapsack. When I

am in state $state[i, j]$, I have already walked past the objects $\langle obj_1, \dots, obj_i \rangle$ and have put a (hopefully optimal) subset of them into my knapsack with the total volume filled so far being at most j . Which subset I have depends on the path of actions I took to get here. As I walk past object obj_{i+1} , I take one of two following possible actions. With action $k = \text{yes}$, I put obj_{i+1} into my knapsack, increasing the volume j filled so far by the volume of this object. The weight of this action/edge is the value of this object. With action $k = \text{no}$, I do not put it in. The weight of this action/edge is zero. Our goal is to get from state $s = state[0, 0]$ to $t = state[n, V]$ with the maximum weighted path.

Longest Common Subsequence: My life goal is to build a longest common subsequence of the two strings $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_m \rangle$. When I am in state $state[i, j]$, I have already moved my first string's cursor past $\langle x_1, \dots, x_i \rangle$, my second past $\langle y_1, \dots, y_j \rangle$, and I hold a (hopefully longest) common subsequence LCS of what I have moved past. Which LCS I hold depends on the path of actions I took to get here. When $x_{i+1} \neq y_{j+1}$, I cannot change my LCS, but my action either moves my first cursor past x_{i+1} or my second past y_{j+1} . These actions are denoted $k = \downarrow$ and $\rightarrow$ because this can be viewed as the direction I am traveling through this matrix of states. The weight of this action/edge is zero. When $x_{i+1} = y_{j+1}$, I extend my LCS with this new character and move both cursors past it. This action is denoted $k = \searrow$. The weight of this action/edge is one. Our goal is to get from state $s = state[0, 0]$ to $t = state[n, m]$ with the maximum weighted path.

Printing Neatly: My life goal is to print a sequence of words neatly. When I am in state $state[i]$, I have already printed the first i words completing some number of lines. How I printed them depends on the path of actions I took to get here. My next action is to decide some number of words k and to print them on the next line. The weight of this action/edge is the cube of the number of blanks on the end of this new line. Our goal is to get from state $s = state[0]$ to $t = state[n]$ with the minimum weighted path.

Same: Note that subinstance $subI[i, j]$ and state $state[i, j]$ are effectively the same and that these define the SAME graph G . It is this graph that you are giving to the oracle.

Edge Length: Consider whether the average length of the edges in G are short or long. Short edges mean that your path from birth to death will traverse lots and lots of short actions. However, we don't care how many edges are in the optimal path. Running time is not the work of traversing this path, but the work of searching for it. The search time depends on the total number of possible edges that we need to search among, whether or not taken.

Long edges are scarier because you don't really know where in the future they are leaping to. The further you go, the more places you can get to, and hence the more edges there are to search and hence the longer this searching takes.

The little bird tells you either the next edge or the last edge that you should take in an optimal path. Because the algorithm must try all the possible bird answers, you want the number to be small. All the edges go from the start node s all the way to the destination t ; then you would need a separate edge for each of the exponential number of solutions to I_p . This would be too many.

There should be an edge in your graph G for every “atomic” action that takes you from one state to another. Non-atomic actions are not needed because they can be broken into a sequence of atomic actions.

Running Time: The running time of our leveled graph dynamic programming oracle is the number of subinstances times the number of edges $\deg(u)$ coming into node u . To be more accurate, the time is $\sum_u \deg(u)$. This equals the number of edges in the graph. Hence, like depth first and breadth first searches, the running time is linear in the number of edges. For this reason, it does not matter how many nodes are in the graph I_{LG} that you give the oracle $Oracle_{LG}(I_{LG})$ who is solving $LeveledGraph_{LG}(G, s, t)$ for you. Give then as many nodes as you like. However, truly try to give them as few edges as possible. **Carefully** consider every edge you are adding to the graph G and make sure that it truly makes a difference to the result.

Can’t Reduce: Parsing is a dynamic programming algorithm that can’t easily be reduced in this way because it recurses on the left and the right sub-trees.

32 Additional Exercises: Part III

32.1 Graph Algorithms

III.1. **Short Answers:** Three or four sentences each.

- (a) What are the generic loop invariants used for searching a graph starting from some node s ?
- (b) What are the extra loop invariants used for Breadth First Search. See Exercise 20.7.1 for tracing it on [Figure 32.1](#).
- (c) What are the extra loop invariants used for Depth First Search?
- (d) What are the pre- and postconditions for the Recursive Depth First Search algorithm?
- (e) How is the extra loop invariant for Depth First Search used to look for cycles? If the graph has a cycle, is the cycle guaranteed to be found in this way?
- (f) What are the extra loop invariants used for Dijkstra's shortest-weighted paths algorithm?
- (g) On [Figure 32.1](#), Dijkstra's algorithm, do the following:

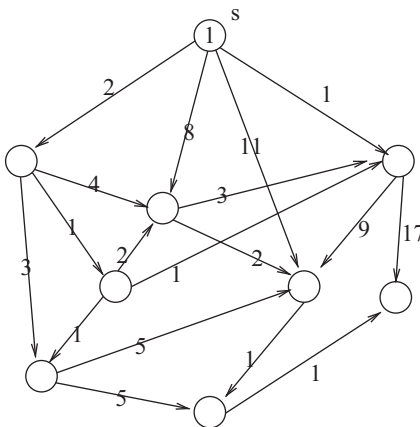


Fig. 32.1 Dijkstra's algorithm.

- i. Start at node s and when there is a choice follow edges from left to right. Number the nodes $1, 2, 3, \dots$, in the order that they are handled, starting with Node $s = 1$.
 - ii. Darken the edges of the Tree specified by the predecessor array π .
 - iii. What is the data structure used to store nodes that are found but not yet handled?
 - iv. For each node, give the list of every d value that it has at various points during the computation.
- (h) The network flow algorithm keeps finding a better flow by augmenting the current flow with a path. It stops when there is no path from s to t in the augmenting graph. In a few sentences explain how we proved that the resulting flow is optimal.
- (i) Sketch the algorithm for bipartite matching.
- (j) What is linear programming? For example, express Network Flows as a Linear Program.
- (k) At a very high, level what is the algorithm to solve linear programming/network flow/standing on Mars?

Graph Search Problems:

- III.2. **Edge in Cycle:** Given a directed graph G and a directed edge $\langle u, v \rangle$, give an algorithm to find (if possible) a cycle in G that contains $\langle u, v \rangle$. Explain.
- III.3. **Connected Components:** Sketch an algorithm that partitions the nodes of an undirected graph G into its connected components.
- III.4. **Graph Searching**
- (a) Briefly describe an algorithm that is given a directed graph and a destination node t and outputs all nodes v for which there is a path from node v to t .
 - (b) Let G be a directed graph where each node of G also has a value: e.g., node 1 might have value 100, node 2 might have value 50, etc., given an $O(|E| + |V|\log|V|)$ time algorithm that computes, for every node, the maximum value reachable from that node. For instance, if G is strongly connected (every nodes is reachable from every node), then for every node this will be the maximum value in the entire graph. Hint: One worthwhile preprocessing step is to sort nodes by value.
- III.5. **Topological Sort:** Given a DAG (directed acyclic graph), what is a total order and what are its properties. As an example, give a total order of the following DAG (Figure 32.2). The steps of the algorithm are not needed.
- III.6. **Proof Tree:** Suppose I have a set S of strange objects and my goal is to prove that all of these objects are fun, i.e., $\forall x \in S, \text{Fun}(x)$. We have considered proofs that look like rooted trees (see Tree of Options page 301). Each internal node is labeled with a question about x with the

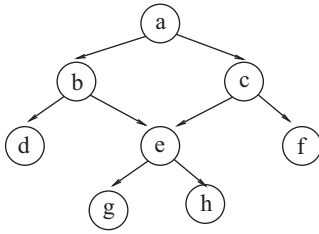


Fig. 32.2

answers labeling the outgoing edges. The leaf nodes are labeled with a proof that some x s are fun.

Give a one-sentence proof that this tree proof proves $\forall x \in S, \text{Fun}(x)$.

Network Flow Problems:

- III.7. **Network Flow:** Consider some large network. As you know, the goal is to find the maximum flow from s to t .
- Suppose edge $\langle a, b \rangle$ has capacity 8 and current flow 3. Suppose edge $\langle b, a \rangle$ has capacity 7. Give the augmenting graph for this edge.
 - If 4 more flow is pushed along edge $\langle b, a \rangle$ then what is the resulting flow along these edges? What can we say about how the total flow from s to t changes?
 - In one sentence, prove that the maximum flow is not worse than the flow returned by the algorithm.
 - In one sentence, prove that the maximum flow is not better than the flow returned by the algorithm.
- III.8. **Flow:** Recall that the flow of a network is defined as the flow out of the source. Suppose you have a network where flow conservation does not apply (i.e., for nodes other than the source and sink, the flow in may not equal the flow out). Would the maxflow-mincut theorem still apply? Why or why not?
- III.9. Once your hill climbing algorithm finds an optimal solution for the given Network Flow instance, describe how you would take “steps” that move you around on top of this plateau from one optimal solution to another. As with a hill climbing step, this step will look at the augmentation graph of the current flow. Clearly, however, you will not be able to find an st -path in this augmentation graph. What do you look for instead? (If you are really bold, attempt to prove that every optimal solution can be reached with such steps.)
- III.10. **Short Answers**
- What is the purpose of considering the Dual of a Linear Program?
 - What is the Steepest Accent version of the Network Flows algorithm?
 - What is the measure of progress for this Steepest Accent Network Flows algorithm and how much does it change each iteration.
- III.11. See Questions III.54 and III.55 in which a Node Capacities and a Greenhouses problems are reduced to Network Flow.

32.2 Greedy Algorithms

III.12. **Short Answers:** Discuss (and correct) the following:

- (a) **Decisions:** Greedy algorithms need to make a decision about each object. Which object does it make this decision about first and how is that decision made? When doing so, does it consider the rest of the input?
For example, the greedy algorithm that tries to find the shortest path from s to t always takes the cheapest next edge.
In what way is this choice optimal? Does this guarantee that the algorithm produces the optimal solution? How does the prover prove that the algorithm did not go wrong with this decision? Is this initial decision ever later changed?
- (b) **What is the loop invariant?**
- (c) A_t : Discuss the meaning of A_t being optimal.
- (d) S_t : We know S_{t-1} is changed into S_t in order to make it extend A_t . Who does it and why? How do you prove that S_t is optimal?
- (e) **How much time does it take the algorithm to change S_{t-1} into S_t ?**
- (f) **Running Time:** What is the running time of a greedy algorithm? Which action dominates this time?

III.13. **Integer-Knapsack Problem**

Instances: An instance consists of $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$. Here, V is the total volume of the knapsack. There are n objects in a store. The volume of the i^{th} object is v_i , and its price is p_i .

Solutions: A solution is a subset $S \subseteq [1..n]$ of the objects that fit into the knapsack, i.e., $\sum_{i \in S} v_i \leq V$.

Measure of Success: The cost (or success) of a solution S is the total value of what is put in the knapsack, i.e., $\sum_{i \in S} p_i$.

Goal: Given a set of objects and the size of the knapsack, the goal is fill the knapsack with the greatest possible total price.

Failed Greedy Algorithms: Three obvious greedy algorithms take first the most valuable object, $\max_i p_i$, the smallest object, $\min_i v_i$, or the object with the highest value per volume, $\max_i \frac{p_i}{v_i}$. However, they do not work. For each of these greedy algorithms, give an instance on which it does not work.

III.14. **Crossing the River:** A man is standing on the bank of a river that he must cross by jumping on stepping stones that are spread in a line across the river. Though he can't jump far, he wants to jump on as few stones as possible. An *instance* to the problem specifies $\langle d_0, d_1, d_2, \dots, d_n \rangle$, where $d_0 = 0$ is the man's initial location, d_i for $i \in [1, \dots, n-1]$ is a real number giving the distance in meters that the i^{th} stone is from him, and d_n is the distance to the opposite shore. Assume that the stones are in order, i.e., $0 = d_0 \leq d_1 \leq d_2 \leq \dots \leq d_n$. Also assume that the stones are not more than one meter apart, i.e., $\forall i \ d_{i+1} - d_i \leq 1$.

A *solution* is a sequence $\langle i_0, i_1, i_2, \dots, i_\ell \rangle$ of the indexes of the stones, indicating that at time t the man jumps onto the i_t^{th} stone. Such a solution is *valid* if at time $t = 0$ the man is in fact on the close shore, i.e., $i_0 = 0$, he finishes on the far shore, i.e., $i_\ell = n$, and at each time step t' the distance he jumps is at most a distance of one meter, i.e., $d_{i_{t'}} - d_{i_{t'-1}} \leq 1$. The *cost* of a solution is the number ℓ of stones jumped onto.

Note that the assumption on the input ensures that a valid solution always exists. Specify a greedy algorithm for finding an optimal solution for this problem. Imagine that you are crossing a river on stones. How would you choose which stone to jump on next?

As done in the steps, prove that your algorithm always returns an optimal solution. Warning: To ensure that a solution is valid, you really must check the distance jumped at *each* time step t' .

III.15. Cards Max Lexicographically

The input is a set of cards $I = \{\langle d_i, c_i \rangle \mid i \in [1..n]\}$ and an integer C .

Here d_i is a “digit” and c_i is a cost.

A solution $\langle i_1, i_2, i_3, \dots, i_m \rangle$ is a subset of the cards in an order.

The solution is valid if the sum of the costs c_i of the cards in it is at most C , i.e., $\sum_{j \in [1..m]} c_{i_j} \leq C$.

Consider the string formed by concatenating the digits of the solution together with zeros added on the end so that each solution has n characters, namely, $D = \langle d_{i_1} d_{i_2} d_{i_3}, \dots, d_{i_m} 00, \dots \rangle$.

The value of the solution that we are to maximize is the integer value and/or the lexicographical order of D . For example, 122 is lexicographically before 21 but the integer 122 is bigger than 21. We added zeros to make it clearer. Solution 12200 is worse than 21000.

For example, $I = \{\langle 2, 5 \rangle, \langle 3, 4 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle, \langle 8, 8 \rangle, \langle 9, 9 \rangle, \langle 9, 10 \rangle\}$ and $C = 14$.

The following are the D and total cost C' for various valid solutions sorted from worst to best:

$D = 000000000$	$C' = 0$	$S = \langle \rangle$
$D = 233300000$	$C' = 5 + 3 + 3 + 3 = 14$	$S = \langle \langle 2, 5 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle \rangle$
$D = 333300000$	$C' = 3 + 3 + 3 + 3 = 12$	$S = \langle \langle 3, 3 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle \rangle$
$D = 390000000$	$C' = 3 + 9 = 12$	$S = \langle \langle 3, 3 \rangle, \langle 9, 9 \rangle \rangle$
$D = 833000000$	$C' = 8 + 3 + 3 = 14$	$S = \langle \langle 8, 8 \rangle, \langle 3, 3 \rangle, \langle 3, 3 \rangle \rangle$
$D = 930000000$	$C' = 9 + 3 = 12$	$S = \langle \langle 9, 9 \rangle, \langle 3, 3 \rangle \rangle$
$D = 930000000$	$C' = 9 + 4 = 13$	$S = \langle \langle 9, 9 \rangle, \langle 3, 4 \rangle \rangle$
$D = 930000000$	$C' = 10 + 3 = 13$	$S = \langle \langle 9, 10 \rangle, \langle 3, 3 \rangle \rangle$
$D = 930000000$	$C' = 10 + 4 = 14$	$S = \langle \langle 9, 10 \rangle, \langle 3, 4 \rangle \rangle$

- Give a **non-adaptive greedy** algorithm for this problem.
Specifically, what are the greedy criteria and the decision-making process?
- Define A_t .
For your algorithm **on the above input**, give $A_0, A_1, A_2, A_3, \dots$
Explain where each of these comes from and why.

- (c) Define S_t .

For your algorithm **on the above input**, give $S_0, S_1, S_2, S_3, \dots$. Maximize the changes.

Explain where each of these comes from and why.

- (d) Now consider a generic input. Discuss A_{t-1}, S_{t-1} , and A_t . Explain how S_t is obtained.

- (e) Prove that your S_t has the required properties.

III.16. **Shortest Wait First:** A server needs to decide in which order to serve their n customers.

- The instance consists of $\langle w_1, w_2, \dots, w_n \rangle$, where w_i is the number of minutes required to serve customer i .
- A solution $S = \langle w_{\pi_1}, w_{\pi_2}, w_{\pi_3}, \dots, w_{\pi_n} \rangle$ specifies the order in which to serve these customers.
- It is valid if each customer appears exactly once.
- The cost of a solution is the total waiting time $T = \sum_{j=1}^n (n-j) \cdot w_{\pi_j}$.

Note the cost of putting the π_j^{th} customer in the j^{th} place in the order is $(n-j) \cdot w_{\pi_j}$ because $n-j$ other customers must wait w_{π_j} for him to complete.

- Give a greedy algorithm for determining an optimal order in which to process the customers in order to minimize this total wait time.
- Suppose $S_{t-1} = \langle w_1, \dots, w_{t-1}, w_{\pi_t}, w_{\pi_{t+1}}, \dots, w_{\pi_{s-1}}, w_t, w_{\pi_{s+1}}, \dots, w_{\pi_n} \rangle$. Give the instructions for modifying S_{t-1} .
- Prove S_t is a valid solution.
- Prove S_t is an optimal solution.

Give a greedy algorithm for determining an optimal order in which to process the customers in order to minimize this total wait time.

III.17. **Minimal Spanning Tree**

- Trace out a minimal spanning tree on [Figure 32.3](#). Darken the selected edges.

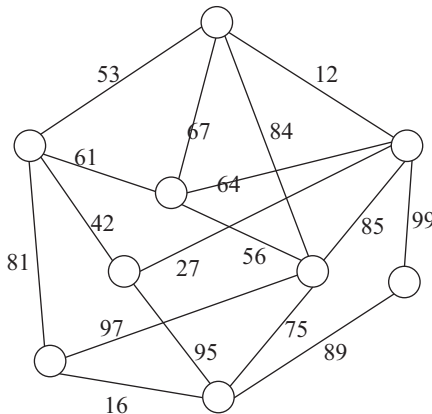


Fig. 32.3

- (b) Can DFS be used to find a minimum spanning tree (MST) of a given connected graph, if all edges have equal weight?
 - (c) In general, what action does a greedy algorithm take each iteration?
 - (d) What is the generic loop invariant for a greedy algorithm?
 - (e) Briefly explain how we prove that this loop invariant is maintained.
- III.18. (MST) Our goal is to find a shortest path from node s to node t in an undirected graph. Our algorithm of choice is to find a minimal spanning tree and then to return the path from s to t in this tree. Give a graph with n nodes for which the ratio between the length of the path returned and the length of the optimal path is as bad as possible.
- III.19. **Bottleneck Spanning Trees:** A bottleneck spanning tree of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G .
- (a) Prove that a minimum spanning tree is a bottleneck spanning tree.
 - (b) Your company needs to lease access to a communications network linking a large number of Canadian cities. The network can be modeled as a connected, undirected graph with vertices representing cities and edges representing direct communication links between cities. Each edge provides a maximum data rate, or bandwidth. You need to be able to move data from any one of the cities to any other, but you do not necessarily need every link. Since you will be tarified according to the number of links you use, you want to minimize the number of links you lease. In other words, you seek a spanning tree of the graph.
 Since your company provides broadband services, you wish to maximize bandwidth. On any path connecting two cities in the network, the bandwidth of the path is limited by the minimum bandwidth edge on the path. Your goal is to determine the spanning tree that connects each pair of cities by a maximum bandwidth path. Your tree will have the property that adding any additional edges from the graph will not produce a path with higher bandwidth.
 Design an algorithm to solve this problem and prove that it is correct.
- III.20. **Unique Solution/Profiles:** Given an undirected graph with weights on the edges, a solution to the MST problem is a subset of the edges that forms a spanning tree. Define the *profile* of a solution to be a multi-set consisting of the weights of the edges in the solution sorted so that we no longer know which edges they came from. For example, if the solution consists of three edges with weights 2, 7, and 2, then the profile would be $[2, 2, 7]$. Prove that every input instance has a unique optimal profile, i.e., every optimal solution has the same profile. Go on to prove that if the instance graph has a different weight on each edge, that the graph has a unique MST. Hint: Watch the optimal solution held by the fairy godmother.

- III.21. The fractional-knapsack problem is the same as the integer-knapsack problem except that there is a given amount of each object and any fractional amount of each it can be put into the knapsack.

Instances: An instance consists of $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$. Here, V is the total volume of the knapsack. There are n objects in a store. The volume of the i^{th} object is v_i , and its price is p_i .

Solutions: A solution specifies for each object $i \in [n]$, the possibly fractional amount $a_i \in [0, 1]$ of this object that is taken. A solution is valid if these amounts of these objects fit into the knapsack, i.e., $\sum_i a_i v_i \leq V$.

Measure of Success: The cost (or success) of a solution is the total value of what is put in the knapsack, i.e., $\sum_i a_i p_i$.

Goal: Given a set of objects and the size of the knapsack, the goal is to fill the knapsack with the greatest possible total price.

Develop a quick greedy algorithm for this problem and prove that it works. Give all the steps of the meta-algorithm including the greedy criteria for selecting the next object, the loop invariant, maintaining the loop invariant, . . .

- III.22. **Greedy:** Job/event scheduling problem when minimizing the number of rooms scheduled:

Here we have an infinite number of rooms, so instead of maximizing the number of events being scheduled, we are minimizing the number of rooms used.

When scheduling the maximum number of events into r rooms, the input is a set of jobs/events $\langle r, \langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \dots, \langle s_n, f_n \rangle \rangle$, where $0 \leq s_i \leq f_i$ are the starting and finishing times for the i^{th} event. A solution is a schedule $S = \langle S_1, \dots, S_r \rangle$ for each of the rooms. Each of these consists of a subset $S_j \subseteq [1 \dots n]$ of the events that don't conflict by overlapping in time. When maximizing the number of events that are scheduled in r rooms, the greedy criteria consider the events ordered by their finishing times. When considering the next event, if it can be scheduled in more than one room, it is scheduled in that with the latest last-scheduled finishing time. For example, suppose the last event scheduled in rooms 1, 2, and 3 finishes at times 10, 15, and 18, and the next event starts at time 17. Then the next event could be scheduled into either room 1 or 2 but not in 3. This algorithm would schedule it in room 2. Note this is the room that minimizes the gap between the finishing time of the previously scheduled job for the room and the starting time of the new job. Here a gap of size $17 - 15 = 2$ is better than one of size $17 - 10 = 7$ or of size $17 - 18 = -1$. On the other hand, if the next event cannot be scheduled in any of the r rooms, then it is not scheduled.

The rest of this question considers scheduling events into an infinite number of rooms, where the goal is to minimize the number of rooms used.

- (a) Because we want to minimize the number of rooms containing any events, we do not start scheduling into an empty room unless absolutely necessary. For example, suppose that the next event can be scheduled in room 1 and that room 2 is empty. Explain how you know that the above unchanged algorithm will automatically schedule this event into room 1 and not into room 2.
- (b) Suppose S_{t-1} looks like [Figure 32.4](#). Using the definitions of S_{t-1} , an input instance, and a solution, determine as much of the input instance as you can. Then run your changed algorithm on this input instance as long as S_{t-1} is reasonable. Given this, which are the events in A_{t-1} ? Why? Which event is Obj_t (also denoted i) and which room j is it put in? What is your new A_t ?

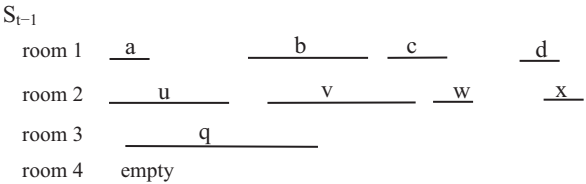


Fig. 32.4

- (c) For the example S_{t-1} given in the last question, draw your new S_t .
- III.23. **Horn Formulas:** Recall our proof that a given greedy algorithm always produces an optimal solution. The prover gives instructions to the fairy godmother on how to modify her solution. In contrast, we say that the given step of greedy algorithm is *forced* if the prover does not need to modify her solution because the prover knows that she is forced to make the same decision as the algorithm. For example, if the algorithm rejects the quarter because the prover's sum with it would be too much, then the fairy godmother must also reject this coin. Some greedy algorithms consist only of forced steps.

In order to display human-level intelligence, a computer must be able to perform at least some logical reasoning. Horn formulas are a particular framework for expressing logical facts and deriving conclusions. The most primitive object in a Horn formula is a Boolean variable, taking value either *true* or *false*. For instance, variables x , y , and z might denote the following possibilities.

- $x \equiv$ the murder took place in the kitchen.
- $y \equiv$ the butler is innocent.
- $z \equiv$ the colonel was asleep at 8 p.m.

A literal is either a variable x or its negation $\bar{x}$ ("NOT x "). In Horn formulas, knowledge about variables is represented by two kinds of clauses:

- (a) Implications, whose left-hand side is an AND of any number of positive literals and whose right-hand side is a single positive literal.
- For example: $(z \wedge w) \Rightarrow u$.
This might mean “if the colonel was asleep at 8 p.m. and the murder took place at 8 p.m. then the colonel is innocent.”
 - These express statements of the form “if all the variables on the left happen to be set to be *true*, then the variable on the right must also be set to *true*.”
 - Everyone knows that the statement $A \Rightarrow B$ is true when A and B are both *true*. The tricky part is that this statement is also true when A is *false* and B is anything you like. Hence, a clause like $(z \wedge w) \Rightarrow u$ is also satisfied by setting one of the variables $\{z, w\}$ on the left to be false.
 - *Forced Implication*: $() \Rightarrow x$. Such an implication clause has its left-hand side as empty. Such a left-hand side is by default *true*. Hence, needing to satisfy such a clause automatically forces x to be *true*: “the murder definitely occurred in the kitchen.”
- (b) Pure negative clauses, consisting of an OR of any number of negative literals.
- For example: $(\bar{u} \vee \bar{v} \vee \bar{y})$.
This might mean “they can’t all be innocent.”
 - These express statements of the form “At least one of the variables in this set is *false*.” (It does not matter which or how many.)

The two kinds of clauses pull us in different directions. The implications tell us to set some of the variables to *true*, while the negative clauses encourage us to make them *false*.

An *instance* to our problem consists of a set of clauses of these two types. A *solution* consists of a setting of each of the variables to either *true* or *false*. Such a solution is *valid* if it satisfies all the clauses. This is called a *satisfying assignment*. Occam’s razor recommends selecting a hypothesis that makes the fewest new assumptions. Toward this goal, we state that the *cost* of solution is the number of variables that are set to be true. Given a set of clauses, our goal is to find a satisfying assignment with as few *true* variables as possible.

- (a) Easy cases:
- i. Suppose that the given set of clauses has no *Forced Implication* clauses, i.e., it may have clauses of the form $(z \wedge w) \Rightarrow u$ and/or of the form $(\bar{u} \vee \bar{v} \vee \bar{y})$, but it happens to have no clauses of the form $() \Rightarrow x$. Argue that the solution that makes every variable *false* is valid and optimal.
 - ii. Suppose that the given set of clauses has a *Forced Implication* clause, $() \Rightarrow x$. Argue that every valid optimal solution sets this variable x to be *true*.

Hint: How does this help a greedy algorithm?

(b) Examples:

i. Suppose the instance consists of the following set of clauses:

$$(x \wedge z) \Rightarrow a, \quad x \Rightarrow y, \quad () \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad \text{and} \quad (\bar{w} \vee \bar{x} \vee \bar{y} \vee \bar{z}).$$

Find an optimal valid solution. Argue why it is valid. Argue why it is optimal.

ii. Suppose the instance consists of the following set of clauses:

$$(x \wedge z) \Rightarrow a, \quad x \Rightarrow y, \quad () \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\bar{w} \vee \bar{x} \vee \bar{y}), \quad \text{and} \quad (\bar{z}).$$

Find an optimal valid solution. Argue why it is valid. Argue why it is optimal.

(c) Suppose the greedy algorithm had set the *true/false* values of some but not all of the variables. What would the loop invariant be?

Hint: Note that it is completely possible that the given set of clauses has no satisfying assignment. For this reason, begin your loop invariant with “If our instance does have a valid solution, then . . .”

(d) How does the prover establish the loop invariant, i.e., $\langle pre \rangle \rightarrow \langle LI \rangle$?

Hint: Consider both the case in which the instance has a valid solution and the case in which it does not.

(e) Give all the steps for maintaining the loop invariant, i.e., $\langle LI_{t-1} \rangle \& \text{ not } \langle exit \rangle \& \text{code}_{loop} \rightarrow \langle LI_t \rangle$.

Consider an arbitrary iteration.

(f) *Party*: Alice wants to throw a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they don’t know. Give an efficient algorithm that takes as input the list of n people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of n .

III.24. **Greedy (Independent sets in trees)**: (Questions III.24, III.25, and III.38 are related.) The input consists of a rooted tree T . (See Figure 29.6.) A solution consists of a subset S of the nodes of tree T . (See black nodes in Figure 32.5.) Such a solution is valid if no two of the nodes in S have an edge between them (i.e., one is the parent of the other). Said another way, S is valid if each edge includes at most one of its two nodes in the solution. The cost/value of S is the

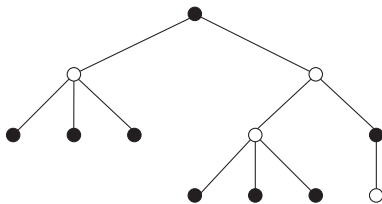


Fig. 32.5

number of nodes $|S|$ included. Our goal, given a tree T , is to find a valid solution whose value is maximized. We will consider a number of different algorithms for it.

- (a) An easy algorithm is to layer the nodes of the graph by their distance from the root. Color the nodes on the evenly indexed levels red and on the odd ones blue. Take the bigger of these two sets as the solution to independent set.
 - Does this algorithm always return a valid solution? Prove.
 - Does this algorithm always return an optimal solution? Prove.
 - What does this say about the number of nodes in the optimal solution to this problem as a function of the number of nodes in the tree? i.e., If I tell you the number of nodes in the instance, what can you tell me about the number of nodes in an optimal solution?
- (b) Consider the following greedy algorithm, given a tree T . The greedy criteria sort the nodes in reverse order by their distance from the root, i.e., the algorithm starts with the bottom leaves and works its way up the tree level by level until it reaches the root. The key point in this order is that when the algorithm considers node u , it has already considered all of its children. The algorithm then commits to include this node u in S unless u has a child that has already been included, in which case it excludes u .
- (c) Which of the following are true? For each, use the prover-adversary game from [Chapter 34](#) to prove it.
 - i. $\forall$ trees T , $\forall$ optimal solutions S of T , $\forall$ leaf nodes u of T , $u \in S$.
 - ii. $\forall$ trees T , $\exists$ optimal solutions S of T , $\forall$ leaf nodes u of T , $u \in S$.
 - iii. $\forall$ trees T , $\exists$ optimal solutions S of T , $\exists$ leaf nodes u of T , $u \in S$.

III.25. **Recursive** (*Independent sets in trees*): (Questions III.24, III.25, and III.38 are related.) Reconsider the exact same problem given above and the exact same greedy algorithm for it. Let S_T denote the optimal solution returned by this greedy algorithm for tree T .

- (a) Give a recursive algorithm that takes as input a tree T and returns the set S_T .
Hint: You may have to redefine the postcondition so that your algorithm returns some extra information. If so, clearly document the change.
- (b) Give a short paragraph arguing why your code gives the correct answer.
- (c) Give a short paragraph explaining the running time of your algorithm.

32.3 Dynamic Programming

III.26. (a) Decisions: In Dynamic Programming (Recursive BackTracking) a decision also needs to be made about each object. Generally, this is about whether or not to take the last object. Do we have a bird that we can trust to make this decision? If so, why can we trust her? If not, how does the algorithm decide which choice to make?

- (b) Question Size: We say that the question asked of the little bird must be “small.” What is the definition here of “small”? Give an example of a big/small question.
 - (c) Friend: Do we have a friend that we can trust? If so, why can we trust him?
 - (d) $optS_{(i,k)}$: Define this in a dynamic programming algorithm. How do you construct it?
 - (e) $optS[i]$: Define this in a dynamic programming algorithm. How do you construct it?
 - (f) Subinstance: Starting with some input instance $\langle a_1, a_2, \dots, a_n \rangle$, what is the set of instances that might get asked in the recursive backtracking algorithm of some friend’s friend’s friend . . . ?
 - (g) Table: In a general dynamic program, what is the table indexed by? In a fast dynamic program, what does it store? Why do we not store the obvious thing?
 - (h) Recursion: Does a dynamic program recurse? Does it have “friends”? Explain.
 - (i) LCS: Determine the longest common subsequence of $X = \langle w, f, q, f, e, w, f \rangle$ and $Y = \langle f, e, q, w, f, w \rangle$.
- III.27. **Cross Array:** Design an algorithm for crossing an array in the minimum number of steps. At each point in time, you are standing on one of the elements. Each step you can move to any element with the same value, move forward one or move back one. For example, for $\{1, 2, 3, 4, 1, 5\}$ move from the first 1 to the second 1 to the 5.
- (a) The obvious greedy algorithm moves each step the furthest to the right it can.
Prove whether or not this works.
 - (b) Given such an array, produce a Game of Life Graph (see [Section 26.3](#)) indicating how you are able to move so that its st -shortest path gives a solution.
 - (c) Also develop the bird and friend algorithm for this and its code.
- III.28. **Integer Knapsack Problem:** For the problem in general, no polynomial algorithm is known. However, if the volume of the knapsack is a small integer, then dynamic programming provides a fast algorithm.

Instances: An instance consists of $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$. Here, V is the total volume of the knapsack. There are n objects in a store. The volume of the i^{th} object is v_i , and its price is p_i .

Solutions: A solution is a subset $S \subseteq [1 \dots n]$ of the objects that fit into the knapsack, i.e., $\sum_{i \in S} v_i \leq V$.

Measure of Success: The cost (or success) of a solution S is the total value of what is put in the knapsack, i.e., $\sum_{i \in S} p_i$.

Goal: Given a set of objects and the size of the knapsack, the goal is to fill the knapsack with the greatest possible total price.

Your Task: A Game of Life solution is given in Section 26.3. As done in Chapter 24.1, do the following: Give the Bird and Friend algorithm for it. What is the set of subinstances? How is the solution and its cost constructed for your subinstance? What is the recursive backtracking algorithm for it? When converting this to dynamic programming, how do you index the table and in what order do you fill it? Give the dynamic programming code. Finally, what is its running time and how do you compute it?

- III.29. **Stock Market Prices:** You are very lucky to have a time machine bring you the value each day of a set of stocks. The input instance to your problem consists of $I = \langle T, S, \text{Price} \rangle$, where T is an integer indicating your last day to be in the market, S is the set of $|S|$ stocks that you consider, and Price is a table such that $\text{Price}(t, s)$ gives the price of buying one share of stock s on day t . Buying stocks costs an overhead of 3%. Hence, if you buy p dollars worth of stock s on day t , then you can sell them on day t' for $p \cdot (1 - 0.03) \cdot \frac{\text{Price}(t', s)}{\text{Price}(t, s)}$. You have one dollar on day 1, can buy the same stock many times, and must sell all your stock on day T . You will need to determine how you should buy and sell to maximize your profits.

Because you know exactly what the stocks will do, there is no advantage in owning more than one stock at a time. To make the problem easier, assume that at each point in time there is at least one stock not going down and hence at each point in time you always own exactly one stock. A solution will be viewed as a list of what you buy and when. More formally, a solution is a sequence of pairs $\langle t_i, s_i \rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1} . (Here $i \geq 1$, $t_1 = 1$, and $t_{\text{last}+1} = T$.) For example, the solution $\langle \langle 1, 4 \rangle, \langle 10, 8 \rangle, \langle 19, 2 \rangle \rangle$ means that on day 1 you put your one dollar into the 4th stock, on day 10 you sell all of this stock and buy the 8th stock, on day 19 you sell and buy the 2nd stock, and finally on day T you sell this last stock. The value of this solution is $\prod_i \left[(1 - 0.03) \cdot \frac{\text{Price}(t_{i+1}, s_i)}{\text{Price}(t_i, s_i)} \right] = 1 \cdot (1 - 0.03) \cdot \frac{\text{Price}(10, 4)}{\text{Price}(1, 4)} \cdot (1 - 0.03) \cdot \frac{\text{Price}(19, 8)}{\text{Price}(10, 8)} \cdot (1 - 0.03) \cdot \frac{\text{Price}(T, 2)}{\text{Price}(19, 2)}$. (Note that the symbol $\prod_i$ works the same as $\sum_i$ except for product.)

Your Task: A Game of Life solution is given in Section 26.3. Do all the steps listed in Chapter 31.

- III.30. Cleaning up signals and compression and machine learning all have a similar idea.

Cleaning:

- We have a prior belief that the signal we are being sent should come from some simple class of functions, e.g., piecewise constant, or linear, or low-degree polynomial.

- Noise tends to high frequency, i.e., it changes fast. An example is the hiss on a radio.
- Given $x = \text{signal} + \text{noise}$, the goal is find x' that is in the simple class and is close to x .

Compression:

- (a) The image taken by our camera has colors that change slowly, and those that change fast.
- (b) Our eyes find it just as nice to look at it if the fast changes have been removed.
- (c) Compression removes these fast changes because then the file is smaller.
- (d) Given $d = \text{photo}$, the goal is find x' that is in the simple class and is close to x .

Machine Learning:

- We have a prior belief that the brain does not distinguish images of cats and people by memorizing them. Instead, the brain uses a simple neural net.
- Machine learning when given data d , instead of memorizing it all, simplifies it to be x' .
- Given $x = \text{data}$, the goal is find x' that is in the simple class and is close to x .
- The hope is that x' will be better able to generalize to new images.

Sebastian Magierowski gave me this problem. Given a noisy signal x , your task is to clean up the noise. The solution x' splits the signal into some b blocks and makes it constant within each block, i.e., x' is piecewise constant. The cost of the solution will be the sum of two components. The first is the sum of the square of the difference between the original signal x and your approximation x' . In order to avoid the solution $x' = x$, the second cost will charge $1000b$, where b denotes the number of constant blocks in the cleaned up signal x' .

Input: The input is the noisy signal $x = \langle x[1], \dots, x[n] \rangle$, where for $\forall i \in [1 \dots n]$, $x[i]$ is a real valued signal level.

Solution: The solution $sol = [r_1, r_2, \dots, r_b]$ breaks the signal into some b blocks within which x' will be constant. Here $b \geq 1$ is the number of blocks and $\forall j \in [1..b]$, add r_j is the index of the first value in the j^{th} block, i.e., the block consists of the subsignal $\langle x[r_j], \dots, x[r_{j+1}-1] \rangle$.

Valid Solutions: The first block starts at $r_1 = 1$, the last ends at $r_{b+1}-1 = n$, and in between $r_j < r_{j+1}$, so the blocks are disjoint.

The Cleaned Up Signal: For the duration of each of its blocks, the signal x' will be set to v_j which is the average value of x within this block, namely, $\forall j \in [1 \dots b]$, $v_j = \text{Avg}_{i \in [r_j, r_{j+1}-1]} x[i]$, and $\forall i \in [r_j \dots r_{j+1}-1]$, $x'[i] = v_j$.

Cost of Solution: The cost of the solution can be broken down into the sum of the cost for each of its blocks, namely, $cost = \sum_{j \in [1 \dots b]} cost(r_j, r_{j+1} - 1)$.

Cost of Block: The cost of the j^{th} block will be $cost(r_j, r_{j+1} - 1) = \sum_{i \in [r_j \dots r_{j+1} - 1]} (x[i] - v_j)^2 + 1000$.

Hint: The solution $sol = [r_1, r_2, \dots, r_b]$ is a sequence of numbers. In such a case, what is the standard thing to ask the little bird?

Hint: It is true that $cost(r_j, r_{j+1} - 1)$ has a complicated equation. (Though it is also fairly standard.) My advice is that you completely ignore what this equation is because it has zero effect on your code, as your code can simply call $cost(r_j, r_{j+1} - 1)$ as a subroutine.

- (a) Give the bird and friend algorithm.
- (b) Write easy-to-read pseudocode for the dynamic programming solution to this problem. Include all the comments given in the slides (except for the bird and friend algorithm, as it has been given already).
- (c) What is the running time of this algorithm? Where do these numbers come from?

III.31. **Printing Neatly:** Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \dots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of “neatness” is, as follows. If a given line contains words i through j and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^j l_k$. We wish to minimize the sum, over all lines of the cubes of the numbers of extra space characters at the ends of lines. (The question in the text excludes the cube of the number of spaces on the end of the last line from this cost, but this makes the problem harder.) Give a dynamic programming algorithm that determines the optimal number of words to be printed on each line.

- (a) What aspects must a computational problem have before we call it an optimization problem?
- (b) What is the “brute force” algorithm for an optimization problem? Generally, what is its running time?
- (c) For the Printing Neatly problem, what is an instance, a solution, the value/cost of a solution, and goal of the algorithm?
- (d) Suppose that you must find an optimal solution $\langle k_1, \dots, k_r \rangle$ for a given instance $\langle M, l_1, \dots, l_n \rangle$ to the Printing Neatly problem. To help, you have a little bird and a group of friends. Explain in **3 or 4 sentences** the algorithm for doing this.
- (e) Suppose the bird dies. What changes do you make to the above algorithm? Explain in **3 or 4 sentences**.

- (f) What is the running time of this recursive algorithm? Explain why.
- (g) How would you convert a recursive algorithm into a dynamic programming algorithm? Explain in **3 or 4 sentences**.
- (h) How do you prove that your set of subinstances is
 - i. Not too small?
 - ii. Not too big?
- (i) How do we reduce the running time of the dynamic program for the Printing Neatly problem from $\Theta(n^3)$ to $\Theta(n^2)$? Be sure to specify where your numbers come from.
- (j) What do we store in the table?
- (k) Write-easy-to read pseudocode for the dynamic programming solution to this problem. It should fill in both the m and the s tables.
- (l) Given the s table, write an algorithm that returns the optimal number of words for each line of text. (No proofs or LI needed.)
- (m) How is the final optimal solution obtained?
- (n) What are the key numbers in computing the running time of the dynamic program? Be sure to specify where your numbers come from.

III.32. Longest Balanced Substring

- (a) Shortest Prefix: The input consists of an array $A[1..n]$, with each element from $\{0, 1, 2, 3\}$. The output is a matrix B , such that for each $x, y, z \in [-n \dots n]$, $B[x, y, z] = i$ where $A[1 \dots i]$ is the **shortest** prefix of A that has x more 1s than 0s, y more 2s than 0s, and z more 3s than 0s. The running time must be $O(n)$. You may assume that memory is initially zeroed.
- (b) The previous question uses lots of memory. How would you use Hash Tables from [Section 3.1](#) in order to save both time and memory?
- (c) Again, the input consists of an array $A[1 \dots n]$, with each element from $\{0, 1, 2, 3\}$. The output now is a longest contiguous subarray $A[i \dots j]$ with equal number of 0's, 1s, 2s, and 3s. The running time must be $O(n)$.

III.33. **Longest “#0–#1 = 2 mod 5” Block:** This time the input consists of an array $A[1 \dots n]$, with each element 0 or 1. The output is a longest contiguous subarray $A[i \dots j]$ whose number of occurrences of 0s minus its number of occurrences of 1s is 2 mod 5. Design and analyze an incremental algorithm to solve this problem in $O(n)$ worst-case time.

III.34. **Edit Distance:** The input to this problem is a pair of strings $A = \langle a_1, \dots, a_n \rangle$ and $B = \langle b_1, \dots, b_m \rangle$. A solution is a sequence of operations to convert A into B . The allowed operations are as follows. For a cost of 3 you can delete any letter. For a cost of 4 you can insert a letter in any position. For a cost of 5 you can replace any letter by any other letter. The goal is to find a way to convert A into B that is as cheap as possible.

$A = a \ b \ c \ a \ b \ c$
 $\quad \quad |0 \ |0 \ |3a \ 2c \ 0 \ 0 \ 1$
 $B = a \ b \ a \ c \ a \ b$
 $Sol = 0, 0, 3a, 2c, 0, 0, 1$
 $cost = 5 + 4 + 3 = 12$

$A = a \ b \ c \ a \ b \ c$
 $\quad \quad |0 \ |0 \ |2a \ 0 \ 0 \ 0 \ 1$
 $B = a \ b \ a \ c \ a \ b$
 $Sol = 0, 0, 2c, 0, 0, 0, 1$
 $cost = 4 + 3 = 7$

For example, you can convert $A = abcabc$ to $B = abacab$ via the following sequence $Sol = \langle 0, 0, 3a, 2c, 0, 0, 1 \rangle$, meaning keep the a and the b the same, change the c into an a , insert a c , keep the a and the b the same, and delete the c . Note that where each operation occurs in the string does not need to be specified because it follows naturally from left to right. The total cost for this would be $0 + 0 + 5 + 4 + 0 + 0 + 3 = 12$. A cheaper solution would be $Sol = \langle 0, 0, 2c, 0, 0, 0, 1 \rangle$, meaning keep the a and the b the same, insert an a , keep the c , a , and b , the same, and delete the c for a cost of only $0 + 0 + 4 + 0 + 0 + 0 + 3 = 7$. This is likely optimal.

To be helpful, I will tell you the question for the little bird. I am trying to find an optimal solution for converting $A = \langle a_1, \dots, a_n \rangle$ into $B = \langle b_1, \dots, b_m \rangle$. I ask them about the last characters a_n and b_m . They give me one of the following answers.

Case 0: Because $a_n = b_m$, you might as well leave a_n and simply have it become b_m . (cost 0)

Case 1: Delete the letter a_n because it is not needed in $B = \langle b_1, \dots, b_m \rangle$. (cost 3)

Case 2: Insert the letter b_m because it will not come from $A = \langle a_1, \dots, a_n \rangle$. (cost 4)

Case 3: Change the letter a_n into b_m . (cost 5)

- (a) For each of these four cases, what subinstance do you give your friend?
- (b) Give the code for the dynamic programming algorithm. You can use sentences in some places to replace some blocks of code. (Do not include the code that constructs the solution.)
- (c) What is the running time to fill in the table? Be sure to indicate where the numbers come from.

III.35. **Counting Solutions:** Consider an arbitrary dynamic programming algorithm that given an instance returns one of the possibly many optimal solutions.

- (a) Suppose that in the bird and friend's version of the dynamic programming algorithm, when given an instance of size n , the bird is asked a question with K possible answers and then one friend is asked to solve a subinstance that is never any bigger than size $n - 1$. Briefly describe what needs to be changed so that in addition to outputting one of the optimal solutions to the given instance, the dynamic program also outputs the number of possible optimal solutions. Your change should not significantly change the running time.

- (b) Let $Num(n)$ denote the maximum number of possible optimal solutions that any dynamic program, as described above, might output given an instance of size n . Give and solve the recurrence relation for $Num(n)$.
- (c) Suppose that there were this maximum number of solutions. What is the minimum number of bits needed to describe each with a unique string? How would you describe a solution as a sequence of the bird's answers? How does the length of this description compare with the minimum length?
- (d) Why can the program in polynomial time output the number of optimal solutions but not be able to output all of these optimal solutions?
- (e) Now suppose that in the bird and friend's version of the dynamic programming algorithm, when given an instance of size n , the bird is asked a question with K possible answers and then two friend each are asked to solve a subinstance that is never any bigger than size $\frac{n}{2}$. (See "Chains of Matrix Multiplications"). Again briefly describe what needs to be changed so that the dynamic program also outputs the number of optimal solutions.
- (f) Now let $Num(n)$ denote the maximum number of optimal solutions that any dynamic program as described in the last question might output given an instance of size n . Give the recurrence relation for $Num(n)$. In order to solve it, assume that $Num(n) = b^{n-1}$ for some constant b and solve for b .

III.36. **Decoding**

Consider an encoding of characters into binary strings in which different characters may be encoded into strings of different length. We may describe such an encoding scheme in a table with two columns, where each character on the left is encoded by the bit string given on the right, as in the example here.

Example Decoding Table	
Character	Encoding
A	0
B	10
C	01
D	11

Using this table, the encoding of ACDC is 001110 (which comes from 0 01 11 01, but the decoder is not given the separation between the code-words). Notice that there may potentially be several decodings of a given bit string. For instance, 001110 can be decoded either as AADB or as ACDA. Your task is to devise an efficient algorithm that, given an encoded string $\langle a_1, \dots, a_n \rangle \in \{0, 1\}^n$ and a decoding table, decodes the string into one of its possible string of characters. For example, if $\langle a_1, \dots, a_n \rangle = 001110$ and the Decoding Table is as above, then $optSol$ is either AADB or ACDA.

- (a) Choose one of these bird questions and argue why it is the best.
 - i. What is the first character decoded?
 - ii. How many bits of the input do I use in the code for the last letter?
 - iii. How many characters are there in the final decoded string?
 - iv. What is the last character decoded?
 - v. What is encoding of the last character decoded?

Give a bird/friend algorithm for this problem.

- (b) Give the code for the dynamic programming algorithm. (Be sure to explain each part of the code as I have been doing.)

algorithm *Decode* ($\langle a_1, \dots, a_n \rangle, \text{DecodingTable}$)

<pre-cond>: An instance consists of a coded message $\vec{a}$ and a Decoding Table.

<post-cond>: *optSol* is a decoded messages.

For example, if $\langle a_1, \dots, a_n \rangle = 001110$ and the Decoding Table is as above, then *optSol* is either AADB or ACDA.

The program also returns the “cost” of the solution, which is 1 if it is a valid decoding and 0 if it is not.

- (c) What is the running time to fill in the table? Be sure to specify where your numbers come from.

In particular, think about what the running time should be when each letter in the table has the exact same length.

III.37. Consider the recurrence relation $T(0) = T(1) = 2$ and for $n > 1$ $T(n) = \sum_{i=1}^{n-1} T(i) \times T(i-1)$. We consider the problem of computing $T(n)$ from n .

- (a) Show that if you implement this recursion directly that the program would use exponentially in n many arithmetic operations.
- (b) Explain how to use dynamic programming that only uses $O(n^2)$ arithmetic operations for this problem.
- (c) Give an algorithm for this problem that only uses $O(n)$ arithmetic operations.
- (d) Which of these approximates $T(n)$ the closest?

$$n^{\Theta(n)} \qquad 2^{\Theta(n)} \qquad 2^{2^{\Theta(n)}}$$

III.38. **Weighted Independent Sets in Trees:** (Questions III.24, III.25, and III.38 are related.) The input consists of a rooted tree T in which each node u has a weight w_u (Figure 32.6). A solution consists of a subset S of the nodes of tree T . Such a solution is valid if no two of the nodes in S have an edge between them (i.e., one is the parent of the other in T). The cost/value of S is the sum of its node weights, i.e., $\text{cost}(S) = \sum_{u \in S} w_u$. Our goal, given a tree T , is to find a valid solution whose value is maximized.

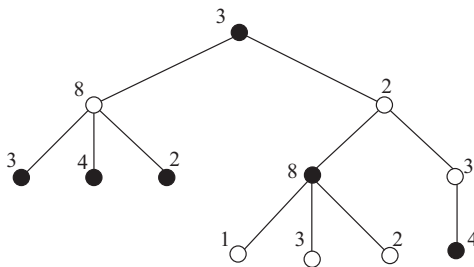


Fig. 32.6

- (a) Suppose you have an instance T , a little bird, and as **many** recursive friends as you like. Give a good bird/friend algorithm solving this problem.

Hint: Ask the bird whether or not to take the root.

For both this question and the coding question, feel free to say things like

“For each child u of node v , consider the subtree T_u and do”

“For each grandchild u of node v , consider the subtree T_u and do”

- (b) What is the set of subinstances that needs to be solved?
In what order should these subinstances be solved.
- (c) Give the dynamic programming code.
- (d) What is the running time of the dynamic programming algorithm?

(Remember there may be more than one friend per bird answer.)

III.39. **Aisled Knapsack Problem:** In the knapsack problem, you must select a subset of the n objects in the store to put in your knapsack. The total volume of the objects taken can’t exceed the volume of the knapsack. Your goal is to maximize the total price of the objects taken. Now the store is laid out into n aisles each with m treasures and for each aisle, you must choose exactly one treasure.

Hint: In understanding this problem, it is best not to consider each of the nm treasures as an object for which you need to decide whether or not to take it. Instead, think of each of the n aisles as being an “object” for which you need to decide which treasure to take it.

More formally, the problem is defined as follows.

Instances: An instance consists of $I = \langle V, \langle \langle v_{\langle i,j \rangle}, p_{\langle i,j \rangle} \rangle \mid i \in [n], j \in [m] \rangle \rangle$.

Here, V is the total volume of the knapsack. There is a store with n aisles, each containing m treasures. For $i \in [n]$ and $j \in [m]$, the j^{th} treasure in the i^{th} aisle has volume $v_{\langle i,j \rangle}$ and price $p_{\langle i,j \rangle}$.

Solutions: A solution is $S = \langle j_1, \dots, j_n \rangle$, where for $i \in [n]$, $j_i \in [m]$ specifies which treasure was taken from aisle i .

Valid: A solution is valid if the taken treasures fit into the knapsack, i.e., $\sum_{i \in [n]} v_{\langle i, j_i \rangle} \leq V$.

Measure of Success: The value of a solution is the total price of what is put in the knapsack, i.e., $\sum_{i \in [n]} p_{\langle i, j_i \rangle}$.

Goal: Given the treasures in the aisles, the goal is to fill the knapsack with one treasure from each aisle with the greatest possible total price.

Hint: If the smallest volume treasure from each aisle together do not fit in the knapsack, then the value of the optimal solution will have value $-\infty$.

- (a) Bird and Friend: Consider a particular instance, $\langle V, \langle \langle v_{\langle i,j \rangle}, p_{\langle i,j \rangle} \rangle \mid i \in [n], j \in [m] \rangle \rangle$ to the Aisled-Knapsack problem. Give an algorithm using trusted bird and friend.

Hint: Is there a case in which you simply know the bird was wrong?

- (b) What is the set of subinstances solved by the recursive backtracking algorithm? How many are there?
- (c) What is the running time of your dynamic programming code in terms of V , n , and m ?
- (d) This aisled-knapsack problem has nm treasures. What is the running time of the dynamic programming algorithm for the normal Aisleless-Knapsack problem when it has nm objects to choose from? Again, give your answer in terms of V , n , and m .
- (e) What is the running time of your program expressed as a function of input size?
- (f) Give the dynamic programming code.

III.40. **Longest Increasing Subsequence:** The input consists of a sequence of integers $I = \langle a_1, a_2, \dots, a_n \rangle$. A solution is a subsequence $S = \langle b_1, b_2, \dots, b_{r-1}, a_n \rangle$ in that the integers in S all appear in I in the same order, but not necessarily contiguously. For such a solution to be valid, it is increasing and ends in a_n , namely, $b_1 < b_2 < \dots < b_{r-1} < a_n$. The cost/value of the solution is the length r of S . The goal is to find such a longest increasing subsequence. For example, an optimal solution for $\langle 5, 1, 8, 10, 11, 12, 13, 14, 5, 2, 7, 2, 4, 9 \rangle$ is $\langle 1, 5, 7, 9 \rangle$ and so is $\langle 1, 2, 4, 9 \rangle$.

- (a) Suppose you have an instance I , a little bird, and a recursive friend. Give a good bird/friend algorithm solving this problem.
Hint: You have been asking the little bird a question that has two answers yes/no; however, **DO NOT** do this. Instead, ask them a question that has about n different answers. This will work out better for you in the long run.
- (b) What is the set of subinstances that needs to be solved?
- (c) Give the dynamic programming code.
- (d) What is the running time?
- (e) The actual problem that we want to solve is the same except that the last element b_r of the solution does not need to be the last element a_n of I . Given an instance $I = \langle a_1, a_2, \dots, a_n \rangle$, how would you solve this problem with the minimum effort on your part?

III.41. **Search for Non-Overlapping Words**

The input is an ASCII string I . A solution is a partition of this string into substrings. Let's represent this with $Sol = \langle k_1, \dots, k_r \rangle$, where k_a tells you how many letters are in the a^{th} substring. Note that $\sum_{a \in [r]} k_a = |I|$. The contribution value/cost $cost_a$ of the a^{th} substring is the square k_a^2 of its length if this substring appears in the dictionary. If the substring is not in the dictionary, then its contribution cost is $cost_a = 0$. The cost of the solution Sol is the sum $cost = \sum_a cost_a$ of these contributions. Given an instance I , the goal is to find the solution with the maximum cost. For example, let the input be $I = \text{goodxkdf luckxyzaonzsthez wexamad}$. The optimal solution would partition I into the substrings *good xkdf luck xyza on zs the zw exam ad*, giving an optimal solution of $optSol = \langle 4, 4, 4, 4, 2, 2, 3, 2, 4, 2 \rangle$ and optimal cost of $optCost = 4^2 +$

$4^2 + 2^2 + 3^2 + 4^2$ because the substrings *good luck on the exam* are each in the dictionary but others are not.

Hint: The algorithm for this is almost identical to that of the Printing Neatly problem.

- (a) Suppose that you must find an optimal solution $\langle k_1, \dots, k_r \rangle$ and its cost for a given instance I to the Non-Overlapping Words problem. To help, you have a trusted little bird and a group of trusted friends. Explain in **4 or 5 sentences** the algorithm for doing this. Be sure to say how the cost is computed.
- (b) Define $optSol_{(I,k)}$. Do not tell how to compute it. State in one sentence.
- (c) What is the running time of the dynamic program algorithm? Be sure to specify where each of your numbers comes from.
- (d) Write the dynamic programming code for this problem.

algorithm *NonOverlappingWords* (I)

pre-cond: I is an ASCII string.

post-cond: $optSol = \langle k_1, \dots, k_r \rangle$ is an optimal partitioning of this string into substrings, where k_a tells you how many letters are in the a^{th} substring. $optCost = \sum_a \langle a \text{ where this substring is in the dictionary} \rangle k_a^2$ is its cost.

begin

32.4 Reductions and NP-Completeness

III.42. Short Answers

- (a) What is a computational problem P ? Give two examples.
- (b) What is an algorithm A ? When does it solve problem P ?
- (c) What is the time complexity of an algorithm A ?
- (d) What is the time complexity of a computational problem P ? What are upper and lower bounds on this?
- (e) What is constant, linear, polynomial, and exponential time? Practically, why does it matter? What are these times when $c = 2$ seconds and $n = 150$?
- (f) What is an *Optimization Problem* P ?
- (g) How do you make an optimization problem into a *decision problem*? Give an example. What is a *witness*?
- (h) What does it mean for a decision problem P to be in *NP* (*Non-Deterministic Polynomial Time*)? This was defined by Jeff Edmond's dad, Jack Edmonds. (Do this without defining or referring to Non-Deterministic Turing Machines.)
- (i) Your boss gives you an instance I of an NP problem that by good luck happens to be a *Yes* instance. You are blessed to have a powerful fairy godmother to help you. How do you convince your boss that the answer for their instance is *Yes* without then knowing or being affected by your fairy godmother?

Similarly, how would you convince your boss that the answer for their instance is *No*?

- (j) For an NP problem, is there a limit on the size of a solution for I and/or on the number of possible such solutions? Why?
- (k) What is the *brute force* algorithm for P and how long does it take?
- (l) What is the famous problem $P = NP$?
- (m) What does it mean for a decision problem P to be *NP-Hard* or *NP-Complete*? What does this say about the time complexity of such a problem?
- (n) How is this useful to you in the real world?
- (o) In practice, what is done to know whether the answer for the instance is yes or no?
- (p) How do you prove that one computational problem is at least as hard as another? $P_1 \leq P_2$. What is an *oracle*? Note that this is used in the definition of NP-Hard.
- (q) Outline the basic code for Alg_{alg} solving P_{alg} using the supposed algorithm Alg_{oracle} supposedly solving P_{oracle} as a subroutine. If Alg_{oracle} is kind and also provides a valid solution S_{oracle} for its instance I_{oracle} , then you should provide a valid solution S_{alg} for your instance I_{alg} .
- (r) What steps do you have to take to prove that this reduction is correct?
- (s) Give two purposes of reductions.

III.43. **Fun Faulty Reductions:** The purpose of this question is to learn to appreciate why [Section 31.2.3](#) has so many steps when doing a reduction.

0) **$P_{oracle} \in \text{NP}$:** Prove that problem P_{oracle} is in NP.

5) **InstanceMap:** Define a polynomial time mapping $I_{oracle} = \text{InstanceMap}(I_{alg})$.

6) **SolutionMap:** Define a polynomial time algorithm mapping each potential solution S_{oracle} for I_{oracle} to a potential solution $S_{alg} = \text{SolutionMap}(S_{oracle})$ for I_{alg} .

7) **Valid to Valid:** Prove that if S_{oracle} is a valid solution for I_{oracle} , then $S_{alg} = \text{SolutionMap}(S_{oracle})$ is a valid solution for I_{alg} .

8) **ReverseSolutionMap:** Define a polynomial time algorithm mapping each potential solution S'_{alg} for I_{alg} to a potential solution $S'_{oracle} = \text{ReverseSolutionMap}(S'_{alg})$ for I_{oracle} .

9) **Reverse Valid to Valid:** Prove that if S'_{alg} is a valid solution for I_{alg} , then $S'_{oracle} = \text{ReverseSolutionMap}(S'_{alg})$ is a valid solution for I_{oracle} .

Consider the problem *Block* whose instance $\langle s, k \rangle$ is a 01-string s and an integer k . Such an instance is a Yes instance if it has a solution consisting of a block $s_i, s_{i+1}, \dots, s_j$ of all ones whose length is k .

- (a) **NP:** Is this Block problem in NP? If so, what would this mean?

- (b) **NP-Complete:** Is this Block problem in NP-Complete? What would it mean?
- (c) **No Instance Steps 6 and 7:** Your homework is to prove that Block is NP-complete by proving that $3\text{-SAT} \leq_{\text{poly}} \text{Block}$. You are to work in pairs. Your partner took on step 5 as follows. P_{alg} is 3-SAT. As an example, $I_{3\text{-SAT}} = (x \text{ or } \bar{y} \text{ or } z) \text{ AND } \dots \text{ AND } (\bar{u} \text{ or } v \text{ or } w)$ is an instance of 3-SAT. Given such an instance, they map it to an instance $I_{\text{Block}} = \langle s, k \rangle$ where $s = 000 \dots 0$ is the string consisting of n zeros and k is 5. Besides a few hints, which they ignored, they noted that they did in fact meet the formal requirements of step 5. Being a bit of a know-it-all, your partner thought steps 8 and 9 are silly and hence decided to skip them. They left you the task of doing steps 6 and 7. Carefully, review what these steps are and complete them if you can. Careful.
- (d) **Yes Instance Steps 8 and 9:** Because your previous partner dropped out of school, you got a new one. Having learned, the new partner took a new approach. They decided to map an instance $I_{3\text{-SAT}}$ of 3-SAT to an instance $I_{\text{Block}} = \langle s, k \rangle$ where $s = 111 \dots 1$ is the string consisting of $n \geq 5$ ones and k is 5. They went on to do steps 6 and 7 as follows. Given a solution $s_i, s_{i+1}, \dots, s_j$ to I_{Block} , they instructed how to construct a valid solution to $I_{3\text{-SAT}}$ as follows. For each of the m clauses $(x \text{ or } \bar{y} \text{ or } z)$ of $I_{3\text{-SAT}}$ choose one of the variables. If, like x , it is not negated in the clause then set the variable to true. If, like y , it is negated, then set it to false. Either way, the clause is satisfied. Doing this for each clause satisfies each clause. Hence, the assignment obtained is a valid assignment of the variables. This new partner learned (from the partner who dropped out) the importance of doing steps 8 and 9 so they left them for you to do. Carefully, review what these steps are and complete them if you can. Careful.
- (e) **Yes Instance Steps 5, 6, and 7:** Are you happy with your new partner? Did they follow steps 5, 6, and 7 correctly? Explain.

III.44. **Reduction between Problem and a Restriction of It:** Let P_{general} and $P_{\text{restricted}}$ be the same computational problem except that $P_{\text{restricted}}$ is defined only on a subset of the instances on which P_{general} is defined. For example, 3-SAT is the same problem as SAT except that each clause has at most three literals. *Clique* is given a graph G and an integer k and asks whether there is a clique in G of size k . 3-*Clique* is the same with k fixed to 3.

- (a) Compare and discuss the meaning of $P_{\text{general}} \leq_{\text{poly}} P_{\text{restricted}}$ vs. $P_{\text{restricted}} \leq_{\text{poly}} P_{\text{general}}$. What are the intuitions as to why they may be true and why they may not be true? Which is easier to prove?
- (b) Which of $\text{SAT} \leq_{\text{poly}} 3\text{-SAT}$ and $3\text{-SAT} \leq_{\text{poly}} \text{SAT}$ are true? Give in one sentence how you know.
- (c) Which of $\text{Clique} \leq_{\text{poly}} 3\text{-Clique}$ and $3\text{-Clique} \leq_{\text{poly}} \text{Clique}$ are true? Give in one sentence how you know.
- (d) Do steps 5 through 9 to prove that $3\text{-SAT} \leq_{\text{poly}} \text{SAT}$. Hint: This REALLY is easy.

III.45. **Card Puzzle:** Consider the following two computational problems and following correspondence between them.

	Card Puzzle	Clause-SAT			
Input:	A box and cards n cards: Each labeled with var. and its negation on back Each card has m rows - card v, j^{th} row, & 1^{st} col: non-hole - card v, j^{th} row, & 2^{nd} col: non-hole	The <i>and</i> of clauses: Each of the <i>or</i> of vars n variables m clauses - var v appears positively in j^{th} clauses - var v appears negatively in j^{th} clauses			
Solution:	Each card flipped or not and put in box	Each variable negated or not			
Satisfied:	Each X has a nonhole of card covering it - e.g., Flipped card a covers last X	Each clause has a variable satisfying it - e.g., Negated a satisfies last clause			
Determine:	Ans Yes iff $\exists$ a satisfying solution	Ans Yes iff $\exists$ a satisfying solution			
See more:	4111/ass/06-ass-NP_Complete.pdf Q3	2001-60-NP_Complete.pptx Quick Reduce			
Example:					
	Box	Cards	Flipped	Clauses	
		a b c d e	$\neg a$		
X		○ ○ ○ ○ ○	○	$(a \text{ or } \neg b \text{ or } d)$	and and and and
X		○ ○ ○ ○ ○	○ ○	$(\neg a \text{ or } \neg c \text{ or } d \text{ or } e)$	
X		○ ○ ○ ○ ○	○ ○	$(b \text{ or } \neg c \text{ or } \neg e)$	
X		○ ○ ○ ○ ○	○ ○	$(c \text{ or } \neg d)$	
X		○ ○ ○ ○ ○	○	$(\neg a \text{ or } \neg c \text{ or } e)$	

- (a) A correspondence between the following has already been spelled out:
- an instance of the card puzzle and
 - an instance of clause-SAT.
- Spell out a similar correspondence between
- a solution of the card puzzle and
 - a solution of clause-SAT.
- (b) Is there a correspondence between
- an instance of the card puzzle requiring a Yes and
 - the corresponding instance of clause-SAT requiring a Yes?
- Explain.
- (c) A problem P is in NP (*Non-Deterministic Polynomial Time*) iff there is a *poly-time* algorithm $Valid(I, S)$ that, given an instance I and a solution S ,

tests whether S is a valid solution for I . A key point is that the algorithm $Valid(I, S)$ runs in polynomial time in the size of the instance I .

Formally, we say that $P \in NP$ iff $\exists$ algorithm $Valid$ and constant c such that $\forall I, I \in P$ iff $\exists S$ $Valid(I, S) = \text{yes}$ and $\forall I, S, \text{Time}(Valid(I, S)) \leq |I|^c$.

Prove that the Card Puzzle is in NP.

- (d) A problem P is NP-Hard iff $Clause\text{-}SAT \leq P$.

Prove that the Card Puzzle is NP-Hard by giving a few sentences for steps 4 and 5 of the “A Quick Reduction” slides.

III.46. Stacking Truck

Instance: The input is $I_{ST} = \langle C_{truck}, B_1, B_2, \dots, B_n \rangle$, where C_{truck} is the capacity of the truck and $B_i = \langle w_i, c_i, v_i \rangle$ gives weight, capacity, and value of the i^{th} box.

Solution: A solution S_{ST} consists of a permutation of a subset of the boxes.

Valid: A solution S_{ST} is valid if the specified boxes can be put on the truck in this order with nothing breaking. The rule is that for each box i , the sum of the weights of the boxes j above it cannot exceed the amount that can be placed on box i , i.e., $\forall i, \sum_{k \text{ above } i} w_k \leq c_i$. In addition, the weight of the boxes taken cannot exceed the capacity of the truck, i.e., $\sum_{k \text{ above truck}} w_k \leq C_{truck}$.

Cost/Value: The value of a valid solution is the sum of the values of the boxes put on the truck, i.e., $Val(S_{ST}) = \sum_{k \text{ above truck}} v_k$.

- (a) Look up the definition of a computational problem being in NP, i.e., Nondeterministic Polynomial. Prove that Stacking Truck is.

Actually, problems in NP are supposed to be decision problems, not optimization problems. Hence, change input to the Stacking Truck problem to include a threshold V and define the answer to be Yes if there exists a valid solution whose value is at least V .

- (b) Suppose you have an oracle that solves the Stacking Truck problem. Your goal is to describe an algorithm that solves the Knapsack Problem. Suppose you are given in a Knapsack instance $I_{KP} = \langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$ for which you want to find a valid solution. In order to get help from the oracle, you need to give her a Stacking Truck Problem instance $I_{ST} = \langle C_{truck}, \langle w_1, c_1, v'_1 \rangle, \dots, \langle w_n, c_n, v'_n \rangle \rangle$. Describe the instance map that you used to obtain this, i.e., $I_{ST} = \text{InstanceMap}(I_{KP})$. How do you construct the truck and the boxes? Why?
- (c) Suppose after giving your instance I_{ST} to the Stacking Truck oracle, she gives you an optimal solution S_{ST} . Your goal is to construct a solution for the Knapsack Problem. You must prove that the resulting algorithm for Knapsack Problem works.
- Describe the solution map that you used, i.e., $S_{KP} = \text{SolutionMap}(S_{ST})$.
 - Prove that $S_{KP} = \text{SolutionMap}(S_{ST})$ is valid if and only if S_{ST} is valid.

- iii. Prove that $S_{KP} = \text{SolutionMap}(S_{ST})$ is optimal if and only if S_{ST} is optimal.
 - (d) What does this prove about the complexity of the Stacking Truck Problem.
- III.47. **Node Capacities Reduced to Network Flow:** Recall the Network Flow problem. Suppose that in addition to edge capacities, we wanted to allow vertex capacities too. For example, we might want to say that vertex u can have at most 5 units of flow passing through it and vertex v can have at most 3. Show how to convert the maximum flow problem with both node and edge capacities to the standard maximum flow problem that is not too much larger. In other words, imagine we had an instance with both kinds of constraints, and we wanted to solve it using a black-box program that only solved the standard edge-capacity problem.
- III.48. **Greenhouses and Plants Reduced to Network Flow:** A botanical garden is planning to build n greenhouses, each of which is intended to represent a different climate zone. They have a list of m kinds of plants that they would like to grow there. Not every kind of plant can grow in every greenhouse, though usually one kind can grow in more than one climate; so associated with every greenhouse $i \in [1 \dots n]$ is a list C_i of plant varieties it can grow. In addition to that, every greenhouse can host no more than 20 varieties of plants.
- (a) Given the n, m , and C_i for $1 \leq i \leq n$, specify an algorithm to determine the maximum number of different varieties of plants that can be grown in these greenhouses (total over all greenhouses). Hint: Design a flow network that represents this problem.
 - (b) How can you produce a list of chosen varieties for each greenhouse given the maximum flow on your flow network? Is it uniquely determined by the original network?

33 Partial Solutions to Additional Exercises: Part III

33.1 Graph Algorithms

III.1. Short Answers

- (a) The generic loop invariant is that nodes are considered to be either *not found*, *found and not handled*, or *found and handled* and each node has a *parent* node $\pi(v)$.

Found: If a node v is considered *found*, then we have trace a path from s to the node. This path can be traced backwards using $v, \pi(v), \pi(\pi(v)), \pi(\pi(\pi(v))), \dots, s$.

Handled: If a node v is considered *handled*, then all of its neighbors have been found.

- (b) The extra loop invariants used for Breadth First Search is that so far the nodes have been found in the order of their distance from s .
- (c) The extra loop invariants used for Depth First Search is that the nodes in the stack form a path starting at s going to the current node u being considered.
- (d) The preconditions for the Recursive Depth First Search algorithm is that a graph is given (directed or undirected) with some of the nodes marked *found* using bread crumbs and some node s' is specified. The postcondition is that every node reachable from s' without visiting a node marked *found* is found.

III.2. **Edge in Cycle:** There is a cycle in G that contains $\langle u, v \rangle$ iff there is a path from v to u . The edge $\langle u, v \rangle$ itself completes the cycle. DFS and BFS will find such a path.

III.6. **Proof Tree:** No matter which x my brother (an adversary) provides, this x will follow a path from the root based on a sequence of questions/answers to some leaf node at which there is a proof of $\text{Fun}(x)$.

33.2 Greedy Algorithms

III.14. Crossing the River

The Greedy Choice: Each iteration the algorithm grabs the object that, according to some simple criteria, seems to be the “best” (or “worst”) from

among the unconsidered objects in the instance. Suppose that the current time is $t - 1$ and the last stone jumped on is the i_{t-1}^{th} stone. In a greedy way, have the man jump on the stone that is as far from i_{t-1} as he can jump, i.e., the largest i_t for which $d_{i_t} - d_{i_{t-1}} \leq 1$. This is an adaptive decision, i.e., it does depend on what objects have been seen already.

You can think of any stone that is before stone i_{t-1} as having greedy criteria weight $-\infty$ because we don't want to jump backward, and any stone that is more than one meter ahead of stone i_{t-1} also has the greedy criteria weight $-\infty$ because we don't want to land in the water and each other stone j has greedy criteria $d_j - d_{i_{t-1}}$. Then you let i_t be the stone j that maximizes this greedy criteria. The algorithm then makes an irrevocable decision about this object. Assuming such a stone exists, the algorithm has the man jump on it. Otherwise, the algorithm stops and states that it is impossible.

The Loop Invariant: We have not gone wrong. If there is a solution, then there is at least one optimal solution S_t extending the choices A_t made so far by the algorithm.

Initially ($\langle pre \rangle \rightarrow \langle LI \rangle$): Initially, no choices have been made and hence all optimal solutions are consistent with these choices.

Maintaining the Loop Invariant ($\langle LI_{t-1} \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI_t \rangle$): Consider an arbitrary iteration.

Fly In from Mars: The algorithm has iterated a few times. **Assume that it is time $t - 1$ and that the algorithm has already decided for the man to step on the stones indexed by $\langle i_0, i_1, i_2, \dots, i_{t-1} \rangle$.** All that we know is that the loop invariant is true and the exit condition is not.

S_{t-1} : The loop invariant states that there is at least one optimal solution extending the choices A_{t-1} made by the algorithm before this iteration. **Let $S_{t-1} = \langle i_0, i_1, i_2, \dots, i_{t-1}, j_t, j_{t+1}, j_{t+2}, \dots, j_\ell \rangle$** denote one such solution. (Note that this is a full solution and as such specifies a decision about each object in the instance and that the first t stones do agree with what the algorithm has done.) I like to have a fairy godmother hold it.

Taking a Step: During the iteration, the algorithm proceeds to choose the “best” object from among those not considered so far and makes an irrevocable decision about it. In a greedy way, the algorithm just had the man jump on the stone that is as far from i_{t-1} as he can jump, i.e., the largest i_t for which $d_{i_t} - d_{i_{t-1}} \leq 1$. There is only this case, because we are assuming that such a stone exists.

Instructions for Modifying S_{t-1} : The prover says to the fairy godmother. “Fairy godmother, I know that you have your full sequence of stones across the river planned out and that like the algorithm, at time $t - 1$, you are standing on the i_{t-1}^{th} stone. If from there you jump to the same stone as the algorithm did,

i.e., $j_t = i_t$, then no changes need to be made to your solution, i.e., $S_t = S_{t-1}$.” Otherwise, because the algorithm stepped as far as possible, she must have stepped closer, i.e., her stone j_t must be between the stones i_{t-1} and i_t . (If she jumped on more than this one stone between i_{t-1} and i_t , then her solution could not have been optimal.) Tell her to jump from stone i_{t-1} not to j_t but to go farther to i_t . From i_t , she should continue on as she did before, i.e., $S_t = S_{t-1} - j_t + i_t = \langle i_0, i_1, i_2, \dots, i_{t-1}, i_t, j_{t+1}, j_{t+2}, \dots, j_\ell \rangle$.

Proving S_t Is an Optimal Solution: By the loop invariant, S_{t-1} is one of the valid solutions for this instance with minimum number of stones. We must prove that going from S_{t-1} to S_t does not increase the number of stones. We added one stone, so we must make sure that we also delete at least one. To prove this, it is sufficient to prove that the next stone j_t that she jumps on after i_{t-1} is to the left of stone i_t . We know that the algorithm chose to jump from stone i_{t-1} to stone i_t because i_t is as the farthest from the initial shore that can be jumped on from i_{t-1} . We also know that the fairy godmother legally jumps from i_{t-1} to j_t . We conclude that j_t is either the same as stone i_t or is closer to stone i_{t-1} than it. It follows that at least one stone j_t is included in the sequence deleted. Hence, S_t is at least as good as S_{t-1} . Hence, S_t is also an optimal solution.

Proving S_t Is a Valid Solution: By the loop invariant, S_{t-1} is a valid solution. (i.e., the initial stone is on the initial shore, the final stone is on the final shore, and each jump has distance at most one). The prover made sure that their modifications did not make it invalid. We have $S_t = \langle i_0, i_1, i_2, \dots, i_{t-1}, i_t, j_{t+1}, j_{t+2}, \dots, j_\ell \rangle$. We know all the jumps from $i_{t'}$ to $i_{t'+1}$ for $t' \in [0, t-2]$ have distance at most one because both the algorithm and the fairy godmother in S_{t-1} have assured it. We know the jump from i_{t-1} to i_t has distance at most one because the algorithm chose it this way. We know the jump from i_t to j_{t+1} has distance at most the distance from j_t to j_{t+1} because we proved that j_t is to the left of i_t , and we know that this distance is at most one because the fairy godmother manages to jump this in S_{t-1} . Finally, we know all the jumps from $j_{t'}$ to $j_{t'+1}$ for $t' \geq t+r$ have distance at most one because the fairy god mother in S_{t-1} has assured it. In conclusion, S_t is a valid solution.

Proving S_t Extends A_t : By the loop invariant S_{t-1} extends all the decisions $\langle i_0, i_1, i_2, \dots, i_{t-1} \rangle$ made by algorithm before this iteration. The prover made sure that their modifications did not change any of these decisions and changed S_{t-1} 's decision j_t about the latest object to be consistent with what the algorithm did, i.e., i_{t+1} . Hence, S_t is consistent with both the earlier decisions made by algorithm and this most recent decision, i.e., extends A_t .

→ $\langle LI_t \rangle$: Because S_t witnesses the fact that there is at least one valid optimal solution that extends A_t , we know that the loop invariant has been maintained.

Exiting Loop ($\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): By the exit condition, the algorithm has made a decision about every object in the instance. Hence, the algorithm has a full solution A_{exit} . By the loop invariant, this extends an optimal valid solution S_{exit} . Hence, the algorithm must have an optimal valid solution.

33.3 Dynamic Programming

- III.26. (a) Decisions: Jeff talks of having a *little bird* to make this decision but there is no little bird. Hence, we try each bird answer. For each, we find the best solution consistent with this bird's answer. Then we take the best of these best.
- (b) Question Size: The size of a question is the number of different answers that she might give. "What is the optimal solution?" is big. "What is the first edge?" is small.
- (c) Friend: He is a recursive version of ourselves, that can be given a smaller instance of the same problem, i.e., meeting is the preconditions. We can trust him because strong induction tells us that we can.
- (d) $optS_{(i,k)}$: This is a best solution for our instance $subI[i]$ from amongst those consistent with the bird's k^{th} answer. We build it from the friend's solution and the bird's answer.
- (e) $optS[i]$: This is a best solution for our instance $subI[i]$. $optS[i] = Best_k optS_{(i,k)}$.
- (f) Subinstance: The set of subinstances might be $S = \{\langle a_1, a_2, \dots, a_i \rangle \mid i \in [1..n]\}$.
- (g) Table: The table indexed by the subinstances. It stores the bird's advice and the cost of the optimal solution. We do not store the entire optimal solution because it likely has size $\Theta(n)$ and passing it from friend to friend can add an extra factor of $\Theta(n)$ to the running time.
- (h) Recursion: It has a friend for each subinstance, but it does not recurse. When we ask a friend, he looks his answer up in the table that he had previously computed.
- (i) LCS: The longest common subsequence of $X = \langle w, f, q, f, e, w, f \rangle$ and $Y = \langle f, e, q, w, f, w \rangle$ is $Z = \langle f, e, w, f \rangle$.
- III.27. **Cross Array**
- (a) It does not work. Here is a counter example.
 $\{1, 2, 1, \dots, 2\}$.
 The greedy algorithm would move from 1 to 1 and then have to walk to the end.
 The optimal solution moves from 1, 2, 2.
 Here is a counterexample.
 $\{1, \dots, 2, 1, \dots, 2\}$.

The algorithm would move from 1 to 1 and then have to walk to the end.

The optimal solution moves from 1, 1, backward to 2, forward to 2.

- (b) A better algorithm has a node for every element and edges to nodes that it can get to in one step and does breadth first search (or Dijkstra) to find the shortest path from start to finish.

III.28. Integer Knapsack Problem

Algorithm Using Trusted Bird and Friend: Consider a particular instance, $\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$ to the knapsack problem. The little bird knows a solution S to it.

2) Question for Bird: I ask the little bird whether or not this optimal solution contains the n^{th} item from the store.

Possible Answers from Bird: There are $K = 2$ possible answers, Yes and No.

No

3) Constructing Subinstances: If the little bird tells me *not* to put the n^{th} item into my solution, then we simply delete this last item from consideration. This leaves us with the smaller instance, $\text{sub}I_{no} = \langle V, \langle v_1, p_1 \rangle, \dots, \langle v_{n-1}, p_{n-1} \rangle \rangle$, which we give to a friend. They give me an optimal solution optSubSol for it.

4) Constructing a Solution for My Instance: Trusting both the bird and the friend, my solution and its value/cost are same as the friend's.

Yes

3) Constructing Subinstances: If, on the other hand, the bird says to include the last item, then we can take this last item and put it into the knapsack first. This leaves a volume of $V - v_n$ in the knapsack. We determine how best to fill the rest of the knapsack with the remaining items by asking our friend to solve the smaller instance $\text{sub}I_{no} = \langle V - v_n, \langle v_1, p_1 \rangle, \dots, \langle v_{n-1}, p_{n-1} \rangle \rangle$. Here too, they give me an optimal solution optSubSol for it.

4) Constructing a Solution for My Instance: Trusting both the bird and the friend, my solution is the same as my friends, except I add the n^{th} item in the space of volume v_n left for it.

5) Costs of Solution: Having the extra item in it, the value of my solution is p_n more than the value of my friend's.

Yes but Wrong

3) Constructing Subinstances: If the last item does not fit into the knapsack because $v_n > V$ and the bird says to include it, then we politely tell them that they are wrong.

5) Costs of Solution: We set the value of this solution to be $-\infty$ to ensure that it is not selected as the best one.

Recursive Back Tracing Algorithm

6) Best of the Best: I can trust the friend because they are a recursive version of myself. Not actually having a little bird, I try all the bird's answers and take best of best.

7) Base Cases: If there are $n = 0$ items or the volume of the knapsack is $V = 0$, then the only solution is to put nothing in the knapsack for a value of zero.

Dynamic Programming Algorithm

1) The Set of Subinstances: The set of subinstances *subI*

(Describe). By tracing the recursive algorithm, we see that the set of subinstances ever given to me, my friends, their friends is $\{\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_i, p_i \rangle \rangle \mid V' \in [0 \dots V], i \in [0 \dots n]\}$. Note that the items considered are a contiguous prefix of the original items, indicating a polytime algorithm. However, in addition we are considering every possible smaller knapsack size.

Closed: Applying the suboperator to an arbitrary subinstance $\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_i, p_i \rangle \rangle$ from this set constructs subinstances $\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_{i-1}, p_{i-1} \rangle \rangle$ and $\langle V' - v_i, \langle v_1, p_1 \rangle, \dots, \langle v_{i-1}, p_{i-1} \rangle \rangle$, which are contained in the stated set of subinstances. (The second one is not when $V - v_n < 0$, but we don't actually recurse in this case.) Therefore, this set contains all subinstances generated by the recursive algorithm.

Generating: For some instances, these subinstances might not get called in the recursive program for every possible value of V' . However, as an exercise you could construct instances for which each such subinstance was called.

3) Construct a Table Indexed by Subinstances: The table indexed by the above set of subinstances will have a dimension for each of the parameters i and V' used to specify a particular subinstance. The tables will be $optCost[0 \dots V, 0 \dots n]$ and $birdAdvice[0 \dots V, 0 \dots n]$.

6) The Order in Which to Fill the Table: The friends solve their subinstances (and the table is filled) in an order so that nobody has to wait (from smaller to larger instances). This could be one row at a time with both i and V' increasing, one column at a time, or even diagonally.

8) Code

algorithm *Knapsack* ($\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$)

pre-cond: V is the volume of the knapsack. v_i and p_i are the volume and the price of the i^{th} objects in a store.

post-cond: *optSol* is a way to fill the knapsack with the greatest possible total price. *optCost* is its price.

begin

```

% Table:  $subI[V', i]$  denotes the subinstance of optimally filling a
knapsack with volume  $V'$  with the first  $i$  objects.
 $optSol[V', i]$  would store an optimal solution for it, but
it is too big. Hence, we store only the bird's advice
 $birdAdvice[V', i]$  given for the subinstance and the cost
 $optCost[V', i]$  of an optimal solution.
table[0...V, 0...n] birdAdvice, optCost

% Base Cases: The base cases are when the number of objects is zero.
For each, the solution is the empty knapsack with cost zero.
(If the knapsack has zero volume, then nothing will fit in it until
there are zero items.)
loop  $V' = 0 \dots V$ 
    %  $optSol[V', 0] = \emptyset$ 
     $optCost[V', 0] = 0$ 
     $birdAdvice[V', 0] = \emptyset$ 
end loop

% General Cases: Loop over subinstances in the table.
loop  $i = 1$  to  $n$ 
    loop  $V' = 0$  to  $V$ 
        % Solve instance  $subI[V', i]$  and fill in the table at
        index  $\langle V', i \rangle$ .
        % The bird and Friend Alg: The bird tells us either (1)
        exclude the  $i^{th}$  item from the knapsack or (2) include
        it. Either way, we remove this last object, but in case
        (2) we decrease the size of the knapsack by the space
        needed for this item. Then we ask the friend for an
        optimal packing of the resulting subinstance. They
        give us (1)  $optSol[V', i - 1]$  or (2)  $optSol[V' - v_i, i - 1]$ 
        which they had stored in the table. If the bird had said
        we were to include the  $i^{th}$  item, then we add this item
        to the friend's solution. Denote this resulting solution
        by  $optSol_{\langle \langle V', i \rangle, k \rangle}$ . It is a best packing for our instance
         $subI[V', i]$  from among those consistent with the bird's
         $k^{th}$  answer.
        % Try each possible bird answers.
        % cases  $k = 1, 2$  where 1 = exclude 2 = include
        %  $optSol_{\langle \langle V', i \rangle, 1 \rangle} = optSol[V', i - 1]$ 
         $optCost_{\langle \langle V', i \rangle, 1 \rangle} = optCost[V', i - 1]$ 
        if  $(V' - v_i \geq 0)$  then
            %  $optSol_{\langle \langle V', i \rangle, 2 \rangle} = optSol[V' - v_i, i - 1] \cup i$ 
             $optCost_{\langle \langle V', i \rangle, 2 \rangle} = optCost[V' - v_i, i - 1] + p_i$ 
        else

```

```

                                % Bird was wrong
                                %  $optSol_{\langle V', i \rangle, 2} = ?$ 
                                 $optCost_{\langle V', i \rangle, 2} = -\infty$ 
                                end if
                                % end cases
                                % Having the best,  $optSol_{\langle V', i \rangle, k}$ , for each bird's
                                answer  $k$ ,
                                we keep the best of these best.
                                 $k_{max} = \text{"a } k \text{ that maximizes } optCost_{\langle V', i \rangle, k}"$ 
                                %  $optSol[V', i] = optSol_{\langle V', i \rangle, k_{max}}$ 
                                 $optCost[V', i] = optCost_{\langle V', i \rangle, k_{max}}$ 
                                 $birdAdvice[V', i] = k_{max}$ 
                                end for
                                end for
                                 $optSol = KnapsackWithAdvice(\langle V, \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle,$ 
                                 $birdAdvice)$ 
                                return  $\langle optSol, optCost[V, n] \rangle$ 
                                end algorithm

```

8') Constructing the Solution: We would run the recursive algorithm with the bird's advice to find the solution to our instance. We exclude this step from our answer.

9) Running Time: The number of subinstances is $\Theta(V \cdot n)$ and the bird chooses between two options: to include or not to include the object. Hence, the running and the space requirements are both $\Theta(V \cdot n)$. However, this running time should be expressed as a function of input size. The number of bits needed to represent the instance $\langle V', \langle v_1, p_1 \rangle, \dots, \langle v_n, p_n \rangle \rangle$ is $N = |V| + n \cdot (|v| + |p|)$, where $|V|$, $|v|$, and $|p|$ are the numbers of bits needed to represent V , v_i , and p_i . Expressed in these terms, the running time is $T(|instance|) = \Theta(nV) = \Theta(n2^{|V|})$. This is quicker than the brute force algorithm because its running time is polynomial in the number of items n . In the worst case, however, V is large and the time can be exponential in the number of bits N , i.e., if $|V| = \Theta(N)$, then $T = \Theta(2^N)$. In fact, the knapsack problem is one of the classic NP complete problems, which means that it is generally believed that not polynomial time algorithm exists for it.

III.29. Stock Market Prices

1) Specifications: (See question.) An input instance consists of $I = \langle T, S, Price \rangle$, where T is the day that I must sell, S is the set of stocks, and $Price$ gives the prices. A solution is a sequence of pairs $\langle t_i, s_i \rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1} . (Here $i \geq 1$, $t_1 = 1$ and $t_{last+1} = T$.) The value of this solution is $\Pi_i \left[(1 - 0.03) \cdot \frac{Price(t_{i+1}, s_i)}{Price(t_i, s_i)} \right]$.

Algorithm Using Trusted Bird and Friend: I have my instance $\langle T, S, Price \rangle$. The little bird knows a solution to it.

2) Question for Bird: I ask the little bird what is the last object $\langle t_k, s_k \rangle$ in the solution, namely, what stock $s_k \in S$ I should buy last and one what day $t_k \in [1 \dots T - 1]$ I should buy it. Note I will sell this stock on day T .

2') Possible Answers from Bird: There are $T \cdot |S|$ different answers $\langle t_k, s_k \rangle$ that they might give.

3) Constructing Subinstances: Given the bird wants me to buy on day t_k and I sell and buy on the same day, I need my friend to tell me what and when to buy and sell so as to sell on day t_k . Therefore, I give my friend the subinstance $subI = \langle t_k, S, Price \rangle$. Note, I do not need to change the set of stocks considered and even though they won't use the entire table $Price$, we don't need to change it either. They give me an optimal solution $optSubSol$ for it.

4) Constructing a Solution for My Instance: I produce an optimal solution $optSol$ for my instance I from the bird's answer k and the friend's solution $optSubSol$ simply by tacking the last object $\langle t_k, s_k \rangle$ on to the end of the friend's solution. This means that my friend and I buy and sell the same stocks on the same days, we both sell on day t_k , then I continue on to buy stock s_k on this same day t_k , later to sell it on day T .

5) Costs of Solution: If $optSubCost$ is the cost of my friend's optimal solution $optSubSol$ for their instance $subI$, then my cost $optCost$ to my solution $optSol$ is $optSubCost \times \left[(1 - 0.03) \cdot \frac{Price(T, s_k)}{Price(t_k, s_k)} \right]$.

Recursive Back Tracing Algorithm:

6) Best of the Best: I can trust the friend because they are a recursive version of myself. Not actually having a little bird, I try all its answers and take best of best.

7) Base Cases: The base case instance is when we have to sell on day one. Its solution is to never buy or sell anything. Its value is one because we still have the dollar that we started with.

Dynamic Programming Algorithm

1) The Set of Subinstances: We determine the set of subinstances $subI = \langle t, S, Price \rangle$ ever given to me, my friends, their friends. For each day $t \in [1 \dots T]$, there is a subinstance that asks what to buy and sell so that on day t you sell your last stock and maximize the amount of money you have on that day. There are T such subinstances. Note that this set is closed under this "sub"-operator and all of these subinstances are needed.

3) Construct a Table Indexed by Subinstances: The table is simply indexed by $t \in [1 \dots T]$. We don't actually store the solution $optS[t]$ for the subinstance $subI[t]$, but $optCost[t]$ is the cost of this solution and $birdAdvice[t]$ stores the birds advice given on this subinstance.

6) The Order in Which to Fill the Table: The friends solve their subinstances (and the table is filled) in an order so that nobody has to wait (from smaller to larger instances). This is simply for $t = 1, \dots, T$.

8) Code

algorithm *Stocks* ($T, S, Price$)

⟨pre-cond⟩: T is the day that I must sell, S is the set of stocks, and $Price$ gives the prices.

⟨post-cond⟩: $optSol$ is an optimal valid schedule. It is a sequence of pairs $\langle t_i, s_i \rangle$, meaning that stock s_i is bought on day t_i and sold on day t_{i+1} . $optCost$ is its cost.

begin

% Table: $subI[t]$ denotes the subinstance of finding an optimal schedule ending on day t .

$optSol[t]$ would store an optimal solution for it, but it is too big. Hence, we store only the bird's advice $birdAdvice[t]$ given for the subinstance and the cost $optCost[t]$ of an optimal solution.

$table[1 \dots T] \ optCost, birdAdvice$

% Base Case: The only base case is for the optimal set ending on day $t = 1$.

It's solution consists of the empty set with value 1.

% $optSol[1] = \emptyset$

% $optCost[1] = 1$

% $birdAdvice[1] = \emptyset$

% General Cases: Loop over subinstances in the table.

for $t = 2$ to T

% Solve instance $subI[t]$.

% Try each possible bird answer.

for each $t_k \in [1 \dots t - 1]$

for each $s_k \in S$

% The bird and friend alg. I want to finish on day t . I ask the bird what stock $s_i \in S$ I should buy last and what day $t_i \in [1 \dots t - 1]$ I should buy it. The bird answers $\langle t_k, s_k \rangle$. I ask my friend to solve the subinstance that ends on day t_k . I produce an optimal solution $optSol$ for my instance $subI[t]$ from the bird's answer k and the friend's solution $optSubSol$ simply by tacking the last object $\langle t_k, s_k \rangle$ on to the end of the friend's solution.

This means that my friend and I buy and sell the same stocks on the same days except after we both sell everything on day t_k , I buy stock s_k later to sell it on day T . Denote this resulting solution by $optSol_{\langle t, \langle t_k, s_k \rangle \rangle}$. It is a best solution for our instance $subI[t]$ from among those consistent with the bird's $\langle t_k, s_k \rangle^{th}$ answer.

```

%  $optSol_{\langle t, \langle t_k, s_k \rangle \rangle} = optSol[t_k] + \langle t_k, s_k \rangle$ 
 $optCost_{\langle t, \langle t_k, s_k \rangle \rangle} =$ 
 $optCost[t_k] \cdot \left[ (1 - 0.03) \cdot \frac{Price(t, s_k)}{Price(t_k, s_k)} \right]$ 
end for
% Having the best,  $optSol_{\langle t, \langle t_k, s_k \rangle \rangle}$ , for each bird's answer
 $\langle t_k, s_k \rangle$ , we keep the best of these best.
 $\langle t_{max}, s_{max} \rangle = \text{"a } \langle t_k, s_k \rangle \text{ that maximizes } optCost_{\langle t, \langle t_k, s_k \rangle \rangle}"$ 
%  $optSol[t] = optSol_{\langle t, \langle t_{max}, s_{max} \rangle \rangle}$ 
 $optCost[t] = optCost_{\langle t, \langle t_{max}, s_{max} \rangle \rangle}$ 
 $birdAdvice[t] = \langle t_{max}, s_{max} \rangle$ 
end for
 $optSol = SchedulingWithAdvice(T, S, Price, birdAdvice)$ 
return  $\langle optSol, optCost[T] \rangle$ 
end algorithm

```

8') Constructing the Solution: We would run the recursive algorithm with the bird's advice to find the solution to our instance. We exclude this step from our answer.

9) Running Time

The number of subinstances in the table is T .

The number of bird answers is $T \cdot |S|$.

The running time is the product of these $O(T^2 \cdot |S|)$.

33.4 Reductions and NP-Completeness

III.42. Short Answers

- A computational problem P is a function $P(I)$ from each possible finite input I that meets the stated preconditions to the corresponding required output that meets the stated postconditions. (Sometimes there is more than one acceptable output.) A simple problem, given a set of numbers as input, is to output those numbers in sorted order. A much harder problem, given a video taken from outside of a car, is to determine whether the Google driver should turn the car in order to avoid a child in the road.
- An algorithm A is a finite sequence of instructions (perhaps a Turing Machine, but more likely written in a language like C or Java) that can mechanically be followed. Starting with an input I , it either halts with an

answer denoted by $A(I)$ or runs forever denoted by $A(I) = \infty$. We say that algorithm A solves problem P if for every input I meeting the precondition, we have that A halts with the correct answer when given I , namely that $A(I) = P(I)$.

- (c) Informally, the time complexity $Time_A(n)$ of an algorithm A is a function from the amount of work needed for an adversary to present an input I to the amount needed for A to complete its computation on this input. Let $Time(A, I)$ denote the “time” until algorithm A halts on input I . This could be measured in seconds, computer cycles, or lines of code executed because these are the same within a multiplicative constant. The “size” of an instance I , denoted $n = |I|$ formally is the number of bits to describe it, but as well could be the number of ASCII characters or the area of the paper needed because these are all equivalent within a multiplicative constant. (If the input is an integer, do not, however, use its value as its size, because this is exponentially larger.) The time complexity of algorithm A is defined to be the time needed for the worst case input of size n , namely, $Time_A(n) = \max_{|I|=n} Time(A, I)$.
- (d) The time complexity $Time_P(n)$ of a computational problem P is defined to be the time complexity $Time_A(n)$ of the fastest algorithm A that solves P . An upper bound $Time_P(n) \leq Time_{upper}(n)$ is proved by providing an algorithm A and proving that it solves P in the stated time. A lower bound $Time_{lower}(n) \leq Time_P(n)$ is much harder to prove because one must prove that for every algorithm A , there is an input I on which it either runs too slowly or gives the wrong answer.
- (e) A running time is said to be constant, denoted $\Theta(1)$, if it is bounded between some two real positive numbers c_1 and c_2 for all sufficiently large values of n . It is said to be linear, denoted $\Theta(n)$, if bounded between $c_1 n$ and $c_2 n$; polynomial, denoted $n^{\Theta(1)}$, if between n^{c_1} and n^{c_2} ; and exponential, denoted $2^{\Theta(n)}$, if between $2^{c_1 n}$ and $2^{c_2 n}$. It is important practically because if $c_2 = 2$ seconds and $n = 150$, then the times are 2 seconds for constant, 5 minutes for linear, 6 hours for polynomial, and 10^{82} years for the exponential. If the input is the size of DNA, then the algorithm really needs to be linear. Otherwise, we say that an algorithm is practical if it runs in polynomial time.
- (f) A computational problem takes as input some instance I . Each such instance has a huge (likely exponential) set of *possible solutions* S . Each solution has well-specified criteria for being *valid* and has a cost $cost(I, S)$. Given instance I , the goal is to find a valid solution of optimal value, i.e., maximum or minimum.
- (g) For each instance, the answer is either *yes* or *no*. It is a *yes* instance if and only if it has a valid solution (that is sufficiently good). For example, “Does network G have valid flow with value at least 17?” This solution is sometimes referred to as a *witness* or even as a *proof* because it witnesses/proves that the instance is a *yes* instance.

- (h) There is a *poly-time* algorithm $Valid(I, S)$ that given an instance I and a solution S , tests whether S is a valid solution for I . A key point is that the algorithm $Valid(I, S)$ runs in polynomial time in the size of the instance I . Formally, we say that $P \in NP$ iff $\exists$ algorithm $Valid$ and constant c such that $\forall I, I \in P$ iff $\exists S \text{ Valid}(I, S) = \text{yes}$ and $\forall I, S, \text{Time}(\text{Valid}(I, S)) \leq |I|^c$.
- (i) The instance is a yes instance iff it has a solution that is easy to check. The fairy godmother can simply give you such a valid solution S . You give this to your boss. Your boss then checks its validity with $Valid(I, S)$. Note your boss can do this because it runs in polynomial time in the size of their instance I .

If it is a *no* instance, then (unless the problem is also in *co-NP*), it does not have a solution/witness to help you and/or your boss.

- (j) Yes. The algorithm $Valid(I, S)$ must run in polynomial time in the size of the instance I . For $Valid$ to even look at the solution S , its size must also be polynomial time in the size of the instance I , i.e., there is a constant c such that for all instances I , the size of the solution S is at most $|I|^c$ bits long. Hence, the number of such solutions can be at most $2^{|I|^c}$.
- (k) The algorithm checks all possible solutions S . This takes at most $2^{|I|^c}$ time because this is the number of such solutions.
- (l) The question is whether every problem in NP can be solved in polynomial time. The general belief is no, that many of them take exponential time.
- (m) A problem P is NP-Hard iff it is as hard as any problem in NP , i.e., $\forall P' \in NP, P' \leq P$. Cook did this for SAT . Hence, a problem P is NP-Hard iff $SAT \leq P$. Being NP-hard means that if there was a polynomial algorithm for P , then there would be one for every problem in NP and hence $P = NP$. If, as general believed, $P \neq NP$, then such problems do not have polynomial time algorithms. A problem P is NP-Complete iff it is NP-Hard and is in NP .
- (n) If your boss gives you a brand-new problem and you are able to prove that it is NP-Complete, then you know that you should not expect to find a poly-time algorithm that works for every instance. You are better off trying to find some heuristic that gives an OK answer for most instances.
- (o) For some problems, there are poly-time algorithms that are guaranteed to give a solutions that are approximately optimal. For example, dynamic programming can find a solution for the knapsack problem whose value is within a multiplicative factor of $1 + \epsilon$ of the optimal value, while the time required is only $\Theta(n^2/\epsilon)$.
Alternatively, there are many heuristics for the problems that sometimes gives a good enough answers fast enough for large instances of the problem. For example, Davis–Putnam recursive backtracking can often solve SAT problems quickly.
- (p) It is hard to prove that P_2 is hard. It is easier to prove that P_1 is “easier” P_2 . Write an algorithm for P_1 using an algorithm for P_2 as a subroutine. We

sometime refer to the algorithm for P_2 as an *oracle* (like a burning bush on top of a mountain or the one at Delphi).

- (q) **algorithm** $Alg_{alg}(I_{alg})$
 $\langle \text{pre-cond} \rangle$: I_{alg} is an instance of P_{alg} .
 $\langle \text{post-cond} \rangle$: Determine whether I_{alg} has a solution S_{alg} and if so returns it.
begin
 $I_{oracle} = InstanceMap(I_{alg})$
 $\langle ans_{oracle}, S_{oracle} \rangle = Alg_{oracle}(I_{oracle})$
if($ans_{oracle} = Yes$) then
 $ans_{alg} = Yes$
 $S_{alg} = SolutionMap(S_{oracle})$
else
 $ans_{alg} = No$
 $S_{alg} = nil$
end if
return($\langle ans_{alg}, S_{alg} \rangle$)
end algorithm
- (r) We have to prove that Alg_{alg} works if Alg_{oracle} works. For this we prove
- 1 Given an instance I_{alg} , $InstanceMap(I_{alg})$ maps this to a valid instance I_{oracle} .
 - 2 If S_{oracle} is a solution for I_{oracle} then $S_{alg} = SolutionMap(S_{oracle})$ is a solution for I_{alg} whose cost is just as good.
 - 3 If S_{alg} is a solution for I_{alg} then $S_{oracle} = ReverseSolutionMap(S_{alg})$ is a solution for I_{oracle} whose cost is just as good.
- (s) 1 Designing new algorithms.
2 Arguing that a problem is hard or easy.
3 Identifying equivalence classes of problems.

III.43. Fun Faulty Reductions

- (a) NP: It can easily be solved in poly-time and hence is automatically in NP. If a fairy godmother wanted to help she could tell you where the block is, but this is not necessary. This is not very exciting.
- (b) NP-Complete: It can be done in poly-time. Hence, if it was NP-complete then all NP problems could be done in poly-time. This is unlikely.
- (c) No instance Steps 6 and 7: No problem. The instance I_{Block} has no valid solutions because it is a no instance. Because there are none, it is easy to map every valid solution of it to a valid solution of I_{3-SAT} .
- (d) Yes instance Steps 8 and 9: No problem. The instance I_{Block} has many valid solutions. One of them is the substring $s_1, s_2, \dots, s_5$. Any valid solutions of I_{3-SAT} (if there are any) can be mapped to it.
- (e) Yes instance Steps 5, 6, and 7: He did 5 fine. 6 and 7 he did in one paragraph, which was a mistake. He should have fully defined the assignment to the variables in Step 6, before trying to prove it valid in Step 7. As is, he may have set some variable x to be true in order to satisfy one clause but later set the same variable to false in order to satisfy another clause.

III.44. Reduction between Problem and a Restriction of It

- (a) $P_{\text{general}} \leq_{\text{poly}} P_{\text{restricted}}$ vs. $P_{\text{restricted}} \leq_{\text{poly}} P_{\text{general}}$: Clearly a restricted problem is easier because there are few instances that might be the worst case. $P_{\text{restricted}} \leq_{\text{poly}} P_{\text{general}}$ is always true. $P_{\text{general}} \leq_{\text{poly}} P_{\text{restricted}}$ will not be true if the instances considered by the restricted problem are the easy ones and not the worst case ones.
- (b) $\text{SAT} \leq_{\text{poly}} 3\text{-SAT}$ and $3\text{-SAT} \leq_{\text{poly}} \text{SAT}$: As said, this is true because the first is a restriction of the second. Surprisingly, $\text{SAT} \leq_{\text{poly}} 3\text{-SAT}$ is also true. One has to show that there are sufficiently difficult instances that have 3 literals per clause. The reduction shows that being able to solve the 3 literal case is enough to be able to solve all of NP .
- (c) $\text{Clique} \leq_{\text{poly}} 3\text{-Clique}$ and $3\text{-Clique} \leq_{\text{poly}} \text{Clique}$: $3\text{-Clique} \leq_{\text{poly}} \text{Clique}$ is trivially true. $\text{Clique} \leq_{\text{poly}} 3\text{-Clique}$ is false (unless $P = NP$), because Clique is NP -complete and 3-Clique can be solved in $O(n^3)$ time.

III.45. Card Puzzle

- (a) A card in a solution of the card puzzle is flipped
iff the corresponding variable in the corresponding solution of clause-SAT is negated.
- (b) For each j and v ,
 - card v covers the j^{th} X ,
 - iff variable v satisfies the j^{th} clause.
 Hence, for each j ,
 - the j^{th} X is covered,
 - iff the j^{th} clause is satisfied.
 Hence,
 - the solution of the card puzzle satisfies the requirement,
 - iff the corresponding solution of clause-SAT does.
 Hence,
 - $\exists$ a satisfying solution for the instance of card puzzle,
 - iff $\exists$ a satisfying solution for corresponding instance of clause-SAT.
 Hence,
 - the instance of card puzzle requires a Yes,
 - iff the corresponding instance of clause-SAT requires a Yes.
- (c) Given cards I and whether they are flipped S , it is easy to tell if all the X in the box are covered.
- (d) Step 4: Given an oracle for the Card Puzzle, we give an algorithm for Card-SAT. Given a set of clauses construct the corresponding cards as described above, give these to the oracle, and answer the same.
Step 5: Clauses satisfiable iff Card Puzzle satisfiable iff Oracle says yes iff Alg. says yes. See question (b).

Part IV

Additional Topics

34 Existential and Universal Quantifiers

Existential and universal quantifiers provide an extremely useful language for making formal statements. You must understand them. A game between a prover and a verifier is a level of abstraction within which it is easy to understand and prove such statements.

The Loves Example: Suppose the relation (predicate) $Loves(p_1, p_2)$ means that person p_1 loves person p_2 . Then we have:

Expression	Meaning
$\exists p_2 Loves(Sam, p_2)$	“Sam loves somebody.”
$\forall p_2 Loves(Sam, p_2)$	“Sam loves everybody.”
$\exists p_1 \forall p_2 Loves(p_1, p_2)$	“Somebody loves everybody.”
$\forall p_1 \exists p_2 Loves(p_1, p_2)$	“Everybody loves somebody.”
$\exists p_2 \forall p_1 Loves(p_1, p_2)$	“There is one person who is loved by everybody.”
$\exists p_1 \exists p_2 (Loves(p_1, p_2) \text{ and } \neg Loves(p_2, p_1))$	“Somebody loves in vain.”

Definition of Relation: A relation like $Loves(p_1, p_2)$ states for every pair of objects (say, $p_1 = Sam$ and $p_2 = Mary$) that the relation either holds between them or does not. Though we will use the word “relation,” $Loves(p_1, p_2)$ is also considered to be a *predicate*. The difference is that a predicate takes only one argument and hence focuses on whether the property is *true* or *false* about the given tuple $\langle p_1, p_2 \rangle = \langle Sam, Mary \rangle$.

Representations: Relations (predicates) can be represented in a number of ways.

Functions: A relation can be viewed as a function mapping tuples of objects either to *true* or to *false*, for example, $Loves: \{p_1 | p_1 \text{ is a person}\} \times \{p_2 | p_2 \text{ is a person}\} \Rightarrow \{true, false\}$.

Set of Tuples: Alternatively, it can be viewed as a set containing the tuples for which it is true, for example, $Loves = \{\langle Sam, Mary \rangle, \langle Sam, Ann \rangle, \langle Bob, Ann \rangle, \dots\}$. $\langle Sam, Mary \rangle \in Loves$ iff $Loves(Sam, Mary)$ is true.

Directed Graph Representation: If the relation only has two arguments, it can be represented by a directed graph. The nodes consist of the objects in the domain. We place a directed edge $\langle p_1, p_2 \rangle$ between pairs for which the relation is true. If the domains for the first and second objects are disjoint, then the

graph is bipartite. Of course, the *Loves* relation could be defined to include *Loves*(*Bob*,*Bob*). See Figure 34.1.

Quantifiers: You will be using the following quantifiers and properties.

The Existential Quantifier: The quantifier $\exists$ means that there is at least one object in the domain with the property. This quantifier relates to the Boolean operator *OR*. For example, $\exists p_1 \text{ Loves}(\text{Sam}, p_1) = [\text{Loves}(\text{Sam}, \text{Mary}) \text{ OR } \text{Loves}(\text{Sam}, \text{Ann}) \text{ OR } \text{Loves}(\text{Sam}, \text{Bob}) \text{ OR } \dots]$.

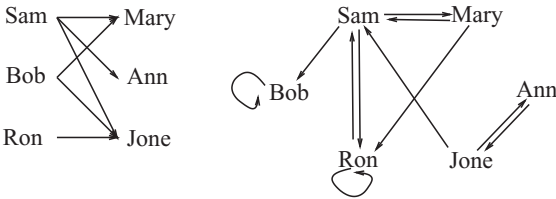


Fig. 34.1 A directed graph representation of the *Loves* relation.

The Universal Quantifier: The quantifier $\forall$ means that all of the objects in the domain have the property. It relates to the Boolean operator *AND*. For example, $\forall p_1 \text{ Loves}(\text{Sam}, p_1) = [\text{Loves}(\text{Sam}, \text{Mary}) \text{ AND } \text{Loves}(\text{Sam}, \text{Ann}) \text{ AND } \text{Loves}(\text{Sam}, \text{Bob}) \text{ AND } \dots]$.

Combining Quantifiers: Quantifiers can be combined. The order of operations is such that $\forall p_1 \exists p_2 \text{ Loves}(p_1, p_2)$ is understood to be bracketed as $\forall p_1 [\exists p_2 \text{ Loves}(p_1, p_2)]$, i.e., “Every person has the property ‘he loves some other person.’” It relates to the following Boolean formula:

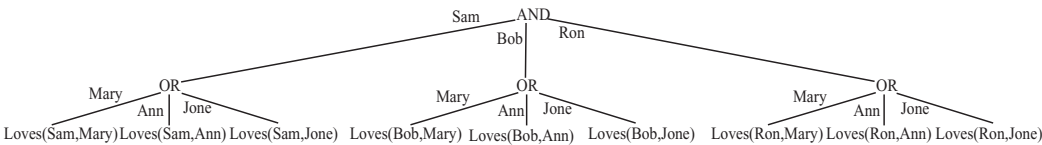


Fig. 34.2 A Boolean formula representing that $\forall p_1 \exists p_2 \text{ Loves}(p_1, p_2)$.

Order of Quantifiers: The order of the quantifiers matters. For example, if *b* is the class of boys and *g* is the class of girls, $\forall b \exists g \text{ Loves}(b, g)$ and $\exists g \forall b \text{ Loves}(b, g)$ mean different things. The second one states that the same girl is loved by every boy. For it to be true, there needs to be a Marilyn Monroe sort of girl that all the boys love. The first statement says that every boy loves some girl. A Marilyn Monroe sort of girl will make this statement true. However, it is also true in a monogamous situation in which every boy loves a different girl. Hence, the first statement can be true in more different ways than the second one. In fact, the second statement implies the first one, but not vice versa.

Definition of Free and Bound Variables: The statement $\exists p_2 \text{ Loves}(\text{Sam}, p_2)$ means “Sam loves someone.” This is a statement about Sam. Similarly, the statement $\exists p_2 \text{ Loves}(p_1, p_2)$ means “ p_1 loves someone.” This is a statement about person p_1 .

Whether the statement is true depends on who p_1 is referring to. The statement is *not* about p_2 . The variable p_2 is used as a local variable (similar to $\text{for}(i = 1; i \leq 10; i++)$) to express “someone.” It could be a brother or a friend or a dog. In this expression, we say that the variable p_2 is *bound*, while p_1 is *free*, because p_2 has a quantifier and p_1 does not.

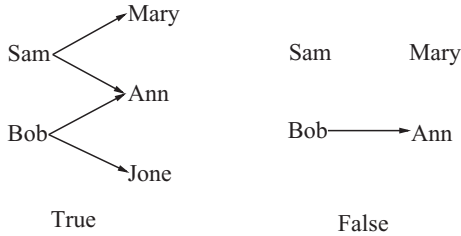


Fig. 34.3 $\forall g \exists b \exists p (Loves(b, g) \text{ and } Loves(b, p) \text{ and } g \neq p)$. On the left is an example of a situation in which the statement is true, and on the right is one in which it is false.

Defining Other Relations: You can define other relations by giving an expression with free variables. For example, you can define the unary relation $LovesSomeone(p_1) \equiv \exists p_2 Loves(p_1, p_2)$.

Building Expressions: Suppose you wanted to state that *every girl has been cheated on*, using the *Loves* relation. It may be helpful to break the problem into three steps.

Step 1. Assuming Other Relations: Suppose you have the relation $Cheats(Sam, Mary)$, indicating that Sam cheats on Mary. How would you express the statement that every girl has been cheated on? The advantage of using this function is that we can focus on this one part of the statement. We are not claiming that every boy cheats. One boy may have broken every girl’s heart.

Given this, the answer is $\forall g \exists b Cheats(b, g)$.

Step 2. Constructing the Other Predicate: Here we do not have a *Cheats* function. Hence, we must construct a sentence from the *loves* function stating that Sam cheats on Mary.

Clearly, there must be someone else involved besides Mary, so let’s start with $\exists p$. Now, in order for cheating to occur, who needs to love whom? (For simplicity’s sake, let’s assume that cheating means loving more than one person at the same time.) Certainly, Sam must love p . He must also love Mary. If he did not love her, then he would not be cheating on her. Must Mary love Sam? No. If Sam tells Mary he loves her dearly and then a moment later he tells Sue he loves *her* dearly, then he has cheated on Mary regardless of how Mary feels about him. Therefore, Mary does not have to love Sam. In conclusion, we might define $Cheats(Sam, Mary) \equiv \exists p (Loves(Sam, Mary) \text{ and } Loves(Sam, p))$. However, we have made a mistake here. In our example, the other person and Mary cannot be the same person. Hence, we must define the relation as $Cheats(Sam, Mary) \equiv \exists p (Loves(Sam, Mary) \text{ and } Loves(Sam, p) \text{ and } p \neq Mary)$.

Step 3. Combining the Parts: Combining the two relations together gives you $\forall g \exists b \exists p (\text{Loves}(b, g) \text{ and } \text{Loves}(b, p) \text{ and } p \neq g)$. This statement expresses that every girl has been cheated on. See [Figure 34.3](#).

The Domain of a Variable: Whenever you state $\exists g$ or $\forall g$, there must be an understood set of values that the variable g might take on. This set is called the *domain* of the variable. It may be explicitly given or implied, but it must be understood. Here the domain is “the” set of girls. You must make clear whether this means all girls in the room, all the girls currently in the world, or all girls that have ever existed. For example,

$$\forall x \exists y x \times y = 1$$

states that every value has a reciprocal. It is certainly not true of the domain of integers, because two does not have an integer reciprocal. It seems to be true of the domain of reals. Be careful, however: zero does not have a reciprocal. It would be better to write

$$\forall x \neq 0, \exists y x \times y = 1,$$

or equivalently,

$$\forall x \exists y (x \times y = 1 \text{ OR } x = 0).$$

The Negation of a Statement: The negation of a statement is formed by putting a negation sign on the left-hand side. (Brackets sometimes help.) A negated statement, however, is best understood by moving the negation as deep (as far right) into the statement as possible. This is done as follows.

Negating AND and OR: A negation on the outside of an AND or an OR statement can be moved deeper into the statement using De Morgan’s law. Recall that the AND is replaced by an OR and the OR is replaced with an AND.

$\neg(\text{Loves}(S, M) \text{ AND } \text{Loves}(S, A))$ iff $\neg\text{Loves}(S, M) \text{ OR } \neg\text{Loves}(S, A)$: The negation of “Sam loves Mary and Ann” is “Either Sam does not love Mary or he does not love Ann.” He can love one of the girls, but not both.

A common mistake is to make the negation $\neg\text{Loves}(Sam, Mary) \text{ AND } \neg\text{Loves}(Sam, Ann)$. However, this says that Sam loves neither Mary nor Ann.

$\neg(\text{Loves}(S, M) \text{ OR } \text{Loves}(S, A))$ iff $\neg\text{Loves}(S, M) \text{ AND } \neg\text{Loves}(S, A)$: The negation of “Sam either loves Mary or he loves Ann” is “Sam does not love Mary and he does not love Ann.”

Negating Quantifiers: Similarly, a negation can be moved past one or more quantifiers either to the right or to the left. However, you must then change these quantifiers from existential to universal and vice versa. Suppose d is the set of dogs. Then we have:

$\neg(\exists d \text{ Loves}(Sam, d))$ iff $\forall d \neg\text{Loves}(Sam, d)$: The negation of “There is a dog that Sam loves” is “There is no dog that Sam loves” or “All dogs are unloved by Sam.” A common mistake is to state the negation as

$\exists d \neg \text{Loves}(\text{Sam}, d)$. However, this says that “There is a dog that is not loved by Sam.”

$\neg(\forall d \text{Loves}(\text{Sam}, d))$ iff $\exists d \neg \text{Loves}(\text{Sam}, d)$: The negation of “Sam loves every dog” is “There is a dog that Sam does not love.”

$\neg(\exists b \forall d \text{Loves}(b, d))$ iff $\forall b(\forall d \neg \text{Loves}(b, d))$ iff $\forall b \exists d \neg \text{Loves}(b, d)$: The negation of “There is a boy who loves every dog” is “There are no boys who love every dog” or “For every boy, it is not the case that he loves every dog” or “For every boy, there is some dog that he does not love.”

$\neg(\exists d_1 \exists d_2 \text{Loves}(\text{Sam}, d_1) \text{ AND } \text{Loves}(\text{Sam}, d_2) \text{ AND } d_1 \neq d_2)$ iff $\forall d_1 \forall d_2 \neg(\text{Loves}(\text{Sam}, d_1) \text{ AND } \text{Loves}(\text{Sam}, d_2) \text{ AND } d_1 \neq d_2)$ iff $\forall d_1 \forall d_2 \neg(\text{Loves}(\text{Sam}, d_1) \text{ OR } \neg \text{Loves}(\text{Sam}, d_2) \text{ OR } d_1 = d_2)$: The negation of “There are two (distinct) dogs that Sam loves” is “Given any pair of (distinct) dogs, Sam does not love both” or “Given any pair of dogs, either Sam does not love the first or he does not love the second, or you gave me the same dog twice.”

The Domain Does Not Change: The negation of $\exists x \geq 5, x + 2 = 4$ is $\forall x \geq 5, x + 2 \neq 4$. The negation does *not* begin $\exists x < 5 \dots$. Both the statement and its negation are about numbers greater than 5. Is there or is there not a number with the property such that $x + 2 = 4$?

Proving a Statement True: There are a number of seemingly different techniques for proving that an existential or universal statement is true. The core of all these techniques, however, is the same. Personally, I like to view the proof as a strategy for winning a game against an adversary.

Techniques for Proving $\exists d \text{Loves}(\text{Sam}, d)$

Proof by Example or by Construction: The classic technique to prove that something with a given property exists is by example. You either directly provide an example, or you describe how to construct such an object. Then you prove that your example has the property. For the above statement, the proof would state, “Let d be Fido” and then would prove that Sam loves Fido.

Proof by Adversarial Game: Suppose you claim to an adversary that there is a dog that Sam loves. What will the adversary say? Clearly, they challenge, “Oh, yeah? What dog?” You then meet the challenge by producing a specific dog d and proving that $\text{Loves}(\text{Sam}, d)$, that is, that Sam loves d . The statement is true if you have a strategy guaranteed to beat any adversary in this game.

- If the statement is true, then you can produce some dog d .
- If the statement is false, then you will not be able to.

Techniques for Proving $\forall d \text{ Loves}(\text{Sam}, d)$

Proof by Example Does Not Work: Proving that Sam loves Fido is interesting, but it does not prove that he loves all dogs.

Proof by Case Analysis: The laborious way of proving that Sam loves all dogs is to consider each dog, one at a time, and prove that Sam loves it.

This method is impossible if the domain of dogs is infinite.

Proof by Arbitrary Example: The classic technique to prove that every object from some domain has a given property is to let some symbol represent an arbitrary object from the domain and then to prove that that object has the property. Here the proof would begin, “Let d be any arbitrary dog.” Because we don’t actually know which dog d is, we must either (1) prove $\text{Loves}(\text{Sam}, d)$ simply from the properties that d has *because* d is a dog, or (2) go back to doing a case analysis, considering each dog d separately.

Proof by Adversarial Game: Suppose you claim to an adversary that Sam loves every dog. What will the adversary say? Clearly they challenge, “Oh, yeah? What about Fido?” You meet the challenge by proving that Sam loves Fido. In other words, the adversary provides a dog d' . You win if you can prove that $\text{Loves}(\text{Sam}, d')$.

The only difference between this game and the one for existential quantifiers is who provides the example. Interestingly, the game only has one round. The adversary is only given one opportunity to challenge you.

A proof of the statement $\forall d \text{ Loves}(\text{Sam}, d)$ consists of a strategy for winning the game. Such a strategy takes an arbitrary dog d' , provided by the adversary, and proves that “Sam loves d' .” Again, because we don’t actually know *which* dog d' is, we must either (1) prove that $\text{Loves}(\text{Sam}, d')$ simply from the properties that d' has because he is a dog or (2) go back to doing a case analysis, considering each dog d' separately.

- If the statement $\forall d \text{ Loves}(\text{Sam}, d)$ is true, then you have a strategy. No matter how the adversary plays, no matter which dog d' they give you, Sam loves it. Hence, you can win the game by proving that $\text{Loves}(\text{Sam}, d')$.
- If the statement is false, then there is a dog d' that Sam does not love. Any true adversary (not just a friend) will produce this dog, and you will lose the game. Hence, you cannot have a winning strategy.

Proof by Contradiction: A classic technique for proving the statement $\forall d \text{ Loves}(\text{Sam}, d)$ is proof by contradiction. Except in the way that it is expressed, it is exactly the same as the proof by an adversary game.

By way of contradiction assume that the statement is false, that is, $\exists d \neg \text{Loves}(\text{Sam}, d)$ is true. Let d' be some such dog that Sam does not love. Then you must prove that in fact Sam does love d' . This contradicts

the statement that Sam does not love d' . Hence, the initial assumption is false, and $\forall d \text{ Loves}(\text{Sam}, d)$ is true.

Proof by Adversarial Game for More Complex Statements: The advantage to this technique is that it generalizes into a nice game for arbitrarily long statements.

The Steps of the Game

Left to Right: The game moves from left to right, providing an object for each quantifier.

Prover Provides $\exists b$: You, as the prover, must provide any existential objects.

Adversary Provides $\forall d$: The adversary provides any universal objects.

To Win, Prove the Relation Loves (b', d') : Once all the objects have been provided, you (the prover) must prove that the innermost relation is in fact true. If you can, then you win. Otherwise, you lose.

A Proof Is a Strategy: A proof of the statement consists of a strategy such that you win the game no matter how the adversary plays. For each possible move that the adversary takes, such a strategy must specify what move you will counter with.

Negations in Front: To prove a statement with a negation in the front of it, first put the statement into *standard* form with the negation moved to the right. Then prove the statement in the same way.

Examples

$\exists b \forall d \text{ Loves}(b, d)$: To prove “There is a boy that loves every dog,” you must produce a specific boy b' . Then the adversary, knowing your boy b' tries to prove that $\forall d \text{ Loves}(b', d)$ is false. He does this by providing an arbitrary dog d' that he hopes b' does not love. You must prove “ b' loves d' .”

$\neg(\exists b \forall d \text{ Loves}(b, d))$ iff $\forall b \exists d \neg \text{Loves}(b, d)$: With the negation moved to the right, the first quantifier is universal. Hence, the adversary first produces a boy b' . Then, knowing the adversary’s boy, you produce a dog d' . Finally, you prove that $\neg \text{Loves}(b', d')$.

Your proof of the statement could be viewed as a function G that takes as input the boy b' given by the adversary and outputs the dog $d' = D(b')$ countered by you. Here, $d' = D(b')$ is an example of a dog that boy b' does not love. The proof must prove that $\forall b \neg \text{Loves}(b, D(b))$.

Using the Truth of a Statement: When you know a statement is true, you know that an oracle can win the previously stated game for you.

Techniques for Using $\exists d \text{Loves}(\text{Sam}, d)$: If the oracle assures us that it is true for some d , then they must construct a value for d for which it is true.

Techniques for Using $\forall d \text{Loves}(\text{Sam}, d)$: If the oracle assures us that it is true for all d , then we can specify our favorite value for d and they will assure us that it is true for this value.

Implies: Suppose the prover is trying to prove $A \Rightarrow B$. When A is false, the statement is trivially true. Hence, they assume that A is true and proceeds to prove that B is true. Because the prover is assuming A is true, they know that an oracle can win the game associated with A . The prover uses their help to prove B .

Example: A famous proof is as follows. Suppose that an oracle assures that $\forall x (\text{Human}(x) \rightarrow \text{Mortal}(x))$ and that $\text{Human}(\text{Socrates})$ and the prover wants to prove $\text{Mortal}(\text{Socrates})$. Because their oracle assures that something is true for every value of x , the prover can give them this object Socrates and they will assure them of the statement for this object, namely, that $\text{Human}(\text{Socrates}) \rightarrow \text{Mortal}(\text{Socrates})$. The prover points out to their oracle that they happen to be assuring them that $\text{Human}(\text{Socrates})$. Hence, by Modus Ponens, they can assure the prover that $\text{Mortal}(\text{Socrates})$. This completes the proof.

Exercise 34.0.1 Let $\text{Loves}(b, g)$ denote that boy b loves girl g . If Sam loves Mary and Mary does not love Sam back, then we say that Sam loves in vain.

1. Express the following statements using universal and existential quantifiers. Move any negations to the right.
 - (a) “Sam has loved in vain.”
 - (b) “There is a boy who has loved in vain.”
 - (c) “Every boy has loved in vain.”
 - (d) “No boy has loved in vain.”
2. For each of the above statements and each of the two relations in Figure 34.4, prove either that the statement is true for the relation or that it is false:

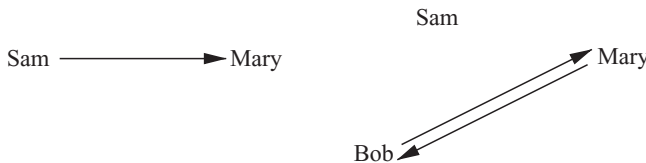


Fig. 34.4 Two relations $\text{Loves}(b, g)$ for Exercise 34.0.2.

Exercise 34.0.2 (See solution in Exercise Solutions.) For each, prove whether true or not when each variable is a real value. Be sure to play the correct game as to who is providing what value:

1. $\forall x \exists y x + y = 5$.

2. $\exists y \forall x x + y = 5$.
3. $\forall x \exists y x \cdot y = 5$.
4. $\exists y \forall x x \cdot y = 5$.
5. $\forall x \exists y x \cdot y = 0$.
6. $\exists y \forall x x \cdot y = 0$.
7. $[\forall x \exists y P(x, y)] \Rightarrow [\exists y \forall x P(x, y)]$.
8. $[\forall x \exists y P(x, y)] \Leftarrow [\exists y \forall x P(x, y)]$.
9. $\forall a \exists y \forall x x \cdot (y + a) = 0$.
10. $\exists a \forall x \exists y [x = 0 \text{ or } x \cdot y = 5]$.

Exercise 34.0.3 The game Ping has two rounds. Player A goes first. Let m_1^A denote his first move. Player B goes next. Let m_1^B denote his move. Then player A goes m_2^A , and player B goes m_2^B . The relation $A \text{ Wins}(m_1^A, m_1^B, m_2^A, m_2^B)$ is true iff player A wins with these moves.

1. Use universal and existential quantifiers to express the fact that player A has a strategy with which they win no matter what player B does. Use $m_1^A, m_1^B, m_2^A, m_2^B$ as variables.
2. What steps are required in the prover–adversary technique to prove this statement?
3. What is the negation of the above statement in standard form?
4. What steps are required in the prover–adversary technique to prove this negated statement?

Exercise 34.0.4 Why does $[\forall n_0, \exists n > n_0, P(n)]$ imply that there are an infinite number of values n for which the property $P(n)$ is true?

35 Time Complexity

It is important to classify algorithms based on whether they solve a given computational problem and, if so, how quickly. Similarly, it is important to classify computational problems based on whether they can be solved and, if so, how quickly.

35.1 The Time (and Space) Complexity of an Algorithm

Purpose

Estimate Duration: To estimate how long an algorithm or program will run.

Estimate Input Size: To estimate the largest input that can reasonably be given to the program.

Compare Algorithms: To compare the efficiency of different algorithms for solving the same problem.

Parts of Code: To help you focus your attention on the parts of the code that are executed the largest number of times. This is the code you need to improve to reduce the running time.

Choose Algorithm: To choose an algorithm for an application:

- If the input size won't be larger than six, don't waste your time writing an extremely efficient algorithm.
- If the input size is a thousand, then be sure the program runs in polynomial, not exponential, time.
- If you are working on the GNOME project and the input size is a billion, then be sure the program runs in linear time.

Time and Space Complexities Are Functions $T(n)$ and $S(n)$: The time complexity of an algorithm is not a single number, but is a function indicating how the running time depends on the size of the input. We often denote this by $T(n)$, giving the number of operations executed on the worst case input instance of size n . An example is $T(n) = 3n^2 + 7n + 23$. Similarly, $S(n)$ gives the size of the rewritable memory the algorithm requires.

Ignoring Details, $\Theta(T(n))$ and $O(T(n))$: Generally, we ignore the low-order terms in the function $T(n)$ and the multiplicative constant in front. We also ignore the function

for small values of n and focus on the *asymptotic* behavior as n becomes very large. Some of the reasons are the following.

Model-Dependent: The multiplicative constant in front of the time depends on how fast the computer is and on the precise definition of “size” and “operation.”

Too Much Work: Counting every operation that the algorithm executes in precise detail is more work than it is worth.

Not Significant: It is much more significant whether the time complexity is $T(n) = n^2$ or $T(n) = n^3$ than whether it is $T(n) = n^2$ or $T(n) = 3n^2$.

Only Large n Matters: One might say that we only consider large input instances in our analysis, because the running time of an algorithm only becomes an issue when the input is large. However, the running time of some algorithms on small input instances is quite critical. In fact, the size n of a realistic input instance depends both on the problem and on the application. The choice was made to consider only large n in order to provide a clean and consistent mathematical definition.

See [Chapter 37](#) on the Theta and BigOh notations.

Definition of Size: The formal definition of the size of an instance is the number of binary digits (*bits*) required to encode it. More practically, the size could be considered to be the number of *digits* or *characters* required to encode it. Intuitively, the size of an instance could be defined to be the area of paper needed to write down the instance, or the number of seconds it takes to communicate the instance along a narrow channel. These definitions are all within a multiplicative constant of each other.

An Integer: Suppose that the input is the value $N = 8,398,346,386,236,876$. The number of bits required to encode it is $\text{Size}(N) = \log_2(N) = \log_2(8,398,346,386,236,876) = 53$, and the number of decimal digits is $\text{Size}(N) = \log_{10}(8,398,346,386,236,876) = 16$. [Chapter 36](#) explains why these are within a multiplicative constant of each other. The one definition that you must *not* use is the *value* of the integer, $\text{Size}(N) = N = 8,398,346,386,236,876$, because it is exponentially different from $\text{Size}(N) = \log_2(N) = 53$.

A Tuple: Suppose that the input is the tuple of b integers $I = \langle x_1, x_2, \dots, x_b \rangle$. The number of bits required to encode it is $\text{Size}(I) = \log_2(x_1) + \log_2(x_2) + \dots + \log_2(x_b) \approx \log_2(x_i) \cdot b$. A natural definition of the size of this tuple is the *number of integers* in it, $\text{Size}(I) = b$. With this definition, it is a much stronger statement to say that an algorithm requires only $\text{Time}(b)$ integer operations independent of how big the integers are.

A Graph: Suppose that the input is the graph $G = \langle V, E \rangle$ with $|V|$ nodes and $|E|$ edges. The number of bits required to encode it is $\text{Size}(G) = 2 \cdot \text{Size}(\text{node}) \cdot |E| = 2 \log_2(|V|) \cdot |E|$. Another reasonable definition of the size of G is the *number of edges*, $G(n) = |E|$. Often the time is given as a function of both $|V|$ and $|E|$. This is within a log factor of the other definitions, which for most applications is fine.

Definition of an Operation: The definition of an operation can be any reasonable operation on two bits, characters, nodes, or integers, depending on whether time is measured in bits, characters, nodes, or integers. An operation could also be defined as any reasonable line of code or the number of seconds that the computation takes on your favorite computer.

Which Input? $T(n)$ is the number of operations required to execute the given algorithm on an input of size n . However, there are 2^n input instances with n bits. Here are three possibilities:

A Typical Input: The problem with considering a typical input instance this is that different applications will have very different typical inputs.

Average or Expected Case: The problem with taking the average over all input instances of size n is that it assumes that all instances are equally likely to occur.

Worst Case: The usual measure is to consider the instance of size n on which the given algorithm is the slowest, namely, $T(n) = \max_{I \in \{I \mid |I|=n\}} \text{Time}(I)$. This measure provides a nice clean mathematical definition and is the easiest to analyze. The only problem is that sometimes the algorithm does much better than the worst case, because the worst case is not a reasonable input. One such algorithm is quick sort (see [Section 12.1](#)).

Time Complexity of a Problem: The time complexity of a problem is the running time of the fastest algorithm that solves the problem.

Example 35.1.1 (Polynomial Time vs. Exponential Time)

Suppose program P_1 requires $T_1(n) = n^4$ operations and P_2 requires $T_2(n) = 2^n$. Suppose that your machine executes 10^6 operations per second. If $n = 1,000$, what is the running time of these programs?

Answer

1. $T_1(n) = (1,000)^4 = 10^{12}$ operations, requiring 10^6 seconds, or 11.6 days.

2. $T_2(n) = 2^{(10^3)}$ operations. The number of years is $\frac{2^{(10^3)}}{10^6 \times 60 \times 60 \times 24 \times 365}$. This is too big for my calculator. The log of this number is $10^3 \times \log_{10}(2) - \log_{10}(10^6 \times 60 \times 60 \times 24 \times 365) = 301.03 - 6 - 7.50 = 287.53$. Therefore, the number of years is $10^{287.53} = 3.40 \times 10^{287}$. Don't wait for it.

Example 35.1.2 (Instance Size N vs. Instance Value N)

Two simple algorithms, summation and factoring.

The Problems and Algorithms

Summation: The task is to sum the N entries of an array, that is, $A(1) + A(2) + A(3) + \cdots + A(N)$.

Factoring: The task is to find divisors of an integer N . For example, on input $N = 5917$ we output that $N = 97 \times 61$. (This problem is central to cryptography.) The algorithm checks whether N is divisible by 2, by 3, by 4, $\dots$ by N .

Time: Both algorithms require $T = N$ operations (additions or divisions).

How Hard? The summation algorithm is considered to be very fast, while the factoring algorithm is considered to be very time-consuming. However, both algorithms take $T = N$ time to complete. The time complexity of these algorithms will explain why.

Typical Values of N : In practice, the N for factoring is much larger than that for summation. Even if you sum all the entries on the entire 8G byte hard drive, then N is still only $N \approx 10^{10}$. On the other hand, the military wants to factor integers $N \approx 10^{100}$. However, the measure of complexity of an algorithm should not be based on how it happens to be used in practice.

Size of the Input: The input for summation contains $n \approx 32N$ bits. The input for factoring contains $n = \log_2 N$ bits. Therefore, with a few hundred bits you can write down a difficult factoring instance that is seemingly impossible to solve.

Time Complexity: The running time of summation is $T(n) = N = \frac{1}{32}n$, which is linear in its input size. The running time of factoring is $T(N) = N = 2^n$, which is exponential in its input size. This is why the summation algorithm is considered to be *feasible*, while the factoring algorithm is considered to be *infeasible*.

Exercise 35.1.1 (See solution in Exercise Solutions.) For each of the two programs considered in [Example 35.1.1](#), if you want it to complete in 24 hours, how big can your input be?

Exercise 35.1.2 In [Example 35.1.1](#), for which input size, approximately, do the programs have the same running times?

Exercise 35.1.3 This problem compares the running times of the following two algorithms for multiplying (see [Section 6.2](#) Multiplying):

algorithm KindergartenAdd(a, b)

⟨*pre-cond*⟩: a and b are integers.

⟨*post-cond*⟩: Outputs $a \times b$.

begin

$c = 0$

 loop $i = 1 \dots a$

$c = c + b$

 end loop

 return(c)

end algorithm

algorithm *GradeSchoolAdd*(a, b)

<pre-cond>: a and b are integers.

<post-cond>: Outputs $a \times b$.

begin

Let $a_{s-1} \dots a_3 a_2 a_1 a_0$ be the decimal digits of a , so that $a = \sum_{i=0}^{s-1} a_i \times 10^i$.

Let $b_{t-1} \dots b_3 b_2 b_1 b_0$ be the decimal digits of b , so that $b = \sum_{j=0}^{t-1} b_j \times 10^j$.

$c = 0$

loop $i = 1 \dots s$

loop $j = 1 \dots t$

$c = c + a_i b_j \times 10^{i+j}$.

end loop

end loop

return(c)

end algorithm

For each of these algorithms, answer the following questions.

1. Suppose that each addition to c requires time 10 seconds and every other operation (for example, multiplying two single digits such as 9×8 and shifting by zero) is free. What is the running time of each of these algorithms, either as a function of a and b or as a function of s and t ? Give everything for this entire question exactly, i.e., not *BigOh*.
2. Let $a = 9,168,391$ and $b = 502$. (Without handing it in, trace the algorithm.) With 10 seconds per addition, how much time (seconds, minutes, etc.) does the computation require?
3. The formal size of an input instance is the number n of bits needed to write it down. What is n as a function of our instance $\langle a, b \rangle$?
4. Suppose your job is to choose the worst case instance $\langle a, b \rangle$ (i.e., the one that maximizes the running time), but you are limited in that you can only use n bits to represent your instance. Do you set a big and b small, a small and b big, or a and b the same size? Give the worst case a and b , or s and t , as a function of n .
5. The running time of an algorithm is formally defined to be a function $T(n)$ from n to the time required for the computation on the worst case instance of size n . Give $T(n)$ for each of these algorithms. Is this polynomial time?

Exercise 35.1.4 (See solution in Exercise Solutions.) Suppose that someone has developed an algorithm to solve a certain problem, which runs in time $T(n, k) \in \Theta(f(n, k))$, where n is the size of the input, and k is a parameter we are free to choose (we can choose it to depend on n). In each case, determine the value of the parameter $k(n)$ to achieve the (asymptotically) best running time. Justify your answer. I recommend not trying much fancy math. Think of n as being some big fixed number. Try some value of k , say, $k = 1$, $k = n^a$, or $k = 2^{an}$ for some constant a . Then note whether increasing or decreasing k increases or decreases f . Recall that “asymptotically” means that we only need the minimum to be within a multiplicative constant.

1. You might want to first prove that $g + h = \Theta(\max(g, h))$.
2. $f(n, k) = \frac{n+k}{\log k}$. This is needed for the radix counting sort in Section 5.4.
3. $f(n, k) = \frac{n^3}{k} + k \cdot n$.
4. $f(n, k) = \log^3 k + \frac{2^n}{k}$.
5. $f(n, k) = \frac{8^n n^2}{k} + k \cdot 2^n + k^2$.

35.2 The Time Complexity of a Computational Problem

The Formal Definition of the Time Complexity of a Problem: As said, the time complexity of a problem is the running time of the fastest algorithm that solves the problem. We will now define this more carefully, using the existential and universal quantifiers that were defined in [Chapter 34](#).

The Time Complexity of a Problem: The time complexity of a computational problem P is the minimum time needed by an algorithm to solve it.

Upper Bound: Problem P is said to be computable in time $T_{upper}(n)$ if there is an algorithm A that outputs the correct answer, namely, $A(I) = P(I)$, within the bounded time, namely, $Time(A, I) \leq T_{upper}(|I|)$, on every input instance I . The formal statement is

$$\exists A, \forall I, [A(I) = P(I) \text{ and } Time(A, I) \leq T_{upper}(|I|)].$$

$T_{upper}(n)$ is said to be only an *upper bound* on the complexity of the problem P , because there may be another algorithm that runs faster. For example, $P = \text{Sorting}$ is computable in $T_{upper}(n) = O(n^2)$ time. It is also computable in $T_{upper}(n) = O(n \log n)$.

Lower Bound of a Problem: A lower bound on the time needed to solve a problem states that no matter how smart you are, you cannot solve the problem faster than the stated time $T_{lower}(n)$, because such algorithm simply does not exist. There may be algorithms that give the correct answer or run sufficiently quickly on some input instances. But for every algorithm, there is at least one instance I for which either the algorithm gives the wrong answer, i.e., $A(I) \neq P(I)$, or it takes too much time, i.e., $Time(A, I) \geq T_{lower}(|I|)$. The formal statement is the negation (except for $\geq$ vs. $>$) of that for the upper bound:

$$\forall A, \exists I, [A(I) \neq P(I) \text{ or } Time(A, I) \geq T_{lower}(|I|)].$$

For example, it should be clear that no algorithm can sort n values in only $T_{lower} = \sqrt{n}$ time, because in that much time the algorithm could not even look at all the values.

Proofs Using the Prover–Adversary Game: Recall the technique described in [Chapter 34](#) for proving statements with existential and universal quantifiers.

Upper Bound: We can use the prover–adversary game to prove the upper bound statement $\exists A, \forall I, [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq T_{\text{upper}}(|I|)]$ as follows: You, the prover, provide the algorithm A . Then the adversary provides an input I . Then you must prove that your A on input I gives the correct output in the allotted time. Note this is what we have been doing throughout the book: providing algorithms and proving that they work.

Lower Bound: A proof of the lower bound $\forall A, \exists I, [A(I) \neq P(I) \text{ or } \text{Time}(A, I) \geq T_{\text{lower}}(|I|)]$ consists of a strategy that, when given an algorithm A by an adversary, you, the prover, study their algorithm and provide an input I . Then you prove either that their A on input I gives the wrong output or that it runs in more than the allotted time.

Exercise 35.2.1 (See solution in Exercise Solutions.) Let $\text{Works}(P, A, I)$ be true if algorithm A halts and correctly solves problem P on input instance I . Let $P = \text{Halting}$ be the halting problem that takes a Java program I as input and tells you whether or not it halts on the empty string. Let $P = \text{Sorting}$ be the sorting problem that takes a list of numbers I as input and sorts them. For each part, explain the meaning of what you are doing and why you don't do it another way.

1. Recall that a problem is computable if and only if there is an algorithm that halts and returns the correct solution on every valid input. Express in first-order logic that Sorting is computable.
2. Express in first-order logic that Halting is not computable.
3. Express in first-order logic that there are uncomputable problems.
4. Explain what the following means (not simply by saying the same in words), and either prove or disprove it: $\forall I, \exists A, \text{Works}(\text{Halting}, A, I)$.
5. Explain what the following means, and either prove or disprove it: $\forall A, \exists P, \forall I, \text{Works}(P, A, I)$. (Hint: An algorithm A on an input I can either halt and give the correct answer, halt and give the wrong answer, or run forever.)

36 Logarithms and Exponentials

Logarithms $\log_2(n)$ and exponentials 2^n arise often when analyzing algorithms.

Uses: These are some of the places that you will see them.

Divide a Logarithmic Number of Times: Many algorithms repeatedly cut the input instance in half. A classic example is binary search ([Section 1.4](#)): You take something of size n and you cut it in half, then you cut one of these halves in half, and one of these in half, and so on. Even for a very large initial object, it does not take very long until you get a piece of size below 1. The number of divisions required is about $\log_2(n)$. Here the base 2 is because you are cutting them in half. If you were to cut them into thirds, then the number of times to cut would be about $\log_3(n)$.

A Logarithmic Number of Digits: Logarithms are also useful because writing down a given integer value n requires $\lceil \log_{10}(n+1) \rceil$ decimal digits. For example, suppose that $n = 1,000,000 = 10^6$. You would have to divide this number by 10 six times to get to 1. Hence, by definition, $\log_{10}(n) = 6$. This, however, is the number of zeros, not the number of digits. We forgot the leading digit 1. The formula $\lceil \log_{10}(n+1) \rceil = 7$ does the trick. For the value $n = 6,372,845$, the number of digits is given by $\log_{10}(6,372,846) = 6.804333$, rounded up to 7. Being in computer science, we store our values using bits. Similar arguments give that $\lceil \log_2(n+1) \rceil$ is the number of bits needed.

Height and Size of Binary Tree: A complete balanced binary tree of height h has 2^h leaves and $n = 2^{h+1} - 1$ nodes. Conversely, if it has n nodes, then its height is $h \approx \log_2 n$.

Exponential Search: Suppose a solution to your problem is represented by n digits. There are 10^n such strings of n digits. Doing a blind search through them all would take too much time.

Rules: There are lots of rules about logs and exponentials that one might learn. Personally, I like to confine them to the following:

$$b^n = \overbrace{b \times b \times b \times \cdots \times b}^n$$
 This is the definition of exponentiation. Where b^n is n b s multiplied together.

$b^n \times b^m = b^{n+m}$: This is proved simply by counting the number of b s being multiplied:

$$\overbrace{(b \times b \times b \times \cdots \times b)}^n \times \overbrace{(b \times b \times b \times \cdots \times b)}^m = \overbrace{b \times b \times b \times \cdots \times b}^{n+m}.$$

$b^0 = 1$: One might guess that zero b s multiplied together is zero, but it needs to be one. One argument for this is as follows: $b^n = b^{0+n} = b^0 \times b^n$. For this to be true, b^0 must be one.

$b^{\frac{1}{2}} = \sqrt{b}$: By definition, $\sqrt{b}$ is the positive number that when multiplied by itself gives b . $b^{\frac{1}{2}}$ meets this definition because $b^{\frac{1}{2}} \times b^{\frac{1}{2}} = b^{\frac{1}{2} + \frac{1}{2}} = b^1 = b$.

$b^{-n} = 1/b^n$: The fact that this needs to be true can be argued in a similar way. $1 = b^{n+(-n)} = b^n \times b^{-n}$. For this to be true, b^{-n} must be $1/b^n$.

$(b^n)^m = b^{n \times m}$: Again, we count the number of b s:

$$\begin{aligned} & \overbrace{\overbrace{(b \times b \times b \times \cdots \times b)}^n \times \overbrace{(b \times b \times b \times \cdots \times b)}^n \times \cdots \times \overbrace{(b \times b \times b \times \cdots \times b)}^n}^m \\ &= \overbrace{b \times b \times b \times \cdots \times b}^{n \times m}. \end{aligned}$$

If $x = \log_b(n)$, then $n = b^x$: This is the definition of logarithms.

$\log_b(1) = 0$: This follows from $b^0 = 1$.

$\log_b(b^x) = x$ and $b^{\log_b(n)} = n$: Substituting $n = b^x$ into $x = \log_b(n)$ gives the first, and substituting $x = \log_b(n)$ into $n = b^x$ gives the second.

$\log_b(n \times m) = \log_b(n) + \log_b(m)$: The number of digits to write down the product of two integers is the number to write down each of them separately (up to rounding errors). We prove it by applying the definition of logarithms and the above rules: $b^{\log_b(n \times m)} = n \times m = b^{\log_b(n)} \times b^{\log_b(m)} = b^{\log_b(n) + \log_b(m)}$. It follows that $\log_b(n \times m) = \log_b(n) + \log_b(m)$.

$\log_b(n^d) = d \times \log_b(n)$: This is an extension of the above rule.

$\log_b(n) - \log_b(m) = \log_b(n) + \log_b(\frac{1}{m}) = \log_b(\frac{n}{m})$: This is another extension of the above rule.

$d^{c \log_2(n)} = n^{c \log_2(d)}$: This rule states that you can move things between the base and the exponent as long as you insert or remove a log. The proof is as follows: $d^{c \log_2(n)} = (2^{\log_2(d)})^{c \log_2(n)} = 2^{\log_2(d) \times c \log_2(n)} = 2^{\log_2(n) \times c \log_2(d)} = (2^{\log_2(n)})^{c \log_2(d)} = n^{c \log_2(d)}$.

$\log_2(n) = 3.32 \dots \times \log_{10}(n)$: The number of bits needed to express the integer n is 3.32... times the number of decimal digits needed. This can be seen as follows. Suppose $x = \log_2 n$. Then $n = 2^x$, giving $\log_{10} n = \log_{10}(2^x) = x \cdot \log_{10} 2$. Finally,

$$x = \frac{1}{\log_{10} 2} \log_{10}(n) = 3.32 \dots \log_{10} n.$$

Which Base? We will write $\Theta(\log(n))$ without giving an explicit base. A high school student might use base 10 as the default, a scientist base $e = 2.718\dots$, and a computer scientist base 2. My policy is to exclude the base when it does not matter. As seen above, $\log_{10}(n)$, $\log_2(n)$, and $\log_e(n)$ differ only by multiplicative constants. In general, we ignore multiplicative constants, and hence the base used is irrelevant. I only include the base when the base matters. For example, 2^n and 10^n differ by much more than a multiplicative constant.

The Ratio $\frac{\log a}{\log b}$: When computing the ratio between two logarithms, the base used does not matter, because changing the base will introduce the same constant on both the top and the bottom, which will cancel. Hence, when computing such a ratio, you can choose whichever base makes the calculation the easiest. For example, to compute $\frac{\log 16}{\log 8}$, the obvious base to use is 2, because $\frac{\log_2 16}{\log_2 8} = \frac{4}{3}$. On the other hand, to compute $\frac{\log 9}{\log 27}$, the obvious base to use is 3, because $\frac{\log_3 9}{\log_3 27} = \frac{2}{3}$.

Exercise 36.0.1 (See solution in Exercise Solutions.) Simplify the following exponentials: $a^3 \times a^5$, $3^a \times 5^a$, $3^a + 5^a$, $2^{6 \log_4 n + 7}$, and $n^{3/\log_2 n}$.

37 Asymptotic Growth

Classes of Growth Rates: It is important to be able to classify functions $f(n)$ based on how quickly they grow: The following table outlines the few easy rules with which to classify functions with the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^{en})$.

c	b^a	d	e	Class	Θ	Examples
> 0	> 1	Any	Any	Exponentials	$2^{\Theta(n)}$	$2^n, \frac{3^{0.001n}}{n^{100}}$
	$= 1$	> 0	Any	Polynomials:	$n^{\Theta(1)}$	$n^4, \frac{5n^{0.0001}}{\log^{100}(n)}$
		$= 2$	Any	• Quadratic	$\Theta(n^2)$	$5n^2, 2n^2 + 7n + 8$
		$= 1$	$= 1$	• Sorting time	$\Theta(n \log n)$	$5n \log n + 3n$
		$= 1$	$= 0$	• Linear	$\Theta(n)$	$5n + 3$
		$= 0$	> 0	Polylogarithms:	$\log^{\Theta(1)}(n)$	$5 \log^3(n)$
			$= 1$	• Logarithms	$\Theta(\log n)$	$5 \log(n)$
			$= 0$	Constants	$\Theta(1)$	$5, 5 + \sin n$
		< 0	Any	Decreasing polynomials	$\frac{1}{n^{\Theta(1)}}$	$\frac{1}{n^4}, \frac{5 \log^{100}(n)}{n^{0.0001}}$
	< 1	Any	Any	Decreasing exponentials	$\frac{1}{2^{\Theta(1)}}$	$\frac{1}{2^n}, \frac{1}{3^{0.001n}}$

Asymptotic Notation: When we want to bound the growth of a function while ignoring multiplicative constants, we use the following notation:

Name	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = O(g(n))$	$f(n) \leq O(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$

Purpose:

Time Complexity: Generally, the functions that we will be classifying will be the time or space complexities of programs. On the other hand, these ideas can also be used to classify any function.

Function Growth: The purpose of classifying a function is to give an idea of how fast it grows without going into too much detail.

Asymptotic Growth Rate: When classifying animals, Darwin decided not to consider whether the animal sleeps during the day, but to consider whether it has hair. When classifying functions, complexity theorists decided to not consider its behavior for small values of n or even whether it is monotone increasing, but how quickly it grows when its input n grows really big. This is referred to as the *asymptotics* of the function. Here are some examples of different growth rates:

Function $T(n)$	Approximate value of $T(n)$ for $n =$				
	10	100	1,000	10,000	Item
5	5	5	5	5	Virus
$\log_2 n$	3	6	9	13	Amoeba
$\sqrt{n}$	3	10	31	100	Bird
n	10	100	1,000	10,000	Human
$n \log n$	30	600	9,000	130,000	Giant
n^2	100	10,000	10^6	10^8	Elephant
n^3	1,000	10^6	10^9	10^{12}	Dinosaur
2^n	1,024	10^{30}	10^{300}	10^{3000}	The Universe

Note: The universe contains approximately 10^{80} particles.

Exponential vs. Polynomial: The table shows that an exponential function like $f(n) = 2^n$ grows extremely quickly. In fact, for sufficiently big n , this exponential 2^n grows much faster than any polynomial, even $n^{1,000,000}$. To take this to an extreme, the function $f(n) = 2^{0.001n}$ is also an exponential. It too grows much faster than $n^{1,000,000}$ for sufficiently large n .

Polynomial vs. Logarithmic: The table also shows that a logarithmic function like $f(n) = \log_2 n$ grows, but very slowly. Hence, for sufficiently large n , it is bigger than any constant, but smaller than any polynomial.

Exercise 37.0.1 Give a value of n for which $n^{1,000,000} < 10^n$.

Give a value of n for which $n^{1,000} < 10^{0.001n}$.

Exercise 37.0.2 Give a value of n for which $(\log_{10} n)^{1,000,000} < n$.

Give a value of n for which $(\log_{10} n)^{1,000} < n^{0.001}$.

37.1 Steps to Classify a Function

Given a function $f(n)$, we will classify it according to its growth using the following steps.

1) Put $f(n)$ into Basic Form: Though there are strange functions out there, most functions $f(n)$ can be put into a basic form consisting of the sum of a number of terms,

where each term has the basic form $c \cdot b^{an} \cdot n^d \cdot (\log n)^e$, where a, b, c, d , and e are real constants.

Examples

- If $f(n) = 3 \cdot 2^{4n} \cdot n^7 \cdot (\log n)^5$, then $a = 4$, $b = 2$, $c = 3$, $d = 7$, and $e = 5$.
- Suppose $f(n) = n^2$. This has no exponential part b^{an} , but can be viewed as having $a = 0$, or $b = 1$, or $a^b = 1$. (Recall $x^0 = 1$ and $1^x = 1$.) The exponent on the polynomial n is $d = 2$. There is no logarithmic factor, so we have $e = 0$. Finally, the constant in front is $c = 1$.
- In $f(n) = 1/n^6 = n^{-6}$, it is also useful to see that $d = -6$.
- If $f(n) = n^2 / \log n + 5$, then the function has two terms. In the first, $a^b = 1$, $c = 1$, $d = 2$, and $e = -1$. In the second, $a^b = 1$, $c = 5$, $d = 0$, and $e = 0$.

2) Get the Big-Picture Growth: We classify the set of all vertebrate animals into mammals, birds, reptiles, and so on. Similarly, we will classify functions into the major groups exponentials $2^{\Theta(n)}$, polynomials $n^{\Theta(1)}$, polylogarithms $\log^{\Theta(1)}(n)$, and constants $\Theta(1)$.

Exponentials $2^{\Theta(n)}$: If the function $f(n)$ is the sum of a bunch of things, one of which is $c \cdot b^{an} \cdot n^d \cdot (\log n)^e$, where $b^a > 1$, then $f(n)$ is considered to be an exponential.

Examples Included

- $f(n) = 2^n$ and $f(n) = 3^{5n}$.
- $f(n) = 2^n \cdot n^2 \log_2 n - 7n^8$ and $f(n) = \frac{2^n}{n^2}$.

Examples Not Included

- $f(n) = 1^n = 2^{0 \cdot n} = 1$, $f(n) = 2^{-1 \cdot n} = (\frac{1}{2})^n$, and $f(n) = n^{1,000,000}$ (too small).
- $f(n) = n! \approx n^n = 2^{n \log_2 n}$ and $f(n) = 2^{n^2}$ (too big).

Definition of an Infeasible Algorithm: An algorithm is considered to be *infeasible* if it runs in exponential time. This is because such functions grow extremely quickly as n gets larger.

$(b^a)^n$: We require $b^a > 1$ because $b^{an} = (b^a)^n$, which grows as long as the base b^a is at least one.

The Notation $2^{\Theta(n)}$: We will see later that $\Theta(1)$ denotes any constant greater than zero. The notation $2^{\Theta(n)} = 2^{\Theta(1) \cdot n}$ is used to represent the class of exponentials, because $b^{an} = 2^{(a \log_2 b) \cdot n}$ and the constant $a \log_2 b = \log_2 (b^a)$ is greater than zero as long as b^a is greater than 1. Recall $\log_2 1 = 0$.

Bounded Between: By these rules, $f(n) = c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ is exponential if $b^a > 1$, no matter what the constants c, d, e , and f are. Consider

$f(n) = 2^n / n^{100}$. The rule states that it is an exponential because $b^a = 2^1 > 1$ and $d = -100$. We might question this, thinking that dividing by n^{100} would not let it grow fast enough to be considered to be an exponential. We see that it does grow fast enough by proving that it is bounded between the two exponential functions $2^{0.5n}$ and 2^n .

Polynomial $n^{\Theta(1)}$: If $f(n) = c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ is such that $b^a = 1$, then we can ignore b^{an} , giving $f(n) = c \cdot n^d \cdot (\log n)^e$. If $d > 0$, then the function $f(n)$ is considered to be a polynomial.

Examples Included

- $f(n) = 3n^2$ and $f(n) = 7n^2 - 8n \log n + 2n - 17$.
- $f(n) = \sqrt{n} = n^{1/2}$ and $f(n) = n^{3.1}$.
- $f(n) = n^2 \log_2 n$ and $f(n) = \frac{n^2}{\log_2 n}$.
- $f(n) = 7n^3 \log^7 n - 8n^2 \log n + 2n - 17$.

Examples Not Included

- $f(n) = n^0 = 1$, $f(n) = n^{-1} = \frac{1}{n}$, and $f(n) = \log n$ (too small).
- $f(n) = n^{\log n}$ and $f(n) = 2^n$ (too big).

Definition of a Feasible Algorithm: An algorithm is considered to be *feasible* if it runs in polynomial time. (This is not actually true if $f(n) = n^{1,000,000}$.)

The Notation $n^{\Theta(1)}$: $\Theta(1)$ denotes any constant greater than zero, and hence $n^{\Theta(1)}$ represents any function $f(n) = n^d$ where $d > 0$.

Bounded Between: Though it would not be considered one in a mathematical study of polynomials, we also consider $f(n) = 3n^2 \log n$ to be a polynomial, because it is bounded between n^2 and n^3 , which clearly are polynomials.

Polylogarithms $\log^{\Theta(1)}(n)$: Powers of logs like $(\log n)^3$ are referred to as *polylogarithms*. These are often written as $\log^3 n = (\log n)^3$. This is different from $\log(n^3) = 3 \log n$.

Example Included

- $f(n) = 7(\log_2 n)^5$, $f(n) = 7 \log_2 n$, and $f(n) = 7\sqrt{\log_2 n}$.
- $f(n) = 7(\log_2 n)^5 + 6(\log_2 n)^3 - 19 + 7(\log_2 n)^2 / n$.

Example Not Included

- $f(n) = n$ (too big).

Constants $\Theta(1)$: A constant function is one whose output does not depend on its input, for example, $f(n) = 7$. One algorithm for which this function arises is popping an element off a stack that is stored as a linked list. This takes maybe seven operations, independent of the number n of elements in the stack.

The Notation $n^{\Theta(1)}$: We use the notation $\Theta(1)$ to replace any constant when we do not care what the actual constant is because determining whether it is 7, 9, or 8.829 may be more work and more detail than we need. On the other hand, in most applications being negative [$f(n) = -1$] or zero [$f(n) = 0$] would be quite a different matter. Hence, these are excluded.

Bounded Between: A function like $f(n) = 8 + \sin n$ changes continuously between 7 and 9, and $f(n) = 8 + \frac{1}{n}$ changes continuously on approaching 8. However, if we don't care whether it is 7, 9, or 8.829, why should we care if it is changing between them? Hence, both of these functions are included in $\Theta(1)$. On the other hand, the function $f(n) = \frac{1}{n}$ is not included, because the only constant that it is bounded below by is zero and the zero function is not included.

Examples Included

- $f(n) = 7$ and $f(n) = 8.829$.
- $f(n) = 8 + \sin n, f(n) = 8 + \frac{1}{n}$.

Examples Not Included

- $f(n) = -1$ and $f(n) = 0$ (fails $c > 0$).
- $f(n) = \sin n$ (fails $c > 0$).
- $f(n) = \frac{1}{n}$ (too small).
- $f(n) = \log_2 n$ (too big).

3) Determine $\Theta(f(n))$: We further classify mammals into humans, cats, dogs, and so on. Similarly, we further classify the polynomials $n^{\Theta(1)}$ into linear functions $\Theta(n)$, the time for sorting $\Theta(n \log n)$, quadratics $\Theta(n^2)$, and so on. These are classes that ignore the multiplicative constant.

Steps: One “takes the Theta” of a function $f(n)$ by dropping the low-order terms and then dropping the multiplicative constant c in front of the largest term.

Dropping Low-Order Terms: If $f(n)$ is a set of things added or subtracted together, then each of these things is called a *term*. We determine which of the terms grows fastest as n gets large. The slower-growing terms are referred to as *low-order terms*. We drop them because they are not significant.

Ordering Terms: The fastest-growing term is determined by first taking the term $c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ with the largest b^a value. If the b^a 's of terms are equal, then we take the term with the largest d value. If the d 's are also equal, then we take the term with the largest e value.

Dropping the Multiplicative Constant: The running time of an algorithm might be $f(n) = 3n^2$ or $f(n) = 100n^2$. We say it is $\Theta(n^2)$ when we do not care what the multiplicative constant c is. The function $f(n) = c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ is in the class of functions denoted $\Theta(b^{an} \cdot n^d \cdot (\log n)^e)$.

Examples of Functions

- $f(n) = 3n^3 \log n - 1000n^2 + n - 29$ is in the class $\Theta(n^3 \log n)$.
- $f(n) = 7 \cdot 4^n \cdot n^2 / \log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$ is in the class $\Theta(4^n \cdot n^2 / \log^3 n)$.
- $\frac{1}{n} + 18$ is in the class $\Theta(1)$. Since $\frac{1}{n}$ is a lower-order term than 18, it is dropped.
- $\frac{1}{n^2} + \frac{1}{n}$ is in the class $\Theta(\frac{1}{n})$, because $\frac{1}{n^2}$ is a smaller term.

Examples of Classes

Linear Functions $\Theta(n)$: The classic linear function is $f(n) = c \cdot n + b$. The notation $\Theta(n)$ excludes any with $c \leq 0$ but includes any function that is bounded between two such functions.

What Can Be Done in $\Theta(n)$ Time: Given an input of n items, it takes $\Theta(n)$ time simply to look at the input. Looping over the items and doing a constant amount of work for each takes another $\Theta(n)$ time. Say we take $t_1(n) = 2n$ and $t_2(n) = 4n$ for a total of $6n$ time. Now if you do not want to do more than linear time, are you allowed to do any more work? Sure. You can do something that takes $t_3(n) = 3n$ time and something else that takes $t_4(n) = 5n$ time. You are even allowed to do a few things that take a constant amount of time, totaling, say, $t_5(n) = 13$. The entire algorithm then takes the sum of these, $t(n) = 14n + 13$ time. This is still considered to be linear time.

Examples Included

- $f(n) = 7n$ and $f(n) = 8.829n$.
- $f(n) = (8 + \sin n)n$ and $f(n) = 8n + \log^{10} n + \frac{1}{n} - 1,000,000$.

Examples Not Included

- $f(n) = -n$ and $f(n) = 0n$ (fails $c > 0$).
- $f(n) = \frac{n}{\log_2 n}$ (too small).
- $f(n) = n \log_2 n$ (too big).

Quadratic Functions $\Theta(n^2)$: Two nested loops from 1 to n take $\Theta(n^2)$ time if each inner iteration takes a constant amount of time. An $n \times n$ matrix requires $\Theta(n^2)$ space if each element takes constant space.

Time for Sorting, $\Theta(n \log n)$: Another running time that arises often in algorithms is $\Theta(n \log n)$. For example, this is the number of comparisons needed to sort n elements.

Not Linear: The function $f(n) = n \log n$ grows slightly too quickly to be in the linear class of functions $\Theta(n)$. This is because $n \log n$ is $\log n$ times n , and $\log n$ is not constant.

A Polynomial: The classes $\Theta(n)$, $\Theta(n \log n)$, and $\Theta(n^2)$ are subclasses of the class of polynomial functions $n^{\Theta(1)}$. For example, though the function $f(n) = n \log n$ is too big for $\Theta(n)$ and too small $\Theta(n^2)$, it is in $n^{\Theta(1)}$ because it is bounded between n^1 and n^2 , both of which are in $n^{\Theta(1)}$.

Logarithms $\Theta(\log(n))$: See Chapter 36 for how logarithmic functions like $\log_2(n)$ arise and for some of their rules.

Which Base: We write $\Theta(\log(n))$ without giving an explicit base. As shown in the list of rules about logarithms, $\log_{10}(n)$, $\log_2(n)$, and $\log_e(n)$ differ only by a multiplicative constant. Because we are ignoring multiplicative constants anyway, which base is used is irrelevant. The rules also indicate that $8 \log_2(n^5)$ also differs only by a multiplicative constant. All of these functions are include in $\Theta(\log(n))$.

Exercise 37.1.1 Which grows faster, 3^{4n} or 4^{3n} ?

Exercise 37.1.2 Does the notation $(\Theta(1))^n$ mean the same thing as $2^{\Theta(n)}$?

Exercise 37.1.3 Prove that $2^{0.5n} \leq 2^n/n^{100} \leq 2^n$ for sufficiently big n .

Exercise 37.1.4 (See solution in Exercise Solutions.) Prove that $n^2 < 3n^2 \log n \leq n^3$ for sufficiently big n .

Exercise 37.1.5 (See solution in Exercise Solutions.) Sort the terms in $f(n) = 100n^{100} + 3^{4n} + \log^{1,000} n + 4^{3n} + 2^{0.001n}/n^{100}$.

Exercise 37.1.6 For each of the following functions, sort its terms by growth rate. Get the big picture growth by classifying it into $2^{\Theta(n)}$, $n^{\Theta(1)}$, $\log^{\Theta(1)}(n)$, $\Theta(1)$ or into a similar and appropriate class. Also give its Theta approximation.

1. $f(n) = 5n^3 - 17n^2 + 4$.
2. $f(n) = 5n^3 \log n + 8n^3$.
3. $f(n) = 2^{2^{5n}}$.
4. $f(n) = 7^3 \log_2 n$.
5. $f(n) = \{1 \text{ if } n \text{ is odd, } 2 \text{ if } n \text{ is even}\}$.
6. $f(n) = 2 \cdot 2^n \cdot n^2 \log_2 n - 7n^8 + 7 \frac{3^n}{n^2}$.
7. $f(n) = 100n^{100} + 3^{4n} + \log^{1,000}(n) + 4^{3n}$.
8. $f(n) = 6 \frac{n^4}{\log^3 n} + 8n^{100} 2^{-5n} + 17$.
9. $f(n) = \frac{1}{n^2} + \frac{5 \log n}{n}$.
10. $f(n) = 7 \sqrt[5]{n} + 6 \sqrt[3]{n}$.
11. $f(n) = \frac{6n^{5.2} + 7n^{7.5}}{2n^{3.1} + 7n^{2.4}}$.
12. $f(n) = -2n$.
13. $f(n) = 5n^{\log^3 n}$.

Exercise 37.1.7 For each pair of classes of functions, how are they similar? How are they different? If possible, give a function that is included in the first of these but not included in the second. If possible, do the reverse, giving a function that is included in the second but not in the first.

1. $\Theta(2^{2n})$ and $\Theta(2^{3n})$.
2. $\Theta(2^n)$ and $3^{\Theta(n)}$.

37.2 More about Asymptotic Notation

Other Useful Notations:

Name	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = O(g(n))$	$f(n) \leq O(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) << o(g(n))$	$f(n) << g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) >> \omega(g(n))$	$f(n) >> g(n)$
Tilde	$f(n) = \tilde{\Theta}(g(n))$	$f(n) \in \tilde{\Theta}(g(n))$	$f(n) \approx \log^{\Theta(1)} \cdot g(n)$

Same: $7 \cdot n^3$ is within a constant of n^3 . Hence, it is in $\Theta(n^3)$, $O(n^3)$, and $\Omega(n^3)$. However, because it is not much smaller than n^3 , it is not in $o(n^3)$, and because it is not much bigger, it not in $\omega(n^3)$.

Smaller: $7 \cdot n^3$ is asymptotically much smaller than n^4 . Hence, it is in $O(n^4)$ and in $o(n^4)$, but it is not in $\Theta(n^4)$, $\Omega(n^4)$, or $\omega(n^4)$.

Bigger: $7 \cdot n^3$ is asymptotically much bigger than n^2 . Hence, it is in $\Omega(n^2)$ and in $\omega(n^2)$, but it is not in $\Theta(n^2)$, $O(n^2)$, or $o(n^2)$.

Log Factors: $7n^3 \log^2 n = \tilde{\Theta}(n^3)$ ignores the logarithmic factors.

Notation Considerations

“ $\in$ ” vs. “ $=$ ”: I consider $\Theta(n)$ to be a class of functions, so I *ought* to use the set notation, $f(n) \in \Theta(g(n))$, to denote membership.

On the other hand, ignoring constant multiplicative factors, $f(n)$ has the same asymptotic growth as $g(n)$. Because of this, the notation $7n = \Theta(n)$ makes sense. This notation is standard.

Even the statements $3n^2 + 5n - 7 = n^{\Theta(1)}$ and $2^{3n} = 2^{\Theta(n)}$ make better sense when you think of the symbol Θ to mean “some constant.” However, be sure to remember that $4^n \cdot n^2 = 2^{\Theta(n)}$ is also true.

“ $=$ ” vs. “ $<$ ”: $7n = O(n^2)$ is also standard notation. This makes less sense to me. Because it means that $7n$ is at most some constant times n^2 , a better notation would be $7n \leq O(n^2)$. The standard notation is even more awkward, because

$O(n) = O(n^2)$ should be true, but $O(n^2) = O(n)$ should be false. What sense does that make?

More Details: You can decide how much information about a function you want to reveal. If $f(n) = 5n^2 + 3n$, you could say:

- $f(n) \in n^{\Theta(1)}$, i.e., a polynomial.
- $f(n) \in \Theta(n^2)$, i.e., a quadratic.
- $f(n) \in (5 + o(1))n^2 = 5n^2 + o(n^2)$, i.e., $5n^2$ plus some low-order terms.
- $f(n) \in 5n^2 + O(n)$, i.e., $5n^2$ plus at most some linear terms.

The Formal Definitions of Theta and BigOh:

$$\begin{aligned}
 f(n) \in \Theta(g(n)) &\text{ iff } \exists c_1, c_2 > 0 \quad \exists n_0 \forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n). \\
 f(n) \in O(g(n)) &\text{ iff } \exists c > 0 \quad \exists n_0 \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n). \\
 f(n) \in \Omega(g(n)) &\text{ iff } \exists c > 0 \quad \exists n_0 \forall n \geq n_0, c \cdot g(n) \leq f(n). \\
 f(n) \in n^{\Theta(1)} &\text{ iff } \exists c_1, c_2 > 0 \quad \exists n_0 \forall n \geq n_0, n^{c_1} \leq f(n) \leq n^{c_2}. \\
 f(n) \in 2^{\Theta(n)} &\text{ iff } \exists c_1, c_2 > 0 \quad \exists n_0 \forall n \geq n_0, 2^{c_1 n} \leq f(n) \leq 2^{c_2 n}. \\
 f(n) \notin \Theta(g(n)) &\text{ iff } \forall c_1, c_2 > 0 \quad \forall n_0 \exists n \geq n_0, [c_1 \cdot g(n) > f(n) \text{ or } f(n) > c_2 \cdot g(n)].
 \end{aligned}$$

Bounded Between: The statement $f(n) \in \Theta(g(n))$ means that the function $f(n)$ is bounded between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$. See [Figure 37.1](#).

Requirements on c_1 and c_2 : The only requirements on the constants are that c_1 be sufficiently small (e.g., 0.001) but positive and c_2 be sufficiently large (e.g., 1,000) to work, and that they be fixed (that is, do not depend on n). We allow unreasonably extreme values like $c_2 = 10^{100}$, to make the definition mathematically clean and not geared to a specific application.

Sufficiently Large n : Given fixed c_1 and c_2 , the statement $c_1 g(n) \leq f(n) \leq c_2 g(n)$ should be true for all sufficiently large values of n , (i.e., $\forall n \geq n_0$).

Definition of Sufficiently Large n_0 : Again, to make the mathematics clean and not geared to a specific application, we will simply require that there exist some definition of sufficiently large n_0 that works. [Exercise 37.0.2](#) gives an example in which n_0 needs to be unreasonably large.

Proving $f(n) \in \Theta(g(n))$: Use the prover–adversary game.

- You as the prover provide c_1 , c_2 , and n_0 .
- Some adversary gives you an n that is at least your n_0 .
- You then prove that $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Example: For example, $2n^2 + 100n = \Theta(n^2)$. Let $c_1 = 2$, $c_2 = 3$, and $n_0 = 100$. Then, for all $n \geq 100$, we have $c_1 g(n) = 2n^2 \leq 2n^2 + 100n = f(n)$ and $f(n) = 2n^2 + 100n \leq 2n^2 + n \cdot n = 3n^2 = c_2 g(n)$. The values of c_1, c_2 , and n_0 are not unique. For example, $n_0 = 1, c_2 = 102$, and $n_0 = 1$ also work, because for all $n \geq 1$ we have $f(n) = 2n^2 + 100n \leq 2n^2 + 100n^2 = 102n^2 = c_2 g(n)$.

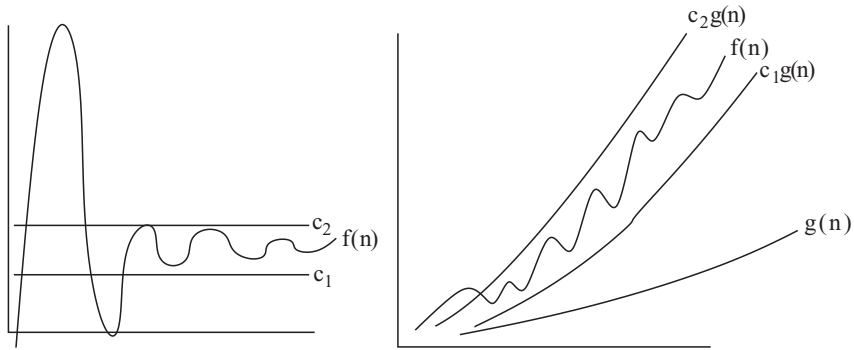


Fig. 37.1 $f(n) \in \Theta(1)$ and $f(n) \in \Theta(g(n))$.

The Formal Definitions of Little Oh and Little Omega

Class	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} =$	A practically equivalent definition
$f(n) = \Theta(g(n))$	Some constant	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
$f(n) = o(g(n))$	Zero	$f(n) = O(g(n))$, but $f(n) \neq \Omega(g(n))$
$f(n) = \omega(g(n))$	∞	$f(n) \neq O(g(n))$, but $f(n) = \Omega(g(n))$

Examples

- $2n^2 + 100n = \Theta(n^2)$ and $\lim_{n \rightarrow \infty} \frac{2n^2 + 100n}{n^2} = 2$.
- $2n + 100 = o(n^2)$ and $\lim_{n \rightarrow \infty} \frac{2n + 100}{n^2} = 0$.
- $2n^3 + 100n = \omega(n^2)$ and $\lim_{n \rightarrow \infty} \frac{2n^3 + 100n}{n^2} = \infty$.

Exercise 37.2.1 As in [Exercise 37.1.7](#), compare the classes $(5 + o(1))n^2$ and $5n^2 + O(n)$.

Exercise 37.2.2 (See solution in Exercise Solutions.) Formally prove or disprove the following:

1. $14n^9 + 5,000n^7 + 23n^2 \log n \in O(n^9)$.
2. $2n^2 - 100n \in \Theta(n^2)$.
3. $14n^8 - 100n^6 \in O(n^7)$.
4. $14n^8 + 100n^6 \in \Theta(n^9)$.
5. $2^{n+1} \in O(2^n)$.
6. $2^{2n} \in O(2^n)$.

Exercise 37.2.3 Prove that if $f_1(n) \in \Theta(g_1(n))$ and $f_2(n) \in \Theta(g_2(n))$, then $f_1(n) + f_2(n) \in \max(\Theta(g_1(n)), \Theta(g_2(n)))$.

Exercise 37.2.4 Prove that if $f_1(n), f_2(n) \in n^{\Theta(1)}$, then $f_1(n) \cdot f_2(n) \in n^{\Theta(1)}$.

Exercise 37.2.5 Let $f(n)$ be a function. As you know, $\Theta(f(n))$ drops low-order terms and the leading coefficient. Explain what each of the following does: $2^{\Theta(\log_2 f(n))}$ and $\log_2(\Theta(2^{f(n)}))$. For each, explain to what extent the function is approximated.

Exercise 37.2.6 Let x be a real value. As you know, $\lfloor x \rfloor$ rounds it down to the next integer. Explain what each of the following does: $2 \cdot \lfloor \frac{x}{2} \rfloor$, $\frac{1}{2} \cdot \lfloor 2 \cdot x \rfloor$, and $2^{\lfloor \log_2 x \rfloor}$.

Exercise 37.2.7 Suppose that $y = \Theta(\log x)$. Which of the following are true: $x = \Theta(2^y)$ and $x = 2^{\Theta(y)}$? Why?

Exercise 37.2.8 (See solution in Exercise Solutions.) It is impossible to algebraically solve the equation $x = 7y^3(\log_2 y)^{18}$ for y .

1. Approximate $7y^3(\log_2 y)^{18}$ and then solve for y . This approximates the value of y .
2. Get a better approximation as follows. Plug in your above approximation for y to express $(\log_2 y)^{18}$ in terms of x . Plug this into $x = 7y^3(\log_2 y)^{18}$. Now solve for y again. (You could repeat this step for better and better approximations.)
3. Observe how a similar technique was used in [Exercises 37.0.1](#) and [37.0.2](#) to approximate a solution for $(\log_{10} n)^{1,000,000} = n$.

38 Adding-Made-Easy Approximations

```

algorithm Eg(n)
  loop i = 1..n
    loop j = 1..i
      loop k = 1..j      The inner loop requires time  $\sum_{k=1}^j 1 = j$ .
        put "Hi"          The next requires  $\sum_{i=1}^i \sum_{k=1}^j 1 = \sum_{j=1}^i j = \Theta(i^2)$ .
      end loop            The total is  $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1 = \sum_{i=1}^n \Theta(i^2) = \Theta(n^3)$ .
    end loop
  end loop
end algorithm

```

Sums arise often in the study of computer algorithms. For example, if the i th iteration of a loop takes time $f(i)$ and it loops n times, then the total time is $f(1) + f(2) + f(3) + \cdots + f(n)$. This we denote as $\sum_{i=1}^n f(i)$. It can be approximated by the integral $\int_{x=1}^n f(x) \delta x$, because the first is the area under the stairs of height $f(i)$ and the second under the curve $f(x)$. (In fact, both $\sum$ (from the Greek letter sigma) and $\int$ (from the old long S) are S for sum.) Note that, even though the individual terms are indexed by i (or x), the total is a function of n . The goal now is to approximate $\sum_{i=1}^n f(i)$ for various functions $f(i)$ (Figure 38.1).

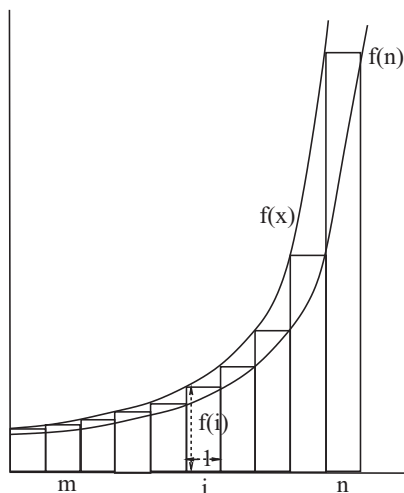


Fig. 38.1 The sum $\sum_{i=1}^n f(i)$ can be approximated by Integral $\int_{x=1}^n f(x) \delta x$.

Beyond learning the classic techniques for computing $\sum_{i=1}^n 2^i$, $\sum_{i=1}^n i$, and $\sum_{i=1}^n \frac{1}{i}$, we do not study how to evaluate sums exactly, but only how to approximate them to within a constant factor. We develop easy rules that most computer scientists use but for some reason are not usually taught, partly because they are not always true. We have formally proven when they are true and when not. We call them collectively the *adding-made-easy technique*.

38.1 The Technique

The following table outlines the few easy rules with which you will be able to compute $\Theta(\sum_{i=1}^n f(i))$ for functions with the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$. (We will consider more general functions at the end of this section.)

b^a	d	e	Type of Sum	$\sum_{i=1}^n f(i)$	Examples
> 1	Any	Any	Geometric increase (dominated by last term)	$\Theta(f(n))$	$\sum_{i=0}^n 2^{2^i} \approx 1 \cdot 2^{2^n}$ $\sum_{i=0}^n b^i = \Theta(b^n)$ $\sum_{i=0}^n 2^i = \Theta(2^n)$
$= 1$	> -1	Any	Arithmetic-like (half of terms approximately equal)	$\Theta(n \cdot f(n))$	$\sum_{i=1}^n i^d = \Theta(n \cdot n^d) = \Theta(n^{d+1})$ $\sum_{i=1}^n i^2 = \Theta(n \cdot n^2) = \Theta(n^3)$ $\sum_{i=1}^n i = \Theta(n \cdot n) = \Theta(n^2)$
	$= -1$	$= 0$	Harmonic	$\Theta(\ln n)$	$\sum_{i=1}^n \frac{1}{i} = \log_e(n) + \Theta(1)$
	< -1	Any	Bounded tail (dominated by first term)	$\Theta(1)$	$\sum_{i=1}^n \frac{1}{i^{1.001}} = \Theta(1)$ $\sum_{i=1}^n \frac{1}{i^2} = \Theta(1)$
< 1	Any	Any			$\sum_{i=1}^n (\frac{1}{2})^i = \Theta(1)$ $\sum_{i=0}^n b^{-i} = \Theta(1)$

Four Different Classes of Solutions: All of the sums that we will consider have one of four different classes of solutions. The intuition for each is quite straightforward.

Geometrically Increasing: If the terms grow very quickly, the total is dominated by the last and biggest term $f(n)$. Hence, one can approximate the sum by only considering the last term: $\sum_{i=1}^n f(i) = \Theta(f(n))$.

Examples: Consider the classic sum in which each of the n terms is twice the previous, $1 + 2 + 4 + 8 + 16 + \dots + 2^n$. Either by examining areas within Figure 38.2.a or 38.1.b or using simple induction, one can prove that the total is always one less than twice the biggest term: $\sum_{i=0}^n 2^i = 2 \times 2^n - 1 = \Theta(2^n)$. More generally, $\sum_{i=0}^n b^i \approx \frac{b}{b-1} \cdot b^n$, which can be approximated by $\Theta(f(n)) = \Theta(b^n)$. (Similarly, $\int_{x=0}^n b^x \delta x = \frac{1}{\ln b} b^n$.) The same is true for even faster growing functions like $\sum_{i=0}^n 2^{2^i} \approx 1 \times 2^{2^n}$.

Basic-Form Exponentials: The same technique, $\sum_{i=1}^n f(i) = \Theta(f(n))$, works for all basic-form exponentials, i.e., for $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ with $b^a > 1$, we have that $\sum_{i=1}^n f(i) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$.

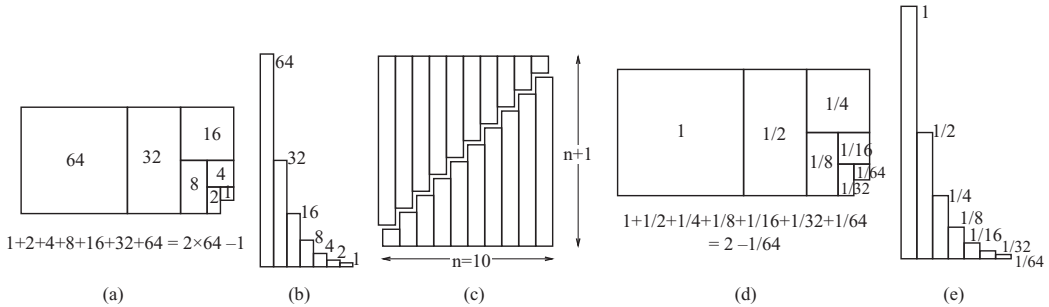


Fig. 38.2 Examples of geometrically increasing, arithmetic-like, and bounded-tail function.

Arithmetic-like: If half of the terms are roughly the same size, then the total is roughly the number of terms times the last term, namely, $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$.

Examples

Constant: Clearly, the sum of n ones is n , i.e., $\sum_{i=1}^n 1 = n$. This is $\Theta(n \cdot f(n))$.

Linear: The classic example is the sum in which each of the n terms is only one bigger than the previous, $\sum_{i=1}^n i = 1 + 2 + 3 + 4 + 5 + \dots + n = \frac{n(n+1)}{2}$. This can be approximated using $\Theta(n \cdot f(n)) = \Theta(n^2)$. See Figure 38.2.

Polynomials: Both $\sum_{i=1}^n i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$ and more generally $\sum_{i=1}^n i^d = \frac{1}{d+1}n^{d+1} + \Theta(n^d)$ can be approximated with $\Theta(n \cdot f(n)) = \Theta(n \cdot n^d) = \Theta(n^{d+1})$. (Similarly, $\int_{x=0}^n x^d dx = \frac{1}{d+1}n^{d+1}$.)

Above Harmonic: $\sum_{i=1}^n \frac{1}{n^{0.999}} \approx 1,000 n^{0.001}$ can be approximated with $\Theta(n \cdot f(n)) = \Theta(n \cdot n^{-0.999}) = \Theta(n^{0.001})$.

Basic-Form Polynomials: The same technique, $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$, works for all basic-form polynomials, constants, and slowly decreasing functions, i.e., for $f(n) = \Theta(n^d \cdot \log^e n)$ with $d > 1$ we have that $\sum_{i=1}^n f(i) = \Theta(n^{d+1} \cdot \log^e n)$.

Bounded Tail: If the terms shrink quickly, the total is dominated by the first and biggest term $f(1)$, which is assumed here to be $\Theta(1)$, i.e., $\sum_{i=1}^n f(i) = \Theta(1)$.

Examples: The classic sum here is when each term is half of the previous, $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots + \frac{1}{2^n}$. See Figure 38.2.d and 38.2.e. The total approaches but never reaches 2, so that $\sum_{i=0}^n (\frac{1}{2})^i = 2 -$

$(\frac{1}{2})^n = \Theta(1)$. Similarly, $\sum_{i=1}^n \frac{1}{n^{1.001}} \approx 1,000 = \Theta(1)$ and $\sum_{i=1}^n \frac{1}{n^2} \approx \frac{\pi}{6} \approx 1.5497 = \Theta(1)$.

Basic-Form with Bounded Tail: The same technique, $\sum_{i=1}^n f(i) = \Theta(1)$, works for all basic-form polynomially or exponentially decreasing functions, i.e., for $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ with $b^a = 1$ and $d < 1$ or with $b^a < 1$.

The Harmonic Sum: The sum $\sum_{i=1}^n \frac{1}{i}$ is referred to as the *harmonic sum* because of its connection to music. It arises surprisingly often and it has an unexpected total (Figure 38.3).

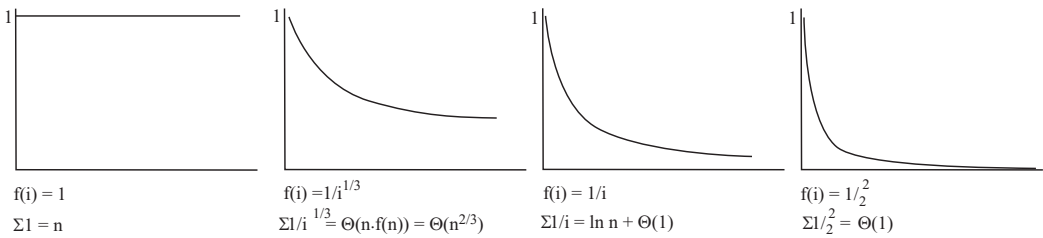


Fig. 38.3 The harmonic sum $\sum_{i=1}^n \frac{1}{i}$ is on the boundary between those total grows with n and those totals are bounded by a constant.

On the Boundary: The boundary between those sums for which $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ and those for which $\sum_{i=1}^n f(i) = \Theta(1)$ occurs when these approximations meet, i.e., when $\Theta(n \cdot f(n)) = \Theta(1)$. This occurs at the harmonic function $f(n) = \frac{1}{n}$. Given that both approximations say the total is $\sum_{i=1}^n \frac{1}{i} = \Theta(1)$, it is reasonable to think that this is the answer, but it is not.

The Total: It turns out that the total is within 1 of the natural logarithm, $\sum_{i=1}^n \frac{1}{i} = \log_e n + \Theta(1)$. (Similarly, $\int_{x=1}^{n+1} \frac{1}{x} \delta x = \log_e n + \Theta(1)$.) See Figure 38.4.

More Examples

Geometric Increasing

- $\sum_{i=1}^n 8 \frac{2^i}{i^{100}} + i^3 = \Theta(\frac{2^n}{n^{100}})$
- $\sum_{i=1}^n 3^{i \log i} + 5^i + i^{100} = \Theta(3^{n \log n})$
- $\sum_{i=1}^n 2^{i^2} + i^2 \log i = \Theta(2^{n^2})$
- $\sum_{i=1}^n 2^{2^i - i^2} = \Theta(2^{2^n - n^2})$.

Arithmetic (Increasing)

- $\sum_{i=1}^n i^4 + 7i^3 + i^2 = \Theta(n^5)$
- $\sum_{i=1}^n i^{4.3} \log^3 i + i^3 \log^9 i = \Theta(n^{5.3} \log^3 n)$.

Arithmetic (Decreasing)

- $\sum_{i=2}^n \frac{1}{\log i} = \Theta\left(\frac{n}{\log n}\right)$
- $\sum_{i=1}^n \frac{\log^3 i}{i^{0.6}} = \Theta(n^{0.4} \log^3 n).$

Bounded Tail

- $\sum_{i=1}^n \frac{\log^3 i}{i^{1.6+3i}} = \Theta(1)$
- $\sum_{i=1}^n \frac{i^{100}}{2^i} = \Theta(1)$
- $\sum_{i=1}^n \frac{1}{2^{2^i}} = \Theta(1).$

Stranger Examples

- A useful fact is $\sum_{i=m}^n f(i) = \sum_{i=1}^n f(i) - \sum_{i=1}^{m-1} f(i)$. Hence,
 $\sum_{i=m}^n \frac{1}{i} = \Theta(\log n) - \Theta(\log m) = \Theta(\log \frac{n}{m}).$

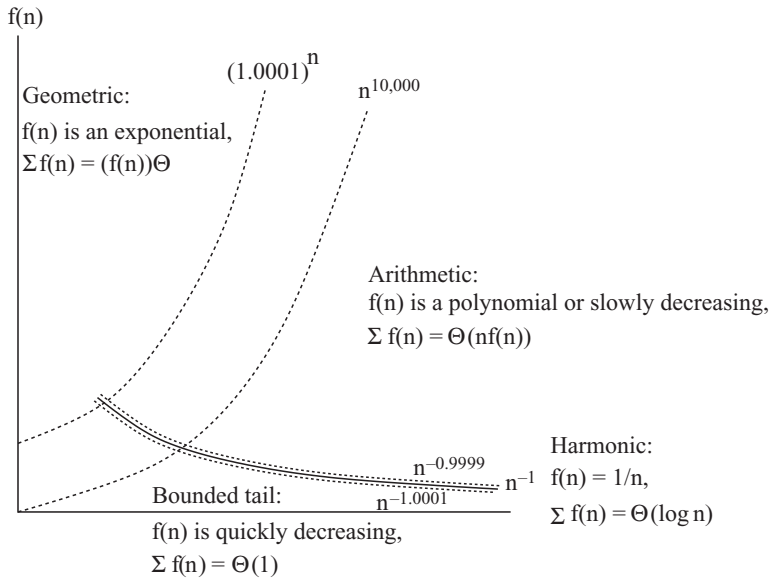


Fig. 38.4 Boundaries between geometric, arithmetic, harmonic, and bounded tail.

- If the sum is arithmetic, then the sum is the number of terms times the largest term. This gives $\sum_{i=m}^{m+n} i^2 = \Theta(n \cdot (m+n)^2)$.
- To solve $\sum_{i=1}^{5n^2+n} i^3 \log i$, let $N = 5n^2 + n$ denote the number of terms. Then $\sum_{i=1}^N i^3 \log i = \Theta(N \cdot f(N)) = \Theta(N^4 \log N)$. Substituting back in for N gives $\sum_{i=1}^{5n^2+n} i^3 \log i = \Theta((5n^2 + n)^4 \log(5n^2 + n)) = \Theta(n^8 \log n)$.
- Between terms, i changes, but n does not. Hence, n can be treated like a constant. For example, $\sum_{i=1}^n i \cdot n \cdot m = nm \cdot \sum_{i=1}^n i = nm \cdot \Theta(n^2) = \Theta(n^3 m)$.

- In $\sum_{i=\frac{n}{2}}^n \frac{1}{i^2}$, the terms are decreasing fast enough to be bounded by the first term. Here, however, the first term is not $\Theta(1)$, but is

$$\Theta\left(\frac{1}{\left(\frac{n}{2}\right)^2}\right) = \Theta\left(\frac{1}{n^2}\right).$$

- When in doubt, start by determining the first term, the last term, and the number of terms. In $\sum_{i=1}^{\log_2 n} 2^{\log_2 n - i} \cdot i^2$, the first term is $f(1) = 2^{\log_2 n - 1} \cdot 1^2 = \Theta(n)$, and the last term is $f(\log n) = 2^{\log_2 n - \log n} \cdot (\log n)^2 = \Theta(\log^2 n)$. The terms decrease geometrically in i . The total is then $\Theta(f(1)) = \Theta(n)$.
- $\sum_{i=1}^n \sum_{j=0}^n i^2 j^3 = \sum_{i=1}^n i^2 [\sum_{j=0}^n j^3] = \sum_{i=1}^n i^2 \Theta(n^4) = \Theta(n^4) [\sum_{i=1}^n i^2] = \Theta(n^4) \Theta(n^3) = \Theta(n^7)$.

Exercise 38.1.1 Give the Θ approximation of the following sums. Indicate which rule you use, and show your work.

1. $\sum_{i=0}^n 7i^3 - 300i^2 + 16$.
2. $\sum_{i=0}^n i^8 + \frac{2^{3i}}{i^2}$.
3. $\sum_{i=0}^n \frac{1}{i^{1.1}}$.
4. $\sum_{i=0}^n \frac{1}{i^{0.9}}$.
5. $\sum_{i=0}^n 7 \frac{i^{3.72}}{\log^2 i} - 300i^2 \log^9 i$.
6. $\sum_{i=1}^n \frac{\log^6 i}{i}$.
7. $\sum_{i=1}^{\log n} n \cdot i^2$.
8. $\sum_{i=0}^n \sum_{j=0}^m \frac{j}{i}$.
9. $\sum_{i=1}^n \sum_{j=1}^i j^i$.
10. $\sum_{i=1}^n \sum_{j=1}^{i^2} ij \log(i)$.

38.2 Some Proofs for the Adding-Made-Easy Technique

This section presents a few of the classic techniques for summing and sketches the proof of the adding-made-easy technique.

Simple Geometric Sums

Theorem: When $b > 1$, $\sum_{i=1}^n b^i = \Theta(f(n))$ and when $b < 1$, $\sum_{i=1}^n f(i) = \Theta(1)$.

Proof:

$$\begin{aligned} S &= 1 + b + b^2 + \dots + b^n \\ b \cdot S &= b + b^2 + b^3 + \dots + b^{n+1}. \end{aligned}$$

Subtracting those two equations gives

$$\begin{aligned} (1 - b) \cdot S &= 1 - b^{n+1} \\ S &= \frac{1 - b^{n+1}}{1 - b} \text{ or } \frac{b^{n+1} - 1}{b - 1} \\ &= \Theta(\max(f(0), f(n))). \end{aligned}$$

Ratio between Terms: To prove that a geometric sum is not more than a constant times the biggest term, we must compare each term $f(i)$ with this biggest term. One way to do this is to first compare each consecutive pairs of terms $f(i)$ and $f(i+1)$.

Theorem: If for all sufficiently large i , the ratio between terms is bounded away from one, i.e., $\exists b > 1, \exists n_0, \forall i \geq n_0, f(i+1)/f(i) \geq b$, then $\sum_{i=1}^n f(i) = \Theta(f(n))$.

Conversely, if $\exists b < 1, \exists n_0, \forall i \geq n_0, \frac{f(i+1)}{f(i)} \leq b$, then $\sum_{i=1}^n f(i) = \Theta(1)$.

Examples

Typical: With $f(i) = 2^i/i$, the ratio between consecutive terms is

$$\frac{f(i+1)}{f(i)} = \frac{2^{i+1}}{i+1} \cdot \frac{i}{2^i} = 2 \cdot \frac{i}{i+1} = 2 \cdot \frac{1}{1 + \frac{1}{i}},$$

which is at least 1.99 for sufficiently large i . Similarly this is true for any $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ with $b^a > 1$.

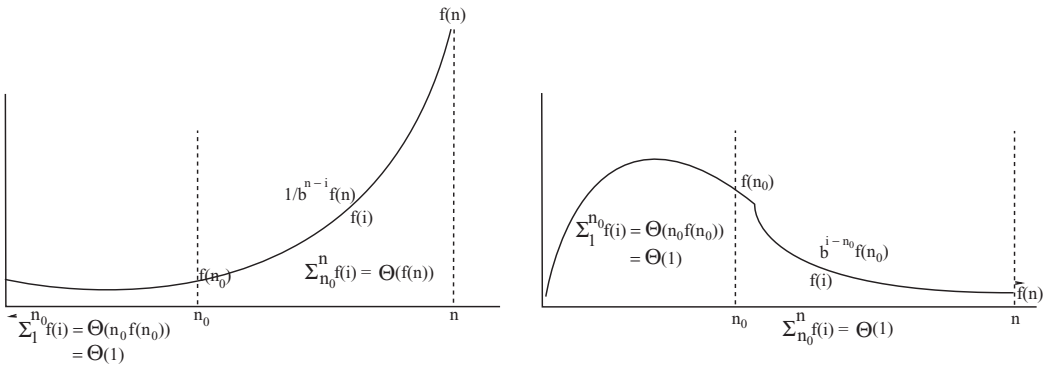


Fig. 38.5 In both graphs, the total before n gets sufficiently large is some constant. On the left, the total for large n is bounded by a growing exponential, and on the right, by a decreasing exponential.

Not Bounded Away: On the other hand, the arithmetic function $f(i) = i$ has a ratio between the terms of $\frac{i+1}{i} = 1 + \frac{1}{i}$. Though this is always bigger than one, it is not bounded away from one by any constant $b > 1$.

Proof: If $\forall i \geq n_0, f(i+1)/f(i) \geq b > 1$, then it follows either by unwinding or induction that

$$f(i) \leq \left(\frac{1}{b}\right)^1 f(i+1) \leq \left(\frac{1}{b}\right)^2 f(i+2) \leq \left(\frac{1}{b}\right)^3 f(i+3) \leq \dots \leq \left(\frac{1}{b}\right)^{n-i} f(n).$$

See Figure 38.5. This gives that

$$\sum_{i=1}^n f(i) = \sum_{i=1}^{n_0} f(i) + \sum_{i=n_0}^n f(i) \leq \Theta(1) + \sum_{i=n_0}^n \left(\frac{1}{b}\right)^{n-i} f(n) \leq \Theta(1) + f(n) \cdot \sum_{i=0}^n \left(\frac{1}{b}\right)^j,$$

which we have already proved is $\Theta(f(n))$.

A Simple Arithmetic Sum: We prove as follows that $\sum_{i=1}^n i = \Theta(n \cdot f(n)) = \Theta(n^2)$:

$$\begin{aligned}
 S &= 1 + 2 + 3 + \cdots + n-2 + n-1 + n, \\
 S &= n + n-1 + n-2 + \cdots + 3 + 2 + 1, \\
 2S &= n+1 + n+1 + n+1 + \cdots + n+1 + n+1 + n+1 \\
 &= n \cdot (n+1), \\
 S &= \frac{1}{2}n \cdot (n+1).
 \end{aligned}$$

Arithmetic Sums: We will now justify the intuition that if half of the terms are roughly the same size, then the total is roughly the number of terms times the last term, namely, $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$.

Theorem: *If for sufficiently large n , the function $f(n)$ is nondecreasing, and $f(\frac{n}{2}) = \Theta(f(n))$, then $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$.*

Examples

Typical: The function $f(n) = n^d$ for $d \geq 0$ is nondecreasing and $f(\frac{n}{2}) = (\frac{n}{2})^d = \frac{1}{2^d}f(n)$. Similarly for $f(n) = \Theta(n^d \cdot \log^e n)$. We consider $-1 < d < 0$ later.

Without the Property: The function $f(n) = 2^n$ does not have this property, because $f(\frac{n}{2}) = 2^{n/2} = \frac{1}{2^{n/2}}f(n)$.

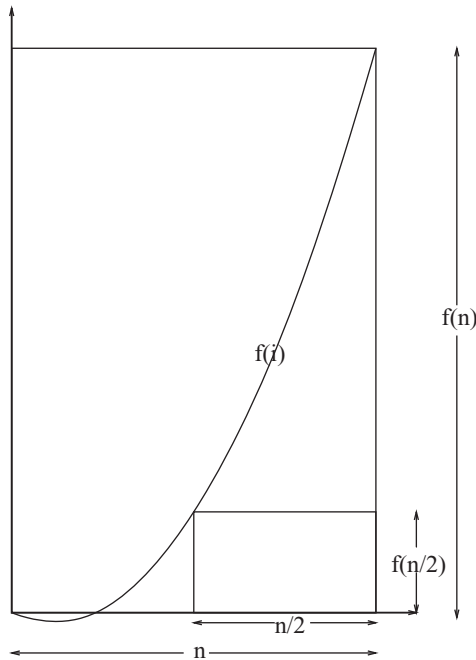


Fig. 38.6 A proof by picture that $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$.

Proof: Because $f(i)$ is nondecreasing, half of the terms are at least the middle term $f(\frac{n}{2})$, and all of the terms are at most the biggest term $f(n)$. Hence, $\frac{n}{2} \cdot f(\frac{n}{2}) \leq \sum_{i=1}^n f(i) \leq n \cdot f(n)$. Because $f(\frac{n}{2}) = \Theta(f(n))$, these bounds match, giving $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ (Figure 38.6).

The Harmonic Sum: The harmonic sum is a famous sum that arises surprisingly often. The total $\sum_{i=1}^n \frac{1}{i}$ is within 1 of $\log_e n$. However, we will not bound it quite so closely.

Theorem: $\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$.

Proof: One way of approximating the harmonic sum is to break it into $\log_2 n$ blocks with 2^k terms in the k th block, and then to prove that the total for each block is between $\frac{1}{2}$ and 1:

$$\sum_{i=1}^n \frac{1}{i} = \underbrace{\frac{1}{1}}_{\leq 1 \cdot \frac{1}{1} = 1} + \underbrace{\frac{1}{2} + \frac{1}{3}}_{\leq 2 \cdot \frac{1}{2} = 1} + \underbrace{\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}}_{\leq 4 \cdot \frac{1}{4} = 1} + \underbrace{\frac{1}{8} + \cdots + \frac{1}{15}}_{\leq 8 \cdot \frac{1}{8} = 1} + \cdots$$

$\geq 1 \cdot \frac{1}{2} = \frac{1}{2}$ $\geq 2 \cdot \frac{1}{4} = \frac{1}{2}$ $\geq 4 \cdot \frac{1}{8} = \frac{1}{2}$ $\geq 8 \cdot \frac{1}{16} = \frac{1}{2}$

From this, it follows that $\frac{1}{2} \cdot \log_2 n \leq \sum_{i=1}^n \frac{1}{i} \leq 1 \cdot \log_2 n$.

Close to Harmonic: We will now use a similar technique to prove the remaining two cases of the adding-made-easy technique.

Theorem: $\sum_{i=1}^n 1/i^{d'}$ is $\Theta(1)$ if $d' > 1$ and is $\Theta(n \cdot f(n))$ if $d' < 1$. (Similarly for $f(n) = \Theta(n^d \cdot \log^e n)$ with $d < -1$ or $d > -1$.)

Proof: As we did with the harmonic sum, we break the sum $\sum_{i=1}^n f(n)$ into blocks where the k th block has the 2^k terms $\sum_{i=2^k}^{2^{k+1}-1} f(i)$. Because the terms are decreasing, the total for the block is at most $F(k) = 2^k \cdot f(2^k)$. The total overall is then at most

$$\sum_{k=0}^{\log_2 n} F(k) = \sum_{k=0}^{\log_2 n} 2^k \cdot f(2^k) = \sum_{k=0}^{\log_2 n} \frac{2^k}{(2^k)^{d'}} = \sum_{k=0}^N \frac{1}{2^{k \cdot (d'-1)}}.$$

If $d' > 1$, then this sum is exponentially decreasing and converges to $\Theta(1)$. If $d' < 1$, then this sum is exponentially increasing and diverges to $\Theta(F(N)) = \Theta(2^{\log_2 n} \cdot f(n)) = \Theta(n \cdot f(n))$.

Functions without the Basic Form: (Warning: This topic may be a little hard.) Until now, we have only considered functions with the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$. We would like to generalize the adding-made-easy technique as follows.

	Geometric increasing	Arithmetic	Harmonic	Bounded tail
If	$f(n) \geq 2^{\Omega(n)}$	$f(n) = n^{\Theta(1)-1}$	$f(n) = \Theta(\frac{1}{n})$	$f(n) \leq n^{-1-\Omega(1)}$
then	$\sum_{i=1}^n f(i) = \Theta(f(n))$	$\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$	$\sum_{i=1}^n f(i) = \Theta(\log n)$	$\sum_{i=1}^n f(i) = \Theta(1)$

Example: Consider $f(n) = n^{8+\frac{1}{n}}$ or $f(n) = n^{-\frac{1}{n}}$. They are bounded between n^{d_1} and n^{d_2} for constants $d_2 \geq d_1 > 0 - 1$, and hence for both we have $f(n) \in n^{\Theta(1)-1}$. Adding-made-easy then gives that $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$, so that $\sum_{i=1}^n i^{8+\frac{1}{i}} = \Theta(n^{9+\frac{1}{n}})$ and $\sum_{i=1}^n i^{-\frac{1}{i}} = \Theta(n^{1-\frac{1}{n}})$.

Counterexample: The goal here is to predict the sum $\sum_{i=1}^n f(i)$ from the value of the last term $f(n)$. We are unable to do this if the terms oscillate like those created with sines, cosines, floors, and ceilings. Exercise 38.2.4 proves that $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$ and $f(n) = 2^{\lfloor \frac{1}{2} \cos(\pi \log_2 n) + 1.5 \rfloor \cdot n}$ are counterexamples for the geometric case and that $f(n) = 2^{2^{\lfloor \log \log n \rfloor}}$ is one for the arithmetic case (Figure 38.7).

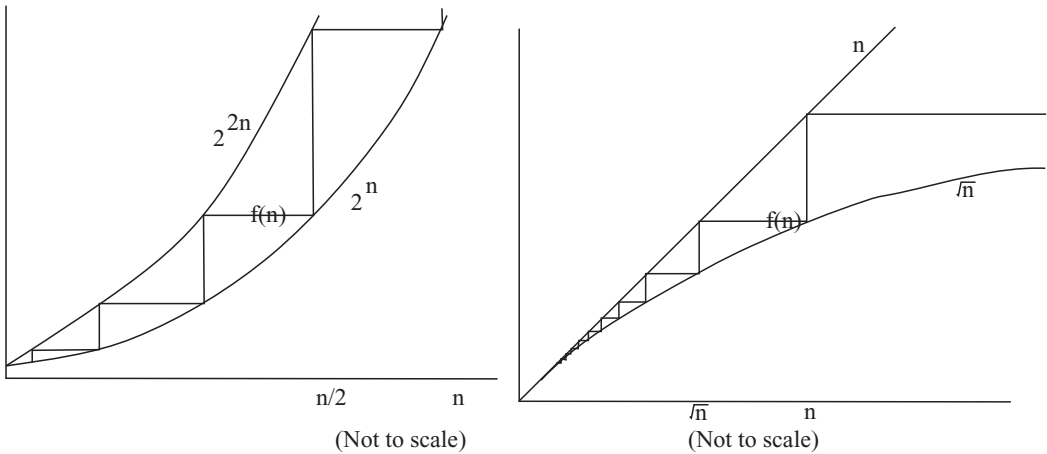


Fig. 38.7 A graph $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$ and that $f(n) = 2^{2^{\lfloor \log \log n \rfloor}}$ which are counter examples to the geometric and the arithmetic cases.

Simple Analytical Functions: We can prove that the adding-made-easy technique works for all functions $f(n)$ that can be expressed with n , real constants, plus, minus, times, divide, exponentiation, and logarithms. Such functions are said to be *simple analytical*.

Proof Sketch: I will only give a sketch of the proof here. For the geometric case, we must prove that if $f(n)$ is simple analytical and $f(n) \geq 2^{\Omega(n)}$, then $\exists b > 1, \exists n_0, \forall n \geq n_0, f(n+1)/f(n) \geq b$. From this, the ratio-between-terms theorem above gives that $\sum_{i=1}^n f(i) = \Theta(f(n))$.

Because the function is growing exponentially, we know that generally it grows at least as fast as b^n for some constant $b > 1$ and hence $f(n+1)/f(n) \geq b$, or equivalently $h(n) = \log f(n+1) - \log f(n) - \log b > 0$ for an infinite number of values for n .

A deep theorem about simple analytical functions is that they cannot oscillate forever and hence can change sign at most a finite number of places. It follows that there must be a last place n_0 at which the sign changes. We can conclude that $\forall n \geq n_0, h(n) > 0$ and hence $f(n+1)/f(n) \geq b$.

The geometrically decreasing case is the same except $f(n+1)/f(n) \leq b$. The arithmetic case is similar except that it proves that if $f(n)$ is simple analytical and $f(n) = n^{\Theta(1)-1}$, then $f(\frac{n}{2}) = \Theta(f(n))$.

Exercise 38.2.1 (See solution in Exercise Solutions.) Zeno's classic paradox is that Achilles is traveling 1 km/hr and has 1 km to travel. First he must cover half his distance, then half of his remaining distance, then half of this remaining distance, ... He never arrives. But Bryan Magee states, "People have found it terribly disconcerting. There must be a fault in the logic, they have said. But no one has yet been fully successful in demonstrating what it is." Resolve this ancient paradox by adding up the time required for all steps.

Exercise 38.2.2 Prove that if $\exists b < 1, \exists n_0, \forall i \geq n_0, f(i+1)/f(i) \leq b$, then $\sum_{i=n_0}^n f(i) = \Theta(f(n_0)) = \Theta(1)$.

Exercise 38.2.3 A seeming paradox is how one could have a vessel that has finite volume and infinite surface area. This (theoretical) vessel could be filled with a small amount of paint but require an infinite amount of paint to paint. For $h \in [1, \infty)$, its cross section at h units from its top is a circle with radius $r = \frac{1}{h^c}$ for some constant c . Integrate (or add up) its cross-sectional circumference to compute its surface area, and integrate (or add up) its cross-sectional area to compute its volume. Give a value for c such that its surface area is infinite and its volume is finite.

Exercise 38.2.4 (See solution in Exercise Solutions.)

1. For $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$, prove that $f(n) \geq 2^{\Omega(n)}$,
2. and that $\sum_{i=1}^n f(i) \neq \Theta(f(n))$.
3. For $f(n) = 2^{2^{\lceil \log \log n \rceil}}$, prove that $f(n) = n^{\Theta(1)-1}$,
4. and that $\sum_{i=1}^n f(i) \neq \Theta(n \cdot f(n))$.
5. Plot $f(n) = 2^{\lfloor \frac{1}{2} \cos(\pi \log_2 n) + 1.5 \rfloor \cdot n}$, and prove that it is also a counterexample for the geometric case.

39 Recurrence Relations

A wise man told the king to give him one grain of rice for the first square of a chessboard, and for each remaining square to give him twice the number for the previous square. Thirty-two days later, the king realized that there is not enough rice in all of world to reward him. The number of grains on the n th square is given by the recurrence relation $T(1) = 1$ and $T(n) = 2T(n - 1)$.

The algebraic equation $x^2 = x + 2$ specifies the value of an unknown real that must be found. The differential equation $\frac{\delta f(x)}{\delta x} = f(x)$ specifies functions from reals to reals that must be found. Similarly, a recurrence relations like $T(n) = 2 \times T(n - 1)$ specifies functions from integers to reals. One way to solve each of these is to guess a solution and check to see if it works. Here, $T(n) = 2^n$ works, i.e., $2^n = 2 \times 2^{n-1}$. However, $T(n) = c \cdot 2^n$ also works for each value of c . Making the further requirement that $T(1) = 1$ narrows the solution set to only $T(n) = \frac{1}{2} \cdot 2^n = 2^{n-1}$.

39.1 The Technique

Timing of Recursive Programs: Recursive relations are used to determine the running time of recursive programs. (See [Chapter 11](#).) For example, if a routine, when given an instance of size n , does $f(n)$ work itself and then recurses a times on subinstances of size $\frac{n}{b}$, then the running time is $T(n) = a \cdot T(\frac{n}{b}) + f(n)$.

See [Section 11.6](#) to learn more about the *tree of stack frames*. Each stack frame consists of one execution of the routine on a single instance, ignoring subroutine calls. The top-level stack frame is called by the user on the required input instance. It recurses on a number of subinstances, creating the next level of stack frames. These in turn recurse again until the instance is sufficiently small that the stack frame returns without recursing. These final stack frames are referred to as *base cases*.

Let $T(n)$ denote the number of “Hi”s that the entire tree of stack frames, given the following code, prints on an instance of size n . The top level stack frame prints “Hi” $f(n)$ times. It then recurses a times on subinstances of size $\frac{n}{b}$. If $T(n)$ is the number of “Hi”s for instances of size n , then it follows that $T(\frac{n}{b})$ is the number for instances of size $\frac{n}{b}$. Repeating this a times will take time $a \cdot T(\frac{n}{b})$. It follows that the total number satisfies the recursive relation $T(n) = a \cdot T(\frac{n}{b}) + f(n)$. The goal of this section is to determine which function $T(n)$ satisfies this relation.

If instead the routine recurses a times on instances of size $n - b$, then the related recurrence relation will be $T(n) = a \cdot T(n - b) + f(n)$.

algorithm $Eg(I_n)$

<pre-cond>: I_n is an instance of size n .

<post-cond>: Prints $T(n)$ “Hi”s.

begin

$n = |I_n|$

if($n \leq 1$) then

put “Hi”

else

loop $i = 1 \dots f(n)$

put “Hi”

end loop

loop $i = 1 \dots a$

$I_{\frac{n}{b}}$ = an input of size $\frac{n}{b}$

$Eg(I_{\frac{n}{b}})$

end loop

end if

end algorithm

When the input has size zero or one, only one “Hi” is printed. In general, we will assume that recursive programs spend $\Theta(1)$ time for instances of size $\Theta(1)$. We express this as $T(1) = 1$, or more generally as $T(\Theta(1)) = \Theta(1)$.

Solving Recurrence Relations: Consider $T(n) = a \cdot T(\frac{n}{b}) + f(n)$, where $f(n) = \Theta(n^c \cdot \log^d n)$ or $f(n) = 0$.

$\frac{\log a}{\log b}$ vs c	d	Dominated by	$T(n)$	Example ($\frac{\log_3 9}{\log_3 3} = 2$)	Solution
<	Any	Top level	$\Theta(f(n))$	$T(n) = 9 \cdot T(\frac{n}{3}) + n^4$	$\Theta(n^4)$
=	> -1	All levels	$\Theta(f(n) \log n)$	$T(n) = 9 \cdot T(\frac{n}{3}) + n^2$	$\Theta(n^2 \log n)$
	< -1	Base cases	$\Theta\left(n^{\frac{\log a}{\log b}}\right)$	$T(n) = 9 \cdot T(\frac{n}{3}) + \frac{n^2}{\log^2 n}$	$\Theta(n^2)$
>	Any			$t(n) = 9 \cdot T(\frac{n}{3})$	

Consider $T(n) = a \cdot T(n - b) + f(n)$, where $f(n) = \Theta(n^c \cdot \log^d n)$ or $f(n) = 0$.

a	$f(n)$	Dominated by	$T(n)$	Example	Solution
> 1	Any	Base cases	$\Theta(a^{\frac{n}{b}})$	$T(n) = 9 \cdot T(n - 3) + n^4$	$\Theta(9^{\frac{n}{3}})$
=	≥ 1	All levels	$\Theta(n \cdot f(n))$	$T(n) = T(n - 3) + n^4$	$\Theta(n^5)$
	= 0	Base cases	$\Theta(1)$	$T(n) = T(n - 3)$	$\Theta(1)$

A Growing Number of Subinstances of Shrinking Size: Each instance having a subinstances means that the number of subinstances grows exponentially by a factor

of a . On the other hand, the sizes of the subinstances shrink exponentially by a factor of b . The amount of work that the instance must do is the function f of this instance size. Whether the growing or the shrinking dominates this process depends upon the relationship between a, b , and $f(n)$.

Dominated By: When total work $T(n)$ done in the tree of stack frames is dominated by the work $f(n)$ done by the top stack frame, we say that the work is *dominated by the top level* of the recursion. The solution in this case will be $T(n) = \Theta(f(n))$. Conversely, we say that it is *dominated by the base cases* when the total is dominated by the sum of the work done by the base cases. Because each base case does only a constant amount of work, the solution will be $T(n) = \Theta(\# \text{ of base cases})$, which is $\Theta(n^{\log a / \log b})$, $\Theta(a^{\frac{n}{b}})$, or $\Theta(1)$ in the above examples. Finally, if the amounts of work at the different levels of recursion are sufficiently close to each other, then we say that the total work is *dominated by all the levels* and the total is the number of levels times this amount of work, namely, $T(n) = \Theta(\log n \cdot f(n))$ or $\Theta(n \cdot f(n))$.

The Ratio $\frac{\log a}{\log b}$: See [Chapter 36](#) for a discussion about logarithms. One trick that it gives us is that when computing the ratio between two logarithms, the base used does not matter, because changing the base will introduce the same constant both on the top and the bottom, which will cancel. Hence, when computing such a ratio, you can choose whichever base makes the calculation the easiest. For example, to compute $\frac{\log 16}{\log 8}$, the obvious base to use is 2, because $\frac{\log_2 16}{\log_2 8} = \frac{4}{3}$. This is useful in giving that $T(n) = 16 \cdot T(\frac{n}{8}) + f(n) = \Theta(n^{\log 16 / \log 8}) = \Theta(n^{4/3})$. On the other hand, to compute $\frac{\log 9}{\log 27}$, the obvious base to use is 3, because $\frac{\log_3 9}{\log_3 27} = \frac{2}{3}$, and hence we have $T(n) = 9 \cdot T(\frac{n}{27}) + f(n) = \Theta(n^{2/3})$. Another interesting fact given is that $\log 1 = 0$, which gives that $T(n) = 1 \cdot T(\frac{n}{2}) + f(n)$, $T(n) = \Theta(n^{\log 1 / \log 2}) = \Theta(n^0) = \Theta(1)$.

Exercise 39.1.1 (See solution in Exercise Solutions.) Give solutions for the following examples:

1. $T(n) = 2T(\frac{n}{2}) + n$.
2. $T(n) = 2T(\frac{n}{2}) + 1$.
3. $T(n) = 4T(\frac{n}{2}) + \Theta(\frac{n^3}{\log^3 n})$.
4. $T(n) = 32T(\frac{n}{4}) + \Theta(\log n)$.
5. $T(n) = 27T(\frac{n}{3}) + \Theta(n^3 \log^4 n)$.
6. $T(n) = 8T(\frac{n}{4}) + \Theta((\frac{n}{\log n})^{1.5})$.
7. $T(n) = 4T(\frac{n}{2}) + \Theta(\frac{n^2}{\log n})$.

Exercise 39.1.2 Give solutions for the following stranger examples:

1. $T(n) = 4T(\frac{n}{2}) + \Theta(n^3 \log \log n)$.
2. $T(n) = 4T(\frac{n}{2}) + \Theta(2^n)$.
3. $T(n) = 4T(\frac{n}{2}) + \Theta(\log \log n)$.
4. $T(n) = 4T(\frac{n}{2} - \sqrt{n} + \log n - 5) + \Theta(n^3)$.

39.2 Some Proofs

I now present a few of the classic techniques for computing recurrence relations. As our example we will solve $T(n) = GT(n/0) + f(n)$, for $f(n) = n^c$.

Guess and Verify: To begin, consider the example $T(n) = 4T(\frac{n}{2}) + n$ and $T(1) = 1$.

Plugging In: If we can guess $T(n) = 2n^2 - n$, the first way to verify that this is the solution is to simply plug it into the two equations and make sure that they are satisfied.

Left side	Right side
$T(n) = 2n^2 - n$	$4T(\frac{n}{2}) + n = 4 \left[2(\frac{n}{2})^2 - (\frac{n}{2}) \right] - n = 2n^2 - n$
$T(1) = 2n^2 - n = 1$	1

Proof by Induction: Similarly, we can use induction to prove that this is the solution for all n (at least for $n = 2^i$).

Base Case: Because $T(1) = 2(1)^2 - 1 = 1$, it is correct for $n = 2^0$.

Induction Step: Let $n = 2^i$. Assume that it is correct for $2^{i-1} = \frac{n}{2}$. Because $T(n) = 4T(\frac{n}{2}) + n = 4 \left[2(\frac{n}{2})^2 - (\frac{n}{2}) \right] + n = 2n^2 - n$, it is also true for n .

Calculate Coefficients: Suppose that instead we are only able to guess that the formula has the form $T(n) = an^2 + bn + c$ for some constants a, b , and c .

Left side	Right side
$T(n) = an^2 + bn + c$	$4T(\frac{n}{2}) + n = 4 \left[a(\frac{n}{2})^2 + b(\frac{n}{2}) + c \right] - n = an^2 + (2b + 1)n + 4c$
$T(1) = a + b + c$	1

These left and right sides must be equal for all n . Both have a as the coefficient of n^2 , which is good. To make the coefficient in front n be the same, we need that $b = 2b + 1$, which gives $b = -1$. To make the constant coefficient be the same, we need that $c = 4c$, which gives $c = 0$. To make $T(1) = a(1)^2 + b(1) + c = a(1)^2 - (1) + 0 = 1$, we need that $a = 2$. This gives us the solution $T(n) = 2n^2 - n$ that we had before.

Calculate Exponent: If we were to guess that $a \cdot T(\frac{n}{b})$ is much bigger than $f(n)$, then $T(n) = a \cdot T(\frac{n}{b}) + f(n) \approx a \cdot T(\frac{n}{b})$. Further, we guess that $T(n) = n^\alpha$ for some constant α . Plugging this into $T(n) = a \cdot T(\frac{n}{b})$ gives $n^\alpha = a \cdot (\frac{n}{b})^\alpha$, or $b^\alpha = a$. Taking the log gives $\alpha \cdot \log b = \log a$, and solving gives $\alpha = \frac{\log a}{\log b}$. In conclusion, $T(n) = \Theta(n^{\log a / \log b}) = \Theta(n^{\log 4 / \log 2}) = \Theta(n^2)$.

Unwinding: A useful technique is to unwind a recursive relation for a few steps and to look for a pattern:

$$\begin{aligned}
 T(n) &= f(n) + a \cdot T\left(\frac{n}{b}\right) = f(n) + a \cdot \left[f\left(\frac{n}{b}\right) + a \cdot T\left(\frac{n}{b^2}\right) \right] \\
 &= f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot T\left(\frac{n}{b^2}\right) = f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot \left[f\left(\frac{n}{b^2}\right) + a \cdot T\left(\frac{n}{b^3}\right) \right] \\
 &= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + a^3 \cdot T\left(\frac{n}{b^3}\right) = \dots \\
 &= \sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right) + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^h a^i \cdot f\left(\frac{n}{b^i}\right)\right).
 \end{aligned}$$

Filling the Table: My recommended way to evaluate recursive relations is to fill out a table like that in [Table 39.1](#).

(a) Number of Stack Frames at the i th Level: Level 0 contains the one initial stack frame at the top of the tree of stack frame. It recursively calls a times. Hence, level 1 has a stack frames. Each of these recursively calls a times, giving a^2 stack frames at level 2. Each successive level, the number of stack frames goes up by a factor of a , giving a^i at level i .

(b) Size of Instance at the i th Level: The top stack frame at level 0 is given an instance of size n . It recurses on a subinstance of size $\frac{n}{b}$. Stack frames at level 1, given instances of size $\frac{n}{b}$, recurse on subinstances of size n/b^2 . Each successive level decreases the instance size by a factor of b , giving size n/b^i at level i .

(c) Time within One Stack Frame: On an instance of size n , a single stack frame requires $f(n)$ time. Hence, a stack frame at the i th level, with an instance of size n/b^i , requires $f(n/b^i)$ time.

(d) Number of Levels: The recursive program stops recursing when the instance becomes sufficiently small, say, of size 0 or 1. Let h denote the level at which this occurs. We have seen that the instances at level h have size n/b^h . Setting $n/b^h = 1$ and solving for h gives $h = \frac{\log n}{\log b}$.

(e) Number of Base Case Stack Frames: The number of stack frames at level i is a^i . Hence, the number of base case stack frames is $a^h = a^{\log n / \log b}$. Though this looks ugly, [Chapter 36](#) gives $a^{\log n / \log b} = (2^{\log a})^{\log n / \log b} = 2^{\log a \cdot \log n / \log b} = (2^{\log n})^{\log a / \log b} = n^{\log a / \log b}$. Given that $\frac{\log a}{\log b}$ is simply some constant, $n^{\log a / \log b}$ is a simple polynomial in n .

(f) $T(n)$ as a Sum: There are a^i stack frames at level i , and each requires $f(n/b^i)$ time, for a total of $a^i \cdot f(n/b^i)$ at the level. We obtain the total time $T(n)$ for the recursion by summing the times at all of these levels. This gives

$$T(n) = \left[\sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right) \right] + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^h a^i \cdot f\left(\frac{n}{b^i}\right)\right).$$

Table 39.1. Solving $T(n) = a \cdot T(\frac{n}{b}) + 2(n^c)$ by filling in the table.

Example	$T(n) = 4T(n/2) + n$	$T(n) = 9T(n/3) + n^2$	$T(n) = 2T(n/4) + n^2$
(a) No. of frames at the i th level	4^i	9^i	2^i
(b) Instance size at i th level	$\frac{n}{2^i}$	$\frac{n}{3^i}$	$\frac{n}{4^i}$
(c) Time within one stack frame	$f(\frac{n}{2^i}) = (\frac{n}{2^i})$	$f(\frac{n}{3^i}) = (\frac{n}{3^i})^2$	$f(\frac{n}{4^i}) = (\frac{n}{4^i})^2$
(d) No. of levels	$\frac{n}{2^h} = 1$ $h = \frac{\log n}{\log 2} = \Theta(\log n)$	$\frac{n}{3^h} = 1$ $h = \frac{\log n}{\log 3} = \Theta(\log n)$	$\frac{n}{4^h} = 1$ $h = \frac{\log n}{\log 4} = \Theta(\log n)$
(e) No. of base case stack frames	$4^h = 4^{\frac{\log n}{\log 2}}$ $= n^{\frac{\log 4}{\log 2}} = n^2$	$9^h = 9^{\frac{\log n}{\log 3}}$ $= n^{\frac{\log 9}{\log 3}} = n^2$	$2^h = 2^{\frac{\log n}{\log 4}}$ $= n^{\frac{\log 2}{\log 4}} = n^{\frac{1}{2}}$
(f) $T(n)$ as a sum	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{\Theta(\log n)} 4^i \cdot (\frac{n}{2^i})$ $= n \cdot \sum_{i=0}^{\Theta(\log n)} 2^i$	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{\Theta(\log n)} 9^i \cdot (\frac{n}{3^i})^2$ $= n^2 \cdot \sum_{i=0}^{\Theta(\log n)} 1$	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{\Theta(\log n)} 2^i \cdot (\frac{n}{4^i})^2$ $= n^2 \cdot \sum_{i=0}^{\Theta(\log n)} (\frac{1}{8})^i$
(g) Dominated by?	Geometric increase: base cases	Arithmetic sum: all levels	Geometric decrease: top level
(h) $\Theta(T(n))$	$T(n) = \Theta(n^{\log a / \log b})$ $= \Theta(n^{\frac{\log 4}{\log 2}}) = \Theta(n^2)$	$T(n) = \Theta(f(n) \log n)$ $= \Theta(n^2 \log n)$	$T(n) = \Theta(f(n))$ $= \Theta(n^2)$

Plugging in $f(n) = n^c$ gives

$$T(n) = \Theta \left(\sum_{i=0}^h a^i \cdot \left(\frac{n}{b^i} \right)^c \right) = \Theta \left(n^c \cdot \sum_{i=0}^h \left(\frac{a}{b^c} \right)^i \right).$$

(g) Dominated By: The key things to remember about this sum are that it has $\Theta(\log n)$ terms, the top term being $a^0 f(n/b^0) = f(n)$ and the base case term being $a^n f(n/b^n) = a^{\log n / \log b} f(n) = n^{\log a / \log b} \Theta(n^{\log a / \log b})$. According to the adding-made-easy approximations given in [Chapter 38](#), if either the top term or the base case term is sufficiently bigger than the other, then the total is dominated by this term. On the other hand, if they are roughly the same, then the total is approximately the number of terms times a typical term.

(h) Evaluating the Sum: If $\frac{\log a}{\log b} < c$, then $a/b^c < 1$, giving that the terms in $T(n) = \Theta(n^c \cdot \sum_{i=0}^h (\frac{a}{b^c})^i)$ decrease exponentially, giving $T(n) = \Theta(\text{top term}) = \Theta(f(n))$. Similarly, if $\frac{\log a}{\log b} > c$, then the terms increase exponentially, giving $T(n) = \Theta(\text{base case term}) = \Theta(n^{\log a / \log b})$. If $\frac{\log a}{\log b} = c$, then $\frac{a}{b^c} = 1$, giving $T(n) = \Theta(n^c \cdot \sum_{i=0}^h (\frac{a}{b^c})^i) = \Theta(n^c \cdot \sum_{i=0}^h 1) = \Theta(n^c \cdot h) = \Theta(f(n) \log n)$.

Exercise 39.2.1 (See solution in Exercise Solutions.) Solve the famous Fibonacci recurrence relation $Fib(0) = 0$, $Fib(1) = 1$, and $Fib(n) = Fib(n-1) + Fib(n-2)$ by plugging in $Fib(n) = \alpha^n$ and solving for α .

Exercise 39.2.2 (See solution in Exercise Solutions.) Solve the following by unwinding them:

1. $T(n) = T(n-1) + n$.
2. $T(n) = 2 \cdot T(n-1) + 1$.

Exercise 39.2.3 Does setting the size of the base case to 5 have any practical effect? How about setting the size to zero, i.e., $n/b^h = 0$?

Why does this happen? If instead instances at the i th level had size $n - AB$, would an instance size of 0, 1, or 2 be better? How many levels h are there?

Exercise 39.2.4 (See solution in Exercise Solutions.) [Section 39.2](#) solves $T(n) = aT(n/b) + f(n)$ for $f(n) = n^c$. If $f(n) = n^c \log^d n$ and $\frac{\log a}{\log b} = c$, then the math is harder. Compute the sum for $d > -1$, $d = -1$, $d < -1$. (Hint: Reverse the order of the terms.)

Exercise 39.2.5 Use the method in [Table 39.2](#) to compute each of the following recursive relations:

1. $T(n) = nT(n-1) + 1$.
2. $T(n) = 2T(\sqrt{n}) + n$.
3. $T(n) = T(u \cdot n) + T(v \cdot n) + \Theta(n)$ where $u + v = 1$.

Table 39.2. Solving $T(n) = a \cdot T(n - b) + 2(n^c)$ by filling in the table.

Example	$T(n) = aT(n - b) + n^c$	$T(n) = T(n - b) + n^c$	$T(n) = T(n - b) + 0$
(a) No. of frames at the i th level	a^i		1
(b) Instance size at i th level		$n - i \cdot b$	
(c) Time within one stack frame		$f(n - i \cdot b) = (n - i \cdot b)^c$	$f(n - i \cdot b) = 0$ except for the base case, which has work $\Theta(1)$
(d) No. of levels		$n - h \cdot b = 0, h = \frac{n}{b}$ Having a base case of size zero makes the math the cleanest.	
(e) No of base case stack frames	$a^h = a^{\frac{1}{b}n}$		1
(f) $T(n)$ as a sum	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{n/b} a^i \cdot (n - i \cdot b)^c$	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{n/b} 1 \cdot (n - i \cdot b)^c$	$\sum_{i=0}^h (\# \text{ at level}) \cdot (\text{time each})$ $= \left(\sum_{i=0}^{n/b-1} 1 \cdot 0 \right) + 1 \cdot \Theta(1)$ $= \Theta(1)$
(g) Dominated by	Geometric increase: base cases	Arithmetic sum: all levels	Geometric decrease: base cases
(h) $\Theta(T(n))$	$T(n) = \Theta(a^{n/b})$	$T(n) = \Theta\left(\frac{n}{b} \cdot n^c\right)$ $= \Theta(n^{c \Downarrow 1})$	$T(n) = \Theta(1)$

Exercise 39.2.6 Running time:

algorithm *Careful*(n)

<pre-cond>: n is an integer.

<post-cond>: $Q(n)$ “Hi”s are printed for some odd function Q .

begin

 if ($n \leq 1$)

 PrintHi(1)

 else

 loop $i = 1 \dots n$

 PrintHi(i)

 end loop

 loop $i = 1 \dots 8$

 Careful($\frac{n}{2}$)

 end loop

 end if

end algorithm

algorithm *PrintHi*(n)

<pre-cond>: n is an integer.

<post-cond>: n^2 “Hi”s are printed

begin

 loop $i = 1 \dots n^2$

 Print(“Hi”)

 end loop

end algorithm

1. Give and solve the recurrence relation for the number of “Hi”s, $Q(n)$. Show your work. In a sentence or two, give the intuition.
2. What is the running time (time complexity) of this algorithm as a function of the size of the input?

40 A Formal Proof of Correctness

Though I mean not to be too formal, it is useful to at least understand the required steps in a formal proof of correctness.

Specifications: Before we prove that an algorithm is correct, we need to know precisely what it is supposed to do.

Preconditions: Assertions that are promised be true about the input instance.

Postconditions: Assertions that must be true about the output.

Correctness: Consider some instance. If this instance meets the preconditions, then after the code has been run, the output must meet the postconditions:

$$\langle pre-cond \rangle \ \& \ code_{alg} \Rightarrow \langle post-cond \rangle.$$

The correctness of an algorithm is only with respect to the stated specifications. It does not guarantee that it will work in situations that are not taken into account by this specification.

Breaking the Computation Path into Fragments: The method to prove that an algorithm is correct is as follows. Assertions are inserted into the code to act as checkpoints. Each assertion is a statement about the current state of the computation's data structures that is either true or false. If it is false, then something has gone wrong in the logic of the algorithm. These assertions break the path of the computation into fragments. For each such fragment, we prove that if the assertion at the beginning of the fragment is true and the fragment gets executed, then the assertion at the end of the fragment will be true. Combining all these fragments back together gives that if the first assertion is true and the entire computation is executed, then the last assertion will be true.

A Huge Number of Paths: There are likely an exponential number or even an infinite number of different paths that the computation might take, depending on the input instance and the tests that occur along the way. In contrast, there are not many different computation path fragments. Hence, it is much easier to prove the correctness of each fragment than of each path.

The following table outlines the computational path fragments that need to be tested for different code structures.

Single line of code:

$\langle \text{pre-assignment-cond} \rangle$:	The variables x and y have meaningful values.
$z = x + y$	
$\langle \text{post-assignment-cond} \rangle$:	The variable z takes on the sum of the value of x and the value of y . The previous value of z is lost.

Blocks of code:

$\langle \text{assertion}_0 \rangle$	
code_1	$[\langle \text{assertion}_0 \rangle \ \& \ \text{code}_1 \Rightarrow \langle \text{assertion}_1 \rangle]$
$\langle \text{assertion}_1 \rangle$	$[\langle \text{assertion}_1 \rangle \ \& \ \text{code}_2 \Rightarrow \langle \text{assertion}_2 \rangle]$
code_2	$\Rightarrow [\langle \text{assertion}_0 \rangle \ \& \ \text{code}_{1\&2} \Rightarrow \langle \text{assertion}_2 \rangle]$
$\langle \text{assertion}_2 \rangle$	

If statements:

$\langle \text{pre-if-cond} \rangle$	
if($\langle \text{test} \rangle$) then	
$\text{code}_{\text{true}}$	$[\langle \text{pre-if-cond} \rangle \ \& \ \langle \text{test} \rangle \ \& \ \text{code}_{\text{true}} \Rightarrow \langle \text{post-if-cond} \rangle]$
	$[\langle \text{pre-if-cond} \rangle \ \& \ \neg \langle \text{test} \rangle \ \& \ \text{code}_{\text{false}} \Rightarrow \langle \text{post-if-cond} \rangle]$
else	
$\text{code}_{\text{false}}$	
end if	$\Rightarrow [\langle \text{pre-if-cond} \rangle \ \& \ \text{code} \Rightarrow \langle \text{post-if-cond} \rangle]$
$\langle \text{post-if-cond} \rangle$	

Loops:

$\langle \text{pre-loop-cond} \rangle$	
loop	
$\langle \text{loop-invar} \rangle$	$[\langle \text{pre-loop-cond} \rangle \Rightarrow \langle \text{loop-invar} \rangle]$
	$[\langle \text{loop-invar}' \rangle \ \& \ \neg \langle \text{exit-cond} \rangle \ \& \ \text{code}_{\text{loop}} \Rightarrow \langle \text{loop-invar}'' \rangle]$
exit when $\langle \text{exit-cond} \rangle$	$[\langle \text{loop-invar} \rangle \ \& \ \langle \text{exit-cond} \rangle \Rightarrow \langle \text{post-loop-cond} \rangle]$
	Termination
$\text{code}_{\text{loop}}$	$\Rightarrow [\langle \text{pre-loop-cond} \rangle \ \& \ \text{code} \Rightarrow \langle \text{post-loop-cond} \rangle]$
end loop	
$\langle \text{post-loop-cond} \rangle$	

Function call:

$\langle \text{pre-call-cond} \rangle$	
$\text{output} = \text{Func}(\text{input})$	$[\langle \text{pre-call-cond} \rangle \Rightarrow \langle \text{pre-cond} \rangle_{\text{Func}}]$
	$[\langle \text{post-cond} \rangle_{\text{Func}} \Rightarrow \langle \text{post-call-cond} \rangle]$
$\langle \text{post-call-cond} \rangle$	$\Rightarrow [\langle \text{pre-call-cond} \rangle \ \& \ \text{code} \Rightarrow \langle \text{post-call-cond} \rangle]$

41 Additional Exercises: Part IV

41.1 Existential and Universal Quantifiers

IV.1. **Logic:** α and β are each statements that are either true or false. For each of the following theorems, explain what it means, argue that it is true, and give an example.

- For each rule in Figure 41.1, give an example, explain what it means, and informally argue that it is true. Also give an example in which $\alpha \rightarrow \beta$ is true and $\beta \rightarrow \alpha$ is not.

Proof Techniques/Lemmas					
	From:	Using:	Conclude:	From:	Proving:
And $\wedge$: Both are true		Separating And $\alpha \wedge \beta$	$\alpha \wedge \beta$		Eval/Build $\wedge$ $\alpha \wedge \beta$ $\alpha \wedge \beta$ $\neg \beta$ $\neg(\beta \wedge \gamma)$
Or $\vee$: At least one is true		Selecting Or $\alpha \vee \beta$ & $\neg \alpha$	β		Eval/Build $\vee$ α $\alpha \vee \gamma$ $\neg \alpha$ & $\neg \beta$ $\neg(\alpha \vee \beta)$
		Cases $\alpha \vee \beta, \alpha \rightarrow \gamma, \text{ \& } \beta \rightarrow \gamma$	γ		Excluded Middle $\alpha \vee \neg \alpha$ & $\neg(\alpha \wedge \neg \alpha)$
Implies $\rightarrow$: If LHS is true, then so is RHS.		Modus Ponens α & $\alpha \rightarrow \beta$	β		Deduction Assume α , prove β $\alpha \rightarrow \beta$
		Cases $\alpha \vee \beta, \alpha \rightarrow \gamma, \text{ \& } \beta \rightarrow \gamma$	γ		Eval/Build $\rightarrow$ β $\gamma \rightarrow \beta$ $\neg \alpha$ $\alpha \rightarrow \gamma$ α & $\neg \beta$ $\neg(\alpha \rightarrow \beta)$
		Equivalence $\alpha \rightarrow \beta$ & $\beta \rightarrow \alpha$	$\alpha \text{ iff } \beta$		Contrapositive $\alpha \rightarrow \beta \text{ iff } \neg \beta \rightarrow \neg \alpha \text{ iff } \neg \alpha \vee \beta$
Negation $\neg$: Flips truth		Transitivity $\alpha \rightarrow \beta$ & $\beta \rightarrow \gamma$	$\alpha \rightarrow \gamma$		De Morgan's Law $\neg(\alpha \wedge \beta) \text{ iff } \neg \alpha \vee \neg \beta$

Double Negation: $\neg \neg \alpha \text{ iff } \alpha$ By Contradiction: From $\neg \alpha \rightarrow \beta$, and $\neg \alpha \rightarrow \neg \beta$, conclude α .
 Commutative: $\alpha \vee \beta \text{ iff } \beta \vee \alpha$ and $\alpha \wedge \beta \text{ iff } \beta \wedge \alpha$ Inconsistent: From β and $\neg \beta$, conclude anything.
 Distributive: $\gamma \wedge (\alpha \vee \beta) \text{ iff } (\gamma \wedge \alpha) \vee (\gamma \wedge \beta)$ and $\gamma \vee (\alpha \wedge \beta) \text{ iff } (\gamma \vee \alpha) \wedge (\gamma \vee \beta)$

Fig. 41.1

- Prove the other rules in the figure using the brute force (table) proof.
- It can be fun to assume that a subset of these rules are true and from them prove that others are true.
- I just don't understand my dad. Maybe if I do well at school, he will love me. Maybe if I follow all his rules, he will love me. I know that at least one

of these is true. I just don't know which. What do I need to do to guarantee his love? Let α denote "I do well at school," β denote "I follow all his rules," γ denote "He will love me," and X denote "The situation", i.e., $(\alpha \rightarrow \gamma) \vee (\beta \rightarrow \gamma)$.

- (a) Suppose I believed that I do not need to both do well at school and follow his rules so I only do one of them. Does this guarantee his love? Give a True/False value of all the variables and statements in which it does not work out for me.
- (b) To be careful, I both do well at school and follow his rules. Prove that this guarantees his love.

IV.2. Computable: A computational problem is a function P from inputs I to required output $P(I)$. An algorithm is a function A from inputs I to actual output $A(I)$. $\text{Time}(A, I)$ gives the running time for algorithm A on input I . Let $T(n)$ be some function.

- (a) Use universal and existential quantifiers to express the following (in standard form):
 - i. A computes P .
 - ii. A does not compute P .
 - iii. P is computable.
 - iv. P is not computable.
 - v. Problem P is computable in time $T(n)$. (No BigOh.)
 - vi. Problem P is not computable in time $T(n)$.
 - vii. P is computable in polynomial time.
 - viii. P is not computable in polynomial time.
 - ix. The computational class "Exponential Time" is strictly bigger than the computational class "Polynomial Time," i.e., something can be done in "Exponential Time" that cannot be done in "Polynomial Time." (Be careful how you use c and n .)
 - x. Let $\text{Loves}(p_1, p_2, t)$ state that person p_1 loves person p_2 at time t . State that at some point in time we all love in vain.
- (b) How does the adversarial game proceed to prove all the statements in (a)?
- (c) The computational problem HALT is an uncomputable problem. (It asks if on input I , when viewed as a program, halts on input 0.) Prove the following true or false: $\forall I \exists A \ A(I) = \text{HALT}(I)$.

IV.3. Clauses: A very useful skill is to express math concepts as first order logic statements.

For example, "Integers have negations" is expressed with $\forall x \exists y \ x+y=0$.

- (a) Yes Instance:
 - i. Define a first order logic statement $\Phi_{3\text{USAT}}(I)$ to state that "Instance I is a Yes instance to the 3-unique-SAT problem."
 - Let I denote an instance to 3-unique-SAT problem, namely, clauses.
 - Let C denote one of the clauses of I .
 - Let x denote one of the variables of I .

- Let S denote a potential solution of I , namely, as setting of each variables to True/False.
 - Let $Valid(I, C, x, S)$ state that when setting the variable according to S , clause C of I is satisfied by variable x (or its negation).
 - Let $\exists$ be an quantifier that states that some value exists uniquely.
- Give this first order logic statement $\Phi_{3USAT}(I)$.
- ii. Define a first order logic statement $\Phi_{AK}(I')$ to state that
- “Instance I' is a Yes instance to the Aisled-Knapsack problem.”
- Here $I' = IM(I)$ is the instant map defined above.
- Let $Valid'(I', i', x', S')$ state that when considering solution S' of I' , row x' contributes a one to the i^{th} bit of the sum of volume/price.
- Give first order logic statement $\Phi_{AK}(I')$.
- (b) In order to proving the above reduction, one needs to prove the following. Explain.
- the above reduction needs to prove the following:
- $\Phi = \forall I [\Phi_{3USAT}(I) \text{ iff } \Phi_{AK}(IM(I))]$.
- Hint: The name of the problem is 3-unique-SAT.
- This problem has two special numbers.
- 3
- unique = 1
- Use these numbers in your answer!

41.2 Time Complexity

- IV.4. **Computing Pi:** Consider the following computational problem. The input is an n bit integer x . The output is $\pi(x)$ which is the x^{th} digit of real number $\pi = 3.14\dots$ written in decimal. What is the time complexity of this problem under each of the following conditions:
- (a) It takes *Time* $1/\epsilon$ to get π to accuracy within ϵ .
 - (b) It takes $\Theta(1)$ time to get the next bit of accuracy of π .
 - (c) Suppose you have a black box oracle to which you pass in integer i and it passes you the i^{th} bit (not digit) of π . How many oracle calls do you need to make and what is the time complex of computing all of these i values?
- IV.5. **Worst Case:** Suppose the input is a graph. Is the worst case an input one with lots of edges or one with only a few edges? It depends. The key factors are how the running time and the input size depend on the number of nodes N and the number of edges E .

If the input lists the nodes and the edges, then the size of the input will be $n = \Theta((N + E) \log N)$ because each node and each edge will require $\Theta(\log N)$ bits to be represented. To simplify our math, let us assume that the input size is simply $n = N + E$. In addition, let us assume that the running time of the

algorithm is $Time = N^3 \cdot E$. We must now determine what the worst case running time is in terms of the size of the input instance.

41.3 Asymptotic Growth

- IV.6. **Inverse:** Suppose that $y = \Theta(\log x)$. Which of the following is true: $x = \Theta(2^y)$ and $x = 2^{\Theta(y)}$? Why?
- IV.7. **Exponentials:** Compare $\Theta(2^n)$ and $2^{\Theta(n)}$. How are they the same? How are they different? If possible, give a function that is included in one of these but not included in the other. If possible, do the reverse, giving a function that is included in the second of these but not included in the first.
- IV.8. **Polynomial:** Suppose $f(n) = ab^{cn}n^d \log^e(n)$. What values can the real numbers a, b, c, d , and e take on and have $f(n)$ be considered in the course notes to have a “polynomial” growth rate? Give an example.
- IV.9. **Product of Polynomials:** The formal definition of $f(n) \in n^{\Theta(1)}$ is $\exists c_1, c_2, n_0, \forall n \geq n_0, n^{c_1} \leq f(n) \leq n^{c_2}$. Formally prove that if $f(n)$ and $g(n) \in n^{\Theta(1)}$, then $f(n) \cdot g(n) \in n^{\Theta(1)}$. What is this class of functions called?

41.4 Adding Made-Easy Approximations

IV.10. Short answers. Three or four sentences each.

- What is $\Theta(\sum_{i=1}^n 2^i i^2)$? Explain.
- What is $\Theta(\sum_{i=1}^n i^3 \log(i))$? Explain.

IV.11. Consider the following functions.

- $f(n) = 5 \cdot n^{5.5} \log^{100}(n)$.
- $f(n) = 5 \cdot 3^{4n} n^{100}$.
- $f(n) = 5 \cdot \sqrt[3]{n} + 5 \cdot \sqrt[4]{n}$.
- $f(n) = 5 \cdot 1^{8n}$.
- $f(n) = 5 + \sin(n^2)$.
- $f(n) = \{ 1 \text{ if } n \text{ is odd, } 2 \text{ if } n \text{ is even} \}$.
- $f(n) = 2^n n^2$.
- $f(n) = n^3 \log n$.

Sort the functions by growth rate from fastest to slowest. While you are at it, sort all the terms. Now answer the following questions for each function, starting with the fastest growing function. Please report your answer for both questions for each function before moving on to the next function.

- Classification:** Provide a Θ -approximation, if possible classifying the function into $\Theta(1)$, $2^{\Theta(n)}$, $\log^{\Theta(1)}(n)$, or $n^{\Theta(1)}$. What is this class called?
- Summation:** Approximate the sum $\Theta(\sum_{i=1}^n f(i))$. Which result did you use and how did you use it? Identify whether the sum is Geometric, Arithmetic, Harmonic, or Bounded Tail.

- IV.12. For each sum $\sum_{i=1}^n f(i)$: identify whether the sum is Geometric, Arithmetic, Harmonic, or Bounded Tail. For $f(i) \leq \frac{1}{i^{1+\epsilon}}$, $\sum_{i=1}^n f(i) = \Theta(1)$.
- $\sum_{i=0}^n i^3 3^i$.
 - $\sum_{i=0}^n 12i^3 + 17i^2$.
 - $\sum_{i=0}^n \frac{1}{i^3}$.
 - $\sum_{i=0}^n \frac{1}{i}$.
 - $\sum_{i=0}^n 2^{2^i}$.
- IV.13. For each of the following, give the Θ bound for the sum. In order to justify your answer, prove that the appropriate test in the notes applies.
- $\sum_{i=1}^n 23 \frac{3^i}{i^8}$.
 - $\sum_{i=1}^n 100i^6 \log^3 i + 10i^8 \log^2 i$.
 - $\sum_{i=1}^n \frac{10}{i^2 + i}$.
- IV.14. Consider the recursive relation $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + \Theta(f(n))$, $T(\Theta(1)) = \Theta(1)$.
- Explain how this recursive relation would arise from a recursive program. Explain what all the key parts are, e.g., n , $T(n)$, 4 , $\frac{n}{2}$, $T\left(\frac{n}{2}\right)$, $f(n)$, $4 \cdot T\left(\frac{n}{2}\right) + \Theta(f(n))$, $\Theta(1)$, $T(\Theta(1))$, and $\Theta(1)$. Who are the players, what does each do, and how much time does each of them spend doing it?
 - Determine $\Theta(T(n))$ for each of the following:
 (i) $f(n) = n$; (ii) $f(n) = n^2$; (iii) $f(n) = n^3$; (iv) $f(n) = n \log^3 n$; (v) $f(n) = 2^n$
 Explain why. Who is doing the most work?
- IV.15. Evaluate each of the following. Give your work. If you can't do it, say so, and guess what a reasonable answer would be.
- $T(n) = 8T(n/2) + n^2 \log^8 n$.
 - $T(n) = 8T(n/2) + n^3$.
 - $T(n) = 8T(n/2) + n^3 \log^2 n$.
 - $T(n) = 8T(n/2) + n^4 \log^3 n$.
- IV.16. For each, give the Θ bound for it.
- $T(n) = 2 \cdot T(n-1) + 1$, $T(n) = \Theta(\quad)$
 - $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 1$, $T(n) = \Theta(\quad)$
 - $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$, $T(n) = \Theta(\quad)$
 - $T(n) = T(n-1) + n$, $T(n) = \Theta(\quad)$
 - If $T(n) = 8T\left(\frac{n}{4}\right) + \Theta(n^{1.5})$. Then $T(n) = \Theta(\quad)$
 - If $T(n) = 8T\left(\frac{n}{4}\right) + \Theta(n^{1.6})$. Then $T(n) = \Theta(\quad)$

42 Partial Solutions to Additional Exercises: Part IV

42.1 Existential and Universal Quantifiers

IV.1. Logic

1. Explanations of the rules:

Separating And: If you know “I love logic and I will do my homework,” then you know both that “I love logic” and that “I will do my homework.” This is the meaning of the connector and. The logical symbol $\wedge$ denotes and. Hence, if the statement $\alpha \wedge \beta$ is true, then by definition the statement α is true, as is the statement β .

Eval/Build $\wedge$: Conversely, if both α and β are true, then their and is true. We call it an evaluation rule because from $\alpha = \beta = T$, you can evaluate $\alpha \wedge \beta = T$. We call it a build rule because if both α and β are in your list of statements that you know are true, you can add the larger statement $\alpha \wedge \beta$. A standard proof only lists things that it has proven to be true. Hence, you state that β is false by stating that $\neg\beta$ is true. Here $\neg$ denotes the logical negation symbol. The second Eval/Build rule, from $\neg\beta$ concludes that (the and of β with anything) is false, i.e., $\neg(\beta \wedge \gamma)$ is true. One fun thing about this is that we can evaluate $\neg(\beta \wedge \gamma) = T$ without knowing the value of γ .

Selecting Or: If you know “I will do my homework or I love logic,” then you know that at least one of these statements is true, perhaps both. This is the meaning of the connector or. From this, you do not know whether or not “I love logic.” On the other hand, if in addition you know that “I will not do my homework,” then you ?? because at least one is true, you can conclude that “I love logic.” The logical symbol $\vee$ denotes or. The rule is that from the statements $\alpha \vee \beta$ and $\neg\alpha$, one can conclude β .

Eval/Build $\vee$: If you know that “I love logic,” then you can conclude that “I love logic or I will do my homework,” independent from knowing whether or not “I will do my homework.” More generally, this rule states that to prove that either α or γ is true, it is sufficient to prove that α is true. We call it an evaluation rule because from only $\alpha = T$, you can evaluate $\alpha \vee \gamma = T$. We call it a build rule because from α in your list of statements that you know are true, you can add the statement $\alpha \vee \gamma$. On the other hand, in order to

prove $\alpha \vee \beta$ is not true, then you need to prove that both α and β are not true.

Excluded Middle: The statements $\alpha \vee \neg\alpha$ and $\neg(\alpha \wedge \neg\alpha)$ are axioms of our logic, i.e., assumed to be true. They state that in our logic, each variable and logical sentence/formula/statement α is either true or false but not both.

De Morgan: If it is not the case that “I love logic” and “I will do my homework,” then at least one of them must be false. More generally, $\neg(\alpha \wedge \beta)$ and $\neg\alpha \vee \neg\beta$ are equivalent statements because if it is not true that both α and β are true, then at least one of them must be false.

Modus Ponens: If you already know that “I love logic implies I will do my homework” and that “I love logic,” then a classic logic conclusion is that “I will do my homework.” More generally, from α and $\alpha \rightarrow \beta$, you can conclude β .

Deduction: The standard way to prove $\alpha \rightarrow \beta$ is by temporarily assuming that α is true and using that belief to prove β . See more below.

No-Causality: The statement $\alpha \rightarrow \beta$ is widely misunderstood by beginners. Formally, it means “If α is true, then so is β .” The statement, however, is not supposed to imply any sense of causality. A better interpretation might be “In every universe in which α is true, it happens to be the case that β is also true.” Equivalently, one might say “There are no universes in which α is true and β is not.” An example is “Is hound” $\rightarrow$ “Is dog.” Note how the set hounds is a subset of the set of dogs. As such, the set of universes in which you are a hound is a subset of the set in which you are a dog. Note that modus ponens also follows from this understanding, because knowing that you are in a universe in which α is true still tells you that β is true in this universe as well.

Reverse Direction: Note that $\alpha \rightarrow \beta$ does not imply $\beta \rightarrow \alpha$. For example, “Is hound” $\rightarrow$ “Is dog” is true and the reverse “Is dog $\rightarrow$ “Is hound” is not.

Eval/Build $\rightarrow$: The statement (I am a billionaire) $\rightarrow (1 + 1 = 2)$ is automatically true whether or not I am a billionaire. More generally, if β happens to be true, then the statement $\alpha \rightarrow \beta$ is automatically true because the contract is not broken. Similarly, the statement $(1 + 1 = 3) \rightarrow (I \text{ am a billionaire})$ is automatically true again whether or not I am a billionaire. If α happens to be false, then again $\alpha \rightarrow \beta$ is automatically true. Finally, the statement $(1 + 1 = 2) \rightarrow (I \text{ am a billionaire})$ is false because the math is wrong and I am not a billionaire. If α is true and β is false, then $\alpha \rightarrow \beta$ is false.

Contrapositive: If you know that “I love logic” implies “I will do my homework,” then it follows that “I will not do my homework” implies “I do not love logic,” because if “I do love logic,” then “I would do my homework.” More generally $\alpha \rightarrow \beta$ and $\neg\beta \rightarrow \neg\alpha$ are considered equivalent statements. They are the contrapositive of each other. Other equivalent statements are

$\neg(\alpha \wedge \neg\beta)$ and by De Morgan $\neg\alpha \vee \beta$. Recall, however, that $\beta \rightarrow \alpha$ is a different statement.

Equivalence: If you know that “I love logic” implies “I will do my homework” and you know that “I will do my homework” implies “I love logic,” then one of these is true if and only if the other is true. These become equivalent/interchangeable statements.

Cases: A very common proof technique is proof by cases. Suppose, for example, you know that you will either do your homework today or tomorrow. You know that if you do it today, then you will pass the exam. You also know that if you do it tomorrow, then you will pass the exam. Either way, you know that you will pass the exam. In general, our goal is to prove γ . We prove that there are only two cases α and β , i.e., at least one of these is true. For the first case, we assume α and prove γ . In the second, we assume β and again prove γ . Either way we can conclude γ . Trying a case and backtracking to try another case is called branching and is the basis of Recursive Backtracking (see [Chapter 23](#)). A common thing to branch on is the value of a variable x that seems to appear in a lot of places, namely $((x = T) \vee (x = F)) \wedge ((x = T) \rightarrow \gamma) \wedge ((x = F) \rightarrow \gamma) \rightarrow \gamma$. For each such case, the algorithm simplifies the sentence and recurses. One can build a more elaborate decision tree of cases. For each path from the root to a leaf, the proof proves that for this case, the statement is valid. Having a tree ensures that all the cases are handled (see Tree of Options on page 301 and Exercise III.6).

Deduction: Let’s revisit deduction. The standard way to prove $\alpha \rightarrow \beta$ is by temporarily assuming that α is true and using that belief to prove β . People struggle with understanding why we can assume α . The reason is that we really are doing proof by cases. In the first case α is false, in which case $\alpha \rightarrow \beta$ is automatically true. We are now doing the second case, in which we assume α is true.

Transitivity: If you can travel/prove from α to β and from β to γ , then you can travel/prove directly from α to γ . Similarly, from $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, you can conclude $\alpha \rightarrow \gamma$.

2. No answers provided.
3. No answers provided.
4. Love
 - (a) Suppose my strategy was to do well in school, but not follow his rules. It turned out that it was the rules he really cares about. Hence, he did not love me. More formally, $\alpha = T$, $\beta = F$, $\alpha \vee \beta = T$, $\gamma = F$, and as needed $X = (\alpha \rightarrow \gamma) \vee (\beta \rightarrow \gamma) = (T \rightarrow F) \vee (F \rightarrow F) = F \vee T = T$.
 - (b) Being careful:
 1. $(\alpha \rightarrow \gamma) \vee (\beta \rightarrow \gamma)X$ is given
 2. α Being careful
 3. β Being careful

4. Deduction Goal: $(\alpha \rightarrow \gamma) \rightarrow \gamma$.
5. $\alpha \rightarrow \gamma$ Assumption
7. γ Modus Ponens from (2) and (5)
8. $(\alpha \rightarrow \gamma) \rightarrow \gamma$ Deduction Conclusion
9. $(\beta \rightarrow \gamma) \rightarrow \gamma$ Similar to (8)
10. γ Cases (1), (8), and (9).

IV.2. Computable

- (a)
 - i. $\forall I A(I) = P(I)$.
 - ii. $\exists I A(I) \neq P(I)$.
 - iii. $\exists A \forall I A(I) = P(I)$.
 - iv. $\forall A \exists I A(I) \neq P(I)$.
 - v. $\exists A, \forall I, A(I) = P(I)$ and $\text{Time}(A, I) \leq T(n)$.
 - vi. $\forall A, \exists I, A(I) \neq P(I)$ or $\text{Time}(A, I) > T(n)$.
- (b)
 - iii. You as the prover give an algorithm A that you claim solves the problem. The adversary gives you an input I that they think is hard for your algorithm. You prove that your program works for this input, i.e., $A(I) = P(I)$.
 - iv. The adversary gives you an algorithm A that they claim solves the problem. You as the prover give an input I that you think is hard for his algorithm. You prove that their program does not work for your input, i.e. $A(I) \neq P(I)$.
 - v. It is true. Consider any input I . I is some fixed string. $\text{HALT}(I)$ has some fixed output. Let A be the algorithm that does not look at the input but simply outputs the answer $\text{HALT}(I)$. This algorithm gives the correct answer on input I .
 - vi. $\exists A \forall I [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq 3|I|^2]$.
 - vii. $\exists c \exists A \forall I [A(I) = P(I) \text{ and } \text{Time}(A, I) \leq |I|^c]$.
The prover provides a c and an algorithm A . The adversary provides an input I . The prover wins if A give the correct answer and runs takes at most time $|I|^c$.
 - viii. $\forall c \forall A \exists I [A(I) \neq P(I) \text{ or } \text{Time}(A, I) > |I|^c]$.
The adversary provides a c and an algorithm A . You must find an input I for which either A give the wrong answer or runs slower than $|I|^c$.
 - ix. $\exists P \quad [\exists c > 0 \exists A \forall I A(I) = P(I) \text{ and } \text{Time}(A, I) \leq 2^{c|I|}] \quad \text{and} \quad [\forall c \forall A \exists I A(I) \neq P(I) \text{ or } \text{Time}(A, I) > |I|^c]$.
 - x. $\forall p_1 \exists p_2 \exists t \text{ Loves}(p_1, p_2, t)$ and not $\text{Loves}(p_2, p_1, t)$.
 - xi. It is not true. The complement $\forall I \exists A A(I) = \text{HALTING}(I)$ is true. The reason is that the algorithm that always says “Yes” solves the problem for I if $P(I)$ “Yes.” Otherwise, the algorithm that always says “No” solves it.
 - xii The adversary supplies p_1 . I supply p_2 and t . Then I prove $\text{Loves}(p_1, p_2, t)$ and not $\text{Loves}(p_2, p_1, t)$.

42.2 Time Complexity

IV.4. Computing Pi

- (a) Determining the x^{th} digit of $\pi = 3.14\dots$ requires an accuracy of $\epsilon = 10^{-x}$. The stated time is then $1/\epsilon = 10^x$. An extra complication is that we must express this time as a function of the “size” n of the input. This size is $n = \log_2 x$, because x is an n bit integer. Solving gives $x = 2^n$. Hence, the time complexity of the problem is 10^{2^n} .
- (b) This implies that the algorithm is to compute each digit of π one at a time until the x^{th} is reached. If each takes $\theta(1)$ time, then the total time is $\theta(x) = \theta(1) =$
It could take (1) time to get the next bit of accuracy of π .
- (c) It takes $\log_2(10) = 3.32$ bits to specify the decimal digit. Rounding this up gives that 4 oracle calls need to be made. One needs to compute $i = 3.32x$. This takes $\theta(n)$ time.

Exercise Solutions

Chapter 1. Iterative Algorithms: Measures of Progress and Loop Invariants

1.4.1 Selection Sort: If the input for selection sort is presented as an array of values, then sorting can happen in place. The first k entries of the array store the sorted sublist, while the remaining entries store the set of values that is on the side. Finding the smallest value from $A[k + 1], \dots, A[n]$ simply involves scanning the list for it. Once it is found, moving it to the end of the sorted list involves only swapping it with the value at $A[k + 1]$. The fact that the value $A[k + 1]$ is moved to an arbitrary place in the right-hand side of the array is not a problem because these values are considered to be an unsorted set anyway. The running time is computed as follows. We must select n times. Selecting from a sublist of size i takes $\Theta(i)$ time. Hence, the total time is $\Theta(n + (n - 1) + \dots + 2 + 1) = \Theta(n^2)$ (see [Chapter 38](#)).

1.4.2 Insertion Sort: There are two steps involved in inserting an element into a sorted list. The most obvious step is to locate where it belongs. The second step to shift all the elements that are bigger than the new element one to the right to make room for it. You can find the location for the new element quickly using a binary search. However, it is easier to search and shift the larger elements simultaneously.

Linked List: Having the sorted elements stored in a linked list allows one to insert the new element in constant time. However, it then takes $\Theta(k)$ time to find where the new element goes.

Running Time: We must insert n times. Inserting into a sublist of size i takes $\Theta(i)$ time. Hence, the total time is $\Theta(1 + 2 + 3 + \dots + n) = \Theta(n^2)$.

Heap Sort: We will see in [Section 13.3](#) that each of these steps can be done in $\Theta(\log n)$ time when the elements are stored in a data structure called a heap.

1.4.3 The algorithm repeatedly passes through the array, swapping adjacent pairs if needed. After k such passes, the largest k elements have bubbled up to where they belong. Hence, it requires at most n passes until all elements are in place. Each pass requires n comparisons.

1.6.1 There are a number of problems.

1. A loop invariant should be a picture of the current state and not say what the iteration does. The loop invariant should simply be $s = \sum_{j=1}^i j$.
2. The loop invariant is not established correctly. With $i = 1$, the loop invariant requires $s = \sum_{j=1}^1 j = 1$, not $s = 0$. The choice $s = 0$ and $i = 0$ would be better.
3. The loop invariant is not maintained correctly. Let s' and i' be the values of s and i when at the top of the loop. Let s'' and i'' be the values after going around again. The loop invariant gives that $s' = \sum_{j=1}^{i'} j$. The code gives that $s'' = s' + i'$ and $i'' = i' + 1$. Together, these give that $s'' = (\sum_{j=1}^{i'} j) + i'$. This is not $\sum_{j=1}^{i'+1} j$ as required, because i' is being added in twice. $i' + 1$ should be added in order to maintain the loop invariant.
4. The exit condition is not very well stated. An equivalent and easier-to-see exit condition would be “exit when $i > I$.”
5. The exit condition, $i > I$, and the loop invariant, $s = \sum_{j=1}^i j$, together do not give the postcondition. Instead, they give that $s = \sum_{j=1}^{I+1} j$ is returned.
6. The algorithm as a whole happens to work. A quick fix is to change the loop invariant to $s = \sum_{j=1}^{i-1} j$.

Chapter 2. Examples Using More-of-the-Input Loop Invariants

2.2.1

Divide: Sorry, not provided.

Calculator

algorithm *Calculator*()

<pre-cond>: A stream of commands is entered.

<post-cond>: The results are displayed on a screen.

begin

 allocate *accum*, *current* $\in \{0 \dots 10^8 - 1\}$

 allocate *screen* $\in \{\text{showA}, \text{showC}\}$

accum = *current* = 0

screen = *showC*

 loop

<loop-invariant>: The bounded memory of the machine remembers the current value of the accumulator and the current value being entered. It also has a Boolean variable that indicates whether the screen should display the current or the accumulator value.

 get(*c*)

 if(*c* $\in \{0 \dots 9\}$) then

current = $10 \times \text{current} + c \bmod 10^8$

screen = *showC*

```

else if (c = '+' ) then
    accum = accum + current mod 108
    current = 0
    screen = showA
else if (c = 'clr' ) then
    accum = 0
    current = 0
    screen = showC
end if
if (screen = showC) then
    display(current)
else
    display(accum)
end if
end loop
end algorithm

```

Longest Block of Ones

algorithm *LongestBlockOfOnes*(A, n)

⟨pre-cond⟩: The input is A , a 0, 1 array of length n .

⟨post-cond⟩: The output is the location $A[k_1 \dots k_2]$ of the longest block of ones and its length $leng$.

```

begin
     $i = 0; p_{max} = 1; q_{max} = 0; leng_{max} = 0; p_{current} = 1;$ 
     $leng_{current} = 0 \in \{0 \dots n\}$ 
    loop
        ⟨loop-invariant⟩:  $A[p_{max}, q_{max}]$  is a longest block of
        ones in  $A[1 \dots i]$  and
         $leng_{max} = q_{max} - p_{max} + 1$  is its length.
         $A[p_{current}, i]$  is the longest block of ones in  $A[1 \dots i]$ 
        ending in  $A[i]$  and
         $leng_{current} = i - p_{current} + 1$  is its length.
        exit when  $i = n$ 
        if ( $A[i + 1] = 1$ ) then
             $leng_{current} = leng_{current} + 1$ 
        else
             $p_{current} = i + 2$ 
             $length_{current} = 0$ 
        end if
        if ( $leng_{max} < leng_{current}$ ) then
             $p_{max} = p_{current}$ 
             $q_{max} = i + 1$ 
             $length_{max} = length_{current}$ 

```

```

        end if
         $i = i + 1$ 
    end loop
    return  $A[p_{max}, q_{max}]$ 
end algorithm

```

2.3.1

(2): The loop invariant is:

- The beginning $[0, a]$ of the cake has been partitioned into $|Q|$ disjoint pieces.
- Each player $p_i \in Q$ has been allocated a piece $[a_i, b_i]$ worth at least $\frac{1}{n}$ to him.
- The remaining $[a, 1]$ interval of the cake is worth at least $(n - |Q|)/n$ to each of the remaining players, i.e., to those in $P - Q$.

(3):

algorithm *Partition*(P)

<pre-cond>: As above

<post-cond>: As above

begin

$a = 0$ and $Q = \emptyset$

 loop

<loop-invariant>: As above.

 exit when $|Q| = n$

 loop $i \in P - Q$

$c_i = \text{Cut}(p_i, a, \frac{1}{n})$

 end loop

i_{min} = the $i \in P - Q$ that minimizes c_i

$[a_{i_{min}}, b_{i_{min}}] = [a, c_{i_{min}}]$

$a = c_{i_{min}}$

$Q = Q + i_{min}$

 end loop

 return all parts $[a_i, b_i]$ for each $i \in P$

end algorithm

(1), (4)–(7): Sorry, the remaining solutions are not provided.

Chapter 3. Abstract Data Types

3.1.5 Instead of bounding the height given the number of nodes, it is easier to compute the reverse relation. Let $N(h)$ be the minimal number of nodes in an AVL tree of height h . In order for a tree to be of height h , it must have at least one subtree of height $h - 1$. In order for it to be an AVL tree, the other subtree can differ by at most one, so it must have height at least $h - 2$. It follows that the number of nodes in this tree is at least $N(h) = N(h - 1) + N(h - 2) + 1$. Except for the +1 of the root, this is that same as the famous Fibonacci numbers defined by

$Fib(n) = Fib(n-1) + Fib(n-2)$. [Exercise 39.2.1](#) goes on to prove that $Fib(n) = \Theta(\alpha^n)$, where $\alpha = \frac{1+\sqrt{5}}{2}$. If $N(h) = \Theta(\alpha^n)$, then $H(n) = \Theta(\log n)$.

- 3.2.4** The tests will be executed in the order that they are listed. If $next = nil$ is tested first and passes, then because there is an *OR* between the conditions, there is no need to test the second. However, if $next.info \geq key$ is the first test and $next$ is nil, then using $next.info$ to retrieve the information in the node pointed to by $next$ will cause a run-time error.

Chapter 4. Narrowing the Search Space: Binary Search

- 4.4.1** Doing binary search in $O(\log(n \times m))$ time is impossible. See the lower bound in question ([Exercise 7.0.7](#)). If you take $O(n \log m)$ time doing binary search in each row, then you are taking too much time. It can be done by examining $n + m - 1$ entries. Observe that the values in the matrix increase from $A[1, 1]$ to $A[n, m]$. Hence, the boundary between values that are less than or equal to x and those that are greater follows some monotonic path from $A[1, m]$ to $A[n, 1]$. The algorithm traces this path starting at $A[1, m]$. When it is at the point $A[i, j]$, the loop invariant is that we have stored the best answer from those outside the subrectangle $A[i \dots n, 1 \dots j]$. Initially, this is true for $[i, j] = [1, m]$, because none of the matrix is excluded. Now suppose it is true for an arbitrary $[i, j]$. The algorithm then compares $A[i, j]$ with x . If it is better than our current best answer, then our current best is replaced. If $A[i, j] \leq x$, then because the values in the row $A[i, 1 \dots j]$ are all smaller than or equal to $A[i, j]$, these are worse answers, and hence we can conclude that we now have the best answer from those outside the subrectangle $A[i + 1 \dots n, 1 \dots j]$. We maintain the loop invariant by increasing i by one. On the other hand, if $A[i, j] > x$, then it is too big and so are all the elements in the column $A[i \dots n, j]$, which are even bigger. We can conclude that we have the best answer from those outside of the subrectangle $A[i \dots n, 1 \dots j - 1]$. We maintain the loop invariant by decreasing j by one. The exit condition is $|i \dots n| = 0$ or $|1 \dots j| = 0$ (i.e., $i > n$ or $j < 1$). When this occurs, the subrectangle $A[i \dots n, 1 \dots j]$ is empty. Hence, our best answer, which by the loop invariant is the best from those outside this subrectangle, must be the best overall. The measure of progress, $|i \dots n| + |1 \dots j| - 1 = (n - i + 1) + (j) - 1$, is initially $n + m - 1$ and decreases by one each iteration. After $n + m - 1$ iterations, either the algorithm has already halted or the measure has reached zero, at which point the exit condition is definitely met.

Chapter 7. The Loop Invariant for Lower Bounds

- 7.0.2** The bound is $n \leq r^t$.

Each round, they select one row; hence, there are r possible answers. After t rounds, there are r^t possible combinations of answers.

The only information that you know is which of these combinations they gave you. Which card you produce depends deterministically (no magic) on the

combination of answers given to you. Hence, depending on their answers, there are at most r^t cards that you might output.

However, there are n cards, any of which may be the selected card. In conclusion, $n \leq r^t$.

The book has $n = 21$, $r = 3$, and $t = 2$. Because $21 = n \not\leq r^t = 3^2 = 9$, the trick in the book does *not* work.

Two rounds is not enough. There need to be three rounds.

7.0.3 It is a trick question, because with a balance there are three, not two, outcomes and hence only $\log_3 n$ operations are needed. Divide the objects into three piles, two of equal size and the third as close as possible. Put the first two piles on the scale. If one is heavier, then it contains the heavier object; otherwise, the third pile does. Recurse on this one pile.

7.0.6 In the lower bound for parity, any starting input I would have worked equally well. Here, however, there is only one input that will work, and that is I being the all-zero string. This ensures that, as before, changing the j th bit of I , for any $j \in J = [1, n]$, changes the answer from the *AND* being zero to the *AND* being one. This proves that any algorithm solving the problem requires time of at least n .

Chapter 11. Abstractions, Techniques, and Theory

11.5.2 R_a : One might complain that if my instance is $\langle n, m \rangle$, then my friend's instance cannot be $\langle n - 1, 2m \rangle$, because $2m$ is not smaller than m . However, we can define the size of instance $\langle n, m \rangle$ to be simply n . According to this measure, my friend's instance is indeed smaller. Moreover, when the instance becomes of size zero or smaller, then $n \leq 0$ and the recursion stops. We prove that the depth of recursion is at most n as follows. On instance $\langle n, m \rangle$, the size starts at n and decreases by at least one every level of recursion, so after n levels the size is at most zero and the algorithm stops recursing further. For example, starting with $\langle 5, 2 \rangle$, it recurses on $\langle 4, 4 \rangle$, $\langle 3, 8 \rangle$, $\dots$, $\langle 0, 64 \rangle$, and then halts.

R_b : One might claim that all is well because both friends get instances $\langle n - 1, m \rangle$ and $\langle n, m - 1 \rangle$ that are smaller. However, for this to be true for both friends, the size must be something like $n + m$. However, according to this definition, the instance $\langle 5, -5 \rangle$ is small, but the algorithm does not halt. There is a path down this recursive tree that is infinite, namely, $\langle n, m \rangle, \langle n, m - 1 \rangle, \langle n, m - 2 \rangle, \dots, \langle n, 1 \rangle, \langle n, 0 \rangle, \langle n, -1 \rangle, \langle n, -2 \rangle, \dots$

R_c : Here the size of the instance $\langle n, m \rangle$ can be defined to be $n + m$. According to this measure, each friend is given a smaller instance. Moreover, if the size on the instance is zero, then either $n \leq 0$ or $m \leq 0$. Either way, the program halts. The depth of recursion can be at most $n + m$ because this is the initial size and the size decreases by one each iteration.

R_d : Let the size of the instance $\langle n, m \rangle$ be $5n + 2m$. Then the first friend's instance $\langle n - 1, m + 2 \rangle$ has size $5(n - 1) + 2(m + 2) = 5n + 2m - 1$, which is one smaller. The second friend's instance $\langle n + 1, m - 3 \rangle$ has size $5(n +$

$1) + 2(m - 3) = 5n + 2m - 1$, which is also one smaller. Moreover, if the size on the instance is zero, then either $n \leq 0$ or $m \leq 0$. Either way, the program halts. The depth of recursion can be at most $5n + 2m$ because this is the initial size and the size decreases by one each iteration.

R_e : I claim that there is a path down this recursion tree that is infinite. If my instance is $\langle n, m \rangle$, then my first friend has $\langle n - 4, m + 2 \rangle$, their first friend has $\langle n - 8, m + 4 \rangle$, their first friend has $\langle n - 12, m + 6 \rangle$, their second friend has $\langle n - 6, m + 3 \rangle$, and their second friend has $\langle n, m \rangle$ which is the same as my instance. This can be repeated infinitely often.

It is interesting that the last two examples can be generalized to the friend's instances of size $\langle n - a, m + b \rangle$ and $\langle n + c, m - d \rangle$. If $ad > bc$ then the program halts, else it does not.

11.6.2 $Fun(1) = X$
 $Fun(2) = Y$
 $Fun(3) = AYBXC$
 $Fun(4) = A AYBXC B Y C$
 $Fun(5) = A AAYBXC BYC B AYBX C$
 $Fun(6) = A AAAYBXC BYCBAYBXC B AAYBXC BYC C$

11.7.2 To prove $S(0)$, let $n = 0$ in the inductive step. There are no values k where $0 \leq k < n$. Hence, no assumptions are being made. Hence, your proof proves $S(0)$ on its own.

Chapter 12. Some Simple Examples of Recursive Algorithms

12.1.1 Insertion sort and selection sort.

12.1.2 1. Given $\langle a_1, a_2, \dots, a_n \rangle$, I remove the last character a_n . I give $\langle a_1, a_2, \dots, a_{n-1} \rangle$ to my friend, and they return the reversed tuple $\langle a_{n-1}, \dots, a_1 \rangle$ to me. I add a_n to the front of the tuple, producing $\langle a_n, a_{n-1}, \dots, a_1 \rangle$ as required. If my initial tuple has only zero (or one) element, then there is nothing to do.

algorithm $Reverse(\langle a_1, a_2, \dots, a_n \rangle)$

<pre-cond>: An instance is a tuple.

<post-cond>: The output is the reverse tuple $\langle a_n, a_{n-1}, \dots, a_1 \rangle$.

begin

 if($n = 0$ or $n = 1$) then

 return (instance unchanged)

 else

 return ($\langle a_n, Reverse(\langle a_1, a_2, \dots, a_{n-1} \rangle) \rangle$)

 end if

end algorithm

2. The iterative program has two (nonnested) loops. The first pushes each element on the stack, one at a time, starting with a_n . The loop invariant is that after i iterations what remains in the tuple is $\langle a_1, a_2, \dots, a_{n-i-1}, a_{n-i} \rangle$ and the stack contains $\langle a_{n-i+1}, a_{n-i+2}, \dots, a_n \rangle$ with a_{n-i+1} at the top. At the $i=0$ iteration, the loop invariant is trivially true. The next iteration removes the last element a_{n-i} from the tuple and pushes it on the stack. This maintains the loop invariant while making progress. In the end, with $i=n$, $\langle a_1, a_2, \dots, a_n \rangle$ is on the stack with a_1 at the top. The second loop pops each element off the stack and puts it at the beginning of the tuple. The loop invariant is that after i iterations, the stack again contains $\langle a_{i+1}, a_{i+2}, \dots, a_n \rangle$ with a_{i+1} at the top, but now the tuple is $\langle a_i, a_{i-1}, \dots, a_2, a_1 \rangle$. With $i=n$, the stack is empty and the tuple is $\langle a_n, a_{n-1}, \dots, a_2, a_1 \rangle$.
3. Recursion is implemented on a computer, using a stack of stack frames. The first stack frame is given $\langle a_1, a_2, \dots, a_n \rangle$, and it removes and remembers the last character a_n . Its friend is the second stack frame, which is given $\langle a_1, a_2, \dots, a_{n-1} \rangle$. It removes and remembers its last character a_{n-1} . As we recurse deeper, the stack frames that have not yet completed are pushed on a stack. After i such stack frames, the loop invariant is that the the friend's friend's friend's ... friend is given the tuple $\langle a_1, a_2, \dots, a_{n-i-1}, a_{n-i} \rangle$ and the stack contains stack frames, each remembering one of the elements $a_{n-i+1}, a_{n-i+2}, \dots, a_n$, with a_{n-i+1} at the top. Note this is the same loop invariant as the iterative program. The recursive base case is reached when $i=n$ and the stack frame is given the empty tuple. Then, one at a time, in reverse order the stack frames complete their computations by each adding its element to the beginning of the tuple. The loop invariant is that after i such returns, the stack of stack frames is remembering $a_{i+1}, a_{i+2}, \dots, a_n$ with a_{i+1} at the top, and the current stack frame is returning the tuple $\langle a_i, a_{i-1}, \dots, a_2, a_1 \rangle$. Again, note that this is the same loop invariant as the iterative algorithm. With $i=n$, the stack is empty and the first stack frame returns the tuple $\langle a_n, a_{n-1}, \dots, a_2, a_1 \rangle$.

12.2.1 1). Given the integers a and b , the iterative algorithm creates two numbers $x=b$ and $y=a \bmod b$. It notes that $GCD(a,b) = GCD(x,y)$, and hence it can return $GCD(x,y)$ instead of $GCD(a,b)$. This algorithm is even easier when you have a friend. We simply give the subinstance $\langle x,y \rangle$ to the friend, and he computes $GCD(x,y)$ for us. For the iterative algorithm, we need to make sure we are making progress, and for the recursive algorithm, we need to make sure that we give the friend a smaller instance. Either way, we make sure that in some way $\langle x,y \rangle$ is smaller than $\langle a,b \rangle$. For the iterative algorithm, we need an exit condition that we are sure to eventually meet, and for the recursive algorithm, we need base cases such that every possible instance is handled. Either way, we consider the case when y or b is zero. The resulting code is

algorithm $GCD(a, b)$

⟨pre-cond⟩: a and b are integers.

⟨post-cond⟩: Returns $GCD(a, b)$.

```
begin
  if( $b = 0$ ) then
    return( $a$ )
  else
    return( $GCD(b, a \bmod b)$ )
  end if
end algorithm
```

2). We will need to understand this relationship $y = a \bmod b$ better. Here y is the remainder when you divide a by b . If we let $r = \lfloor \frac{a}{b} \rfloor$, then $a = r \cdot b + y$ or $y = a - r \cdot b$.

When we generalize the problem, the friend, in addition to g , also gives us u_{sub} and v_{sub} such that $u_{sub} \cdot x + v_{sub} \cdot y = g = GCD(x, y) = GCD(a, b)$. Plugging in $x = b$ and $y = a - r \cdot b$ gives $u_{sub} \cdot b + v_{sub} \cdot (a - r \cdot b) = g$, or $v_{sub} \cdot a + (u_{sub} - v_{sub} \cdot r) \cdot b = g$. Hence, if we set $u = v_{sub}$ and $v = u_{sub} - v_{sub} \cdot r$, then we get $u \cdot a + v \cdot b = g = GCD(a, b)$ as required. I simply provide these answers. For the base case with $b = 0$, we have $g = GCD(a, b) = a$. Hence, $u = 1$ and $v = 0$ gives that $u \cdot a + v \cdot b = g = GCD(a, b)$. The resulting code is

algorithm $GCD(a, b)$

⟨pre-cond⟩: a and b are integers.

⟨post-cond⟩: Returns integers g, u , and v such that

$u \cdot a + v \cdot b = g = GCD(a, b)$.

```
begin
  if( $b = 0$ ) then
    return( $\langle a, 1, 0 \rangle$ )
  else
     $x = b$ 
     $r = \lfloor \frac{a}{b} \rfloor$ 
     $y = a - r \cdot b$ 
     $\langle g, u_{sub}, v_{sub} \rangle = GCD(x, y)$ 
     $u = v_{sub}$ 
     $v = u_{sub} - v_{sub} \cdot r$ 
    return( $\langle g, u, v \rangle$ )
  end if
end algorithm
```

3). Our goal is to find two integers U and V such that $U \cdot a + V \cdot b = w$. Then you give the storekeeper U of the a coins and V of the b coins for a total worth of w dollars. If U or V is negative, this amounts to the storekeeper giving you coins as change.

To find U and V , let's start by calling the GCD algorithm on a and b . This returns integers g, u , and v such that $u \cdot a + v \cdot b = g = GCD(a, b)$.

If g divides evenly into w , then multiplying through by $\frac{w}{g}$ gives $(\frac{uw}{g}) \cdot a + (\frac{vw}{g}) \cdot b = g(\frac{w}{g}) = w$, and we are done.

By the definition of $g = GCD(a, b)$, we know g divides a and b , and hence it divides $U \cdot a + V \cdot b$ evenly. It follows that if g does not divide w evenly, then there is no integer solution to $U \cdot a + V \cdot b = w$.

4). Solution not provided.

Chapter 13. Recursion on Trees

- 13.3.1**
1. Where in the heap can the value 1 go? It must be in one of the leaves. If 1 were not at a leaf, then the nodes below it would need a smaller number, of which there are none.
 2. Which values can be stored in entry $A[2]$? It can contain any value in the range 7–14. It can't contain 15, because 15 must go in $A[1]$. We already know that $A[2]$ must be greater than each of the seven nodes in its subtree. Hence, it can't contain a value less than 7. For each of the other values, a heap can be constructed such that $A[2]$ has that value.
 3. Where in the heap can the value 15 go? The value 15 must go in $A[1]$ (as we have mentioned).
 4. Where in the heap can the value 6 go? 6 can go anywhere except $A[1]$, $A[2]$, or $A[3]$. Entry $A[1]$ must contain 15, and $A[2]$ and $A[3]$ must be at least 7.

13.4.1 algorithm *Derivative*(f, x)

<pre-cond>: f is an equation and x is a variable.

<post-cond>: The derivative of f with respect to x is returned.

begin

 if($f = "x"$) then

 result(1)

 else if(f = a real value or a single variable other than " x ") then

 result(0)

 end if

 % if f is of the form ($goph$)

$g = \text{Copy}(\text{leftSub}(f))$ % Copy needed for "*" and "f".

$h = \text{Copy}(\text{rightSub}(f))$ % Three copies needed for "f".

$g' = \text{Derivative}(\text{leftSub}(f), x)$

$h' = \text{Derivative}(\text{rightSub}(f), x)$

 if($f = g + h$) then

 % See f_1 in Figure 13.11

 result($g' + h'$)

```

else if( $f = g - h$ ) then
    result( $g' - h'$ )
else if( $f = g * h$ ) then
    % See  $f_2$  in Figure 13.11
    result( $g' * h + g * h'$ )
else if( $f = g/h$ ) then
    % See  $f_3$  and  $f_4$  in Figure 13.11
    result( $(g' * h - g * h')/(h * h)$ )
end if
end algorithm

```

13.4.3

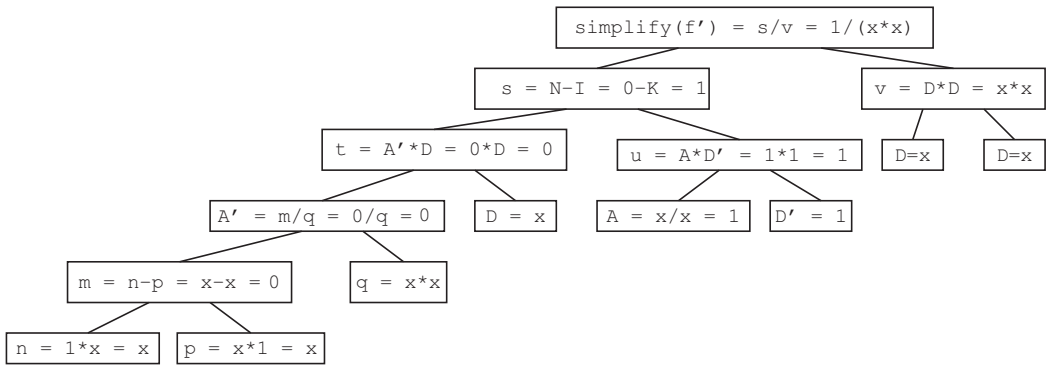


Fig. E.1

Chapter 14. Recursive Images

14.1.1 Falling Line: This construction consists of a single line with image $n - 1$ raised, tilted, and shrunk:

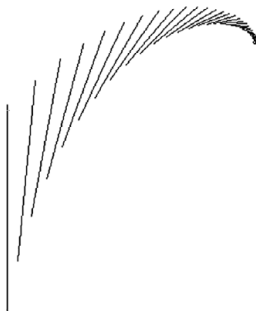


Fig. E.2

14.1.2 Binary Tilt: This image is the same as the birthday cake. The only differences are that the two places to recurse are tilted and one of them has to be flipped upside down:



Fig. E.3

Chapter 15. Parsing with Context-Free Grammars

15.0.1 Sorry, the solutions for 2 and 3 are not included.

```

s = ( ( ( 1 ) * 2 + 3 ) * 5 * 6 + 7 )
    | -exp-----|
    | -term-----|
    | -fact-----|
    ( | -exp-----| )
      | -term-----| + t
      | -fact-----| * f * f f
      ( | -exp-----| ) 5 6 7
        | -t-----| + t
        | -f-| * f f
        ( e ) 2 3
          t
          f
          1
s = ( ( ( 1 ) * 2 + 3 ) * 5 * 6 + 7 )

```

Fig. E.4

Chapter 20. Graph Search Algorithms

20.2.1 The node v can't be in $V_{k'}$ for $k' > k + 1$, because there is a path of length $k + 1$ to it, namely, the path to u followed by the edge $\langle u, v \rangle$. If v has not been found before, then we are just finding it. Its $d(u)$ is being set to $d(v) + 1 = k + 1$. By LI1, this must be its distance from s . Hence, v must be in V_{k+1} . If v has been found before, it is because a shortest path has already been found to it. If the edge $\langle u, v \rangle$ is directed, then this previous path could have any length $k' \leq k + 1$. However, if this edge is undirected, then there is a catch. Suppose v is in $V_{k'}$. Then a possible path to u is that of length $k' + 1$ from s to v followed by the edge $\{u, v\}$ backward to u . because the shortest path to u is of length k , we have $k' + 1 \geq k$ or $k' \in \{k - 1, k, k + 1\}$.

- 20.2.2** The shortest-path algorithm given in this section is identical to the generic search algorithm in [Section 20.1](#) except that a queue is used. Hence, the running time is $\Theta(|E|)$. The time is not less if you are searching for a path to a specific node t .
- 20.3.2** Despite differences in the algorithms, on a graph with edge weights one, breadth-first search and Dijkstra's algorithm are identical. Breadth-first search handles the first node in its queue, whereas Dijkstra's algorithm handles the node with the next smallest $d(v)$. However, breadth-first search's third loop invariant ensures that the nodes are found and added to the queue in the order of distance $d(v)$. Hence, handling the next in the queue amounts to handling the next smallest $d(v)$. Breadth-first search's first loop invariant states that the correct minimal distance $d(v)$ to v is obtained when the node v is first found, whereas with Dijkstra's algorithm we are not sure to have it until the node is handled. However, with edge weights one, when v is first found in Dijkstra's algorithm, $d(v)$ is set to the length of the overall shortest path and never changed again.
- 20.6.2** The shortest path to node v will not contain any nodes u that appear after it in the total order, because by the requirements of the total order there is no path from u to v . Hence, it is fine to handle v , committing to a shortest path to v , before considering u . Hence, it is fine to handle the nodes in the order given by the total order. The advantage of this algorithm is that you do not need to maintain a priority queue, as done in Dijkstra's algorithm. This decreases the time from $\Theta(|E| \log |V|)$ to $\Theta(|E|)$.

Chapter 21. Network Flows and Linear Programming

- 21.2.5** Given a network $\langle G, s, t \rangle$, run the max flow algorithm on it. In addition to returning a maximum flow, it also returns a minimum cut, which is used to witness the fact that there is no better flow.
- 21.5.1** 1. The first does not have a matching. A witness is the fact that nodes 1, 3, and 5 are only connected to B and D. Hence, the three can't be matched to the two. In the language of Hall's theorem, let $A = \{1, 3, 5\}$; then $N(A) = \{B, D\}$. Because $|A| > |N(A)|$, Hall's theorem gives that there is no matching. The second does have a matching. A witness is the following matching:

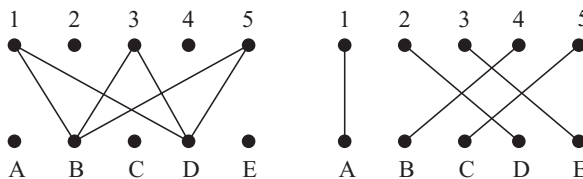


Fig. E.5

2. Consider an arbitrary $A \subseteq L$. Note that the set $B = \{M(u) | u \in A\}$ contains $|A|$ distinct nodes and that $B \subseteq N(A)$. Hence, $|A| \leq |N(A)|$.
3. Let $A = U \cap L$ be the set of nodes that are both on the left side of the bipartite graph and on the left side of the cut. Consider any node $v \in N(A)$. Because $v \in N(A)$, there is a node $u \in A \subset U$ such that $\langle u, v \rangle$ is an edge. If $v \in V$, then this edge (u, v) crosses the cut. But this edge has capacity ∞ . In this case, the capacity of the cut is well over $|L|$. On the other hand, if $v \in U$, then the edge from v to t is across the cut. Now consider any node $u \in L - A \subset V$. The edge from s to u crosses the cut. This proves that the number of edges across the cut is at least $|N(A)| + (|L| - |A|)$, which by our assumption is at least $|A| + (|L| - |A|) = |L|$.
4. We have seen that there is a matching with $|L|$ edges iff the max flow in this graph has value $|L|$ iff the min cut in this graph has capacity $|L|$. The cut with s on one side by itself has $|L|$ edges going across the cut, namely, those edges from s to L . By the last question, if $\forall A \subseteq L, |A| \leq |N(A)|$, then every cut has at least $|L|$ edges across it. Hence, the min cut must be $|L|$. Hence, the max flow has value $|L|$. Hence, there is a matching with $|L|$ edges. All the nodes in L must be matched.
5. By the last question, it is sufficient to prove that $\forall A \subseteq L, |A| \leq |N(A)|$ is true. Consider some set $A \subseteq L$. Because each node in $A \subseteq L$ has degree at least k , we know that at least $k \cdot |A|$ edges leave A . All of the edges that leave A must enter its neighborhood set $N(A)$. Hence, the number that leave A is at most the number that enter $N(A)$. Because every node in $N(A) \subseteq R$ has degree at most k , we know that at most $k \cdot |N(A)|$ enter $N(A)$. It follows that $k \cdot |A| \leq \# \text{ leave } A \leq \# \text{ enter } N(A) \leq k \cdot |N(A)|$. Hence, $|A| \leq |N(A)|$, as is needed.

Chapter 22. Greedy Algorithms

22.2.2 In the following instance of the interval cover problem, the greedy criterion that selects the interval that covers the largest number of uncovered points would commit to the top interval. However, the optimal solution does not contain this interval, but contains the bottom two intervals.



Fig. E.6

22.3.2 Algorithms 1, 2, and 4 are suboptimal for the following counterexample instance:

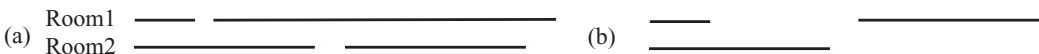


Fig. E.7

Diagram (a) gives the events in the instance and the optimal schedule in two rooms. Diagram (b) gives the suboptimal schedule produced by these three algorithms. Note that the third algorithm, which schedules the next event in the room with the latest last-scheduled finishing time, gives the optimal schedule. We will now prove that it always gives an optimal solution.

As with all greedy algorithms, the loop invariant is that there is at least one optimal solution S_{t-1} consistent with the choices made so far, that is, scheduling in the same rooms the same events whose schedules have been committed to so far and not scheduling the events rejected so far. Initially, no choices have been made, and hence trivially all optimal solutions are consistent with these choices. We prove that the loop invariant is maintained by modifying the schedule S_{t-1} into another schedule S_t and prove that this new schedule is valid, consistent with all previous and current choices, and optimal. There are three cases.

If our greedy algorithm did not schedule the next event i , then this event must conflict in each room with a previously scheduled event. Hence, S_{t-1} cannot have this next event i scheduled either, because it too has scheduled these previous events. Hence, S_{t-1} itself is already consistent with the most recent choice.

If our greedy algorithm did schedule the next event i in room j and S_{t-1} does not schedule this event at all, then we modify the schedule S_{t-1} into S_t by adding i to room j and removing any events from j that conflict with it. Just as done with the one-room scheduling algorithm in [Section 22.2.1](#), we can prove that only one event is removed and hence S_t is valid, consistent, and optimal.

The remaining case occurs when our greedy algorithm scheduled the next event i in room j and S_{t-1} schedules it in room j' . (See diagram (a).) We modify the schedule S_{t-1} into S_t as follows. (See diagram (b).) We cannot move the events whose schedule has already been committed to by the algorithm, because S_t needs to remain consistent with these choices. (See the events in the *Commit* circle.) We need to move event i from room j' to room j so that it too is consistent with what the algorithm has done. But making this change may create conflicts. To fix these, we swap every event scheduled by S_{t-1} in room j' with the finishing time of event i or later with every such job scheduled in room j . (See the events in the rectangle.)

We now prove that the resulting solution S_t is valid, consistent, and optimal.

A Valid Solution: Our modified solution S_t contains no conflicts, because S_{t-1} contained none and we will prove now that no new conflicts were introduced. There are no new conflicts between the previously committed events (circle), because they did not change. There are no new conflicts between the later-committed events (rectangle), because they flipped rooms all together. Event i does not conflict with the previously committed events in room j , because the algorithm scheduled it there. The even later events that were in room j' don't either, because they are even later. The later events that were in room j won't conflict with the previously scheduled events in room j' , because they did not conflict with those in room j and we know by the algorithm's choice of room j that the last-scheduled finishing time for j is later than that for room j' .

Consistent with Choices Made: S_{t-1} was consistent with the previous choices. We moved event i from room j' to room j to make S_t consistent with this most recent choice. We did not move any events in *Commit*.

Optimal: Schedule S_t has the optimal number of events in it, because it has the same number of events as S_{t-1} .

Loop Invariant Has Been Maintained: In conclusion, we have constructed a valid optimal schedule S_t that is consistent with the choices made by the algorithm. This proves that the loop invariant has been maintained.

The rest of the proof of correctness of this greedy algorithm is the same as that of all the others.

Chapter 23. Recursive Backtracking

23.5.1 Asking to provide the best word is not a “little question” for the bird. They would be doing most of the work for you. Asking the friend to provide the best place on the board to put the word is not a subinstance of the same problem as that of the given instance.

23.5.2 The simple brute force algorithm searches the dictionary for each permutation of each subset of the letters. The backtracking algorithm tries all of the possibilities for the first letter and then recurses. Each of these stack frames tries all of the remaining possibilities for the second letter, and so on. This can be pruned by observing that if the word constructed so far, e.g., “xq,” does not match the first letters of any word in the dictionary, then there is no need for this stack frame to recurse any further. (Another improvement on the running time ensures that the words are searched for in the dictionary in alphabetical order.)

- 23.5.3**
1. $\langle 1, 5, 8, 6, 3, 7, 2, 4 \rangle$
 2. $\langle 1, 6, 8, 3, 7, 4, 2, 5 \rangle$
 3. $\langle 1, 7, 4, 6, 8, 2, 5, 3 \rangle$
 4. $\langle 1, 7, 5, 8, 2, 4, 6, 3 \rangle$
 5. $\langle 2, 4, 6, 8, 3, 1, 7, 5 \rangle$
 6. $\langle 2, 5, 7, 1, 3, 8, 6, 4 \rangle$
 7. $\langle 2, 5, 7, 4, 1, 8, 6, 3 \rangle$
 8. $\langle 2, 6, 1, 7, 4, 8, 3, 5 \rangle$
 9. $\langle 2, 6, 8, 3, 1, 4, 7, 5 \rangle$
 10. $\langle 2, 7, 3, 6, 8, 5, 1, 4 \rangle$
 11. $\langle 2, 7, 5, 8, 1, 4, 6, 3 \rangle$
 12. $\langle 2, 8, 6, 1, 3, 5, 7, 4 \rangle$

23.5.4 We will prove that the running time is bounded between $(\frac{n}{2})^{\frac{n}{2}}$ and n^n and hence is $n^{\Theta(n)} = 2^{\Theta(n \log n)}$. Without any pruning, there are n choices on each of n rows as to where to place the row’s queen. This gives n^n different placements of the queens. Each of these solutions would correspond to a leaf of the tree of stack frames. This is clearly an upper bound on the number when there is pruning.

I will now give a lower bound on how many stack frames will be executed by this algorithm. Let j be one of the first $\frac{n}{6}$ rows. I claim that each time that a stack frame is placing a queen on this row, it has at least $\frac{n}{2}$ choices as to where to place it. The stack frame can place the queen on any of the n squares in the row as long as this square cannot be captured by one of the queens placed above it. If row i is above our row j , then the queen placed on row i can capture at most three squares of row j : one by moving on a diagonal to the left, one by moving straight down, and one by moving on a diagonal to the right. Because j is one of the first $\frac{n}{6}$ rows, there are at most this number of rows i above it, and hence at most $3 \times \frac{n}{6}$ of row j 's squares can be captured. This leaves, as claimed, $\frac{n}{2}$ squares on which the stack frame can place the queen.

From the above claim, it follows that within the tree of stack frames, each stack frame within the tree's first $\frac{n}{6}$ levels branches out to at least $\frac{n}{2}$ children. Hence, at the $(\frac{n}{6})$ th level of the tree there are at least $(\frac{n}{2})^{\frac{n}{6}}$ different stack frames. Many of these will terminate without finding a complete valid placement. However, this is a lower bound on the running time of the algorithm, because the algorithm recurses to each of them.

23.5.5 The line “ $k_{min} = a\ k$ that maximizes $cost_k$.”

Chapter 24. Dynamic Programming Algorithms

24.3.3 algorithm *PrintingNeatly*($\langle M; l_1, \dots, l_n \rangle$)

<pre-cond>: $\langle l_1, \dots, l_n \rangle$ are the lengths of the words, and M is the length of each line.

<post-cond>: *optSol* splits the text into lines in an optimal way, and *cost* is its cost.

begin

 % Table: *optSol*[i] would store an optimal way to print the first i words of the input, but actually we store only the bird's advice for the subinstance and the cost of its solution.

table[$0 \dots n$] *birdAdvice*, *cost*

 % Base case: The only base case is for the best printing of the first zero words.

 Its solution is the empty printing with cost zero.

 % *optSol*[0] = $\emptyset$

cost[0] = 0

birdAdvice[0] = $\emptyset$

 % General cases: Loop over subinstances in the table.

 for $i = 1$ to n

 % Solve instance $\langle M; l_1, \dots, l_i \rangle$ and fill in table entry $\langle i \rangle$.

$K =$ maximum number k such that the words of length


```

 $l_{i-k+1}, \dots, l_i$  fit on a single line.
% Try each possible bird answers.
for  $k = 1$  to  $K$ 
    % The bird-and-friend algorithm: The bird tells us to
    % put  $k$  words on the last line. We ask the friend for an
    % optimal printing of the first  $i - k$  words. They give us
    %  $optSol[i - k]$ , which they had stored in the table. To this
    % we add the bird's  $k$  words on a new last line. This gives
    % us  $optSol_k$ , which is a best printing of the first  $i$  words
    % from among those printings consistent with the bird's
    % answer.
    %  $optSol_k = \langle optSub[i - k], k \rangle$ 
     $cost_k = cost[i - k] + (M - k + 1 - \sum_{j=i-k+1}^i l_j)^3$ 
end for
% Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
% we keep the best of these best.
 $k_{min} = a \text{ } k \text{ that minimizes } cost_k$ 
 $birdAdvice[i] = k_{min}$ 
%  $optSol[i] = optSol_{k_{min}}$ 
%  $cost[i] = cost_{k_{min}}$ 
end for
 $optSol = PrintingNeatlyWithAdvice(\langle M; l_1, \dots, l_n \rangle, birdAdvice)$ 
return  $\langle optSol, cost[n] \rangle$ 
end algorithm

```

24.4.1 (a): If $x_n = y_m$, then we must prove that there is at least one optimal solution that contains both of these last characters. Consider an optimal solution. It must end in this last character; otherwise, it could be extended to contain it. It might not contain both of them, as in the case of $X = \langle \underline{A}, \underline{B}, B \rangle$, $Y = \langle \underline{A}, \underline{B} \rangle$, with optimal solution $Z = \langle A, B \rangle$. However, as in this case, we can just as well assume that the optimal solution takes both. If $x_n \neq y_m$, then the optimal solution cannot take both. Hence, it either does not take the last of X or does not take the last of Y . It might not take the last of either, but this is included in both the other two cases. See [Section 23.3](#) for a further answer.

(b) The loop over subinstances is then changed as follows.

```

% Solve instance  $\langle \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \rangle$  and fill in table entry  $\langle i, j \rangle$ .
if  $x_i = y_j$  then
     $birdAdvice[i, j] = 3$ 
    %  $optSol[i, j] = optSol[i - 1, j - 1] + x_i$ 
     $cost[i, j] = cost[i - 1, j - 1] + 1$ 
else
    % Try possible bird answers.
    % cases  $k = 1, 2$ 

```

```

    %  $optSol_1 = optSol[i - 1, j]$ 
     $cost_1 = cost[i - 1, j]$ 
    %  $optSol_2 = optSol[i, j - 1]$ 
     $cost_2 = cost[i, j - 1]$ 
    % end cases
    % Having the best,  $optSol_k$ , for each bird's answer  $k$ ,
    we keep the best of these best.
     $k_{max} = a \in [1, 2]$  that maximizes  $cost_k$ 
    %  $optSol[i, j] = optSol_{k_{max}}$ 
     $cost[i, j] = cost_{k_{max}}$ 
     $birdAdvice[i, j] = k_{max}$ 
end if

```

24.4.2 The running time is the number of subinstances times the number of possible bird answers, and the space is the number of subinstances. The number of subinstances is $\Theta(n^2)$, and the bird has $K = 3$ possible answers for you. Hence, the time and space requirements are both $\Theta(n^2)$.

Chapter 25. Designing Dynamic Programming Algorithms via Reductions

25.0.2 Given an instance $E = \{e_1, e_2, \dots, e_n\}$ of the elephant problem, we map this to an instance $\langle X, Y \rangle$ of the LCS problem as follows. Each elephant will be distinct. Let $X = \langle x_1, \dots, x_n \rangle$ be the elephants sorted by weight w_i , and let $Y = \langle y_1, \dots, y_m \rangle$ be the same sorted by smartness s_i . A solution to LCS is a subsequence $Z = \langle z_1, \dots, z_l \rangle$ that is common to both X and Y . Note that Z is a subset of elephants $S \subseteq E$ for which bigger is smarter. This is because Z is sorted both with respect to weight and with respect to intelligence. The only difference between these problems is that the cost (or success) of an LCS solution is simply the length of Z , while for the elephant problem it is the sum of the values of the elephants. Hence, for this to work, the LCS problem needs to be generalized to have weights on the letters. But this would not change the dynamic programming algorithm at all.

Chapter 27. Solution Is a Tree

- 27.2.1**
1. An AVL tree of height h has left and right subtrees of heights either $\langle h - 2, h - 1 \rangle$, $\langle h - 1, h - 1 \rangle$, or $\langle h - 1, h - 2 \rangle$.
 2. The bird tells me whether the subtree heights are $\langle h - 2, h - 1 \rangle$, $\langle h - 1, h - 1 \rangle$, or $\langle h - 1, h - 2 \rangle$. They also tell me which value a_k will be at the root. I can then ask the friends for the best left and right subtrees of the specified height.
 3. In each of these three cases, the heights are within 1.
 4. The complete set of subinstances is as follows. Recall that in [Chapter 13](#) we proved that the minimum height of an AVL tree with n nodes is $h = \log_2 n$ and that its maximum height is $h = 1.455 \log_2 n$. Hence, the complete set of subinstances is $S = \{ \langle h; a_i, \dots, a_j; p_i, \dots, p_j \rangle \mid 1 \leq i \leq j \leq n, h \in [\log_2(j -$

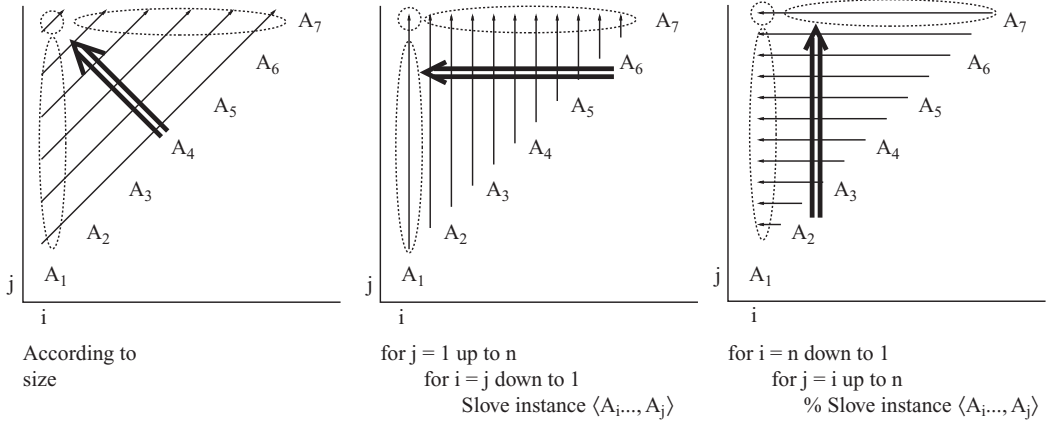


Fig. E.8

$i + 1) \dots 1.455 \log_2(j - i + 1)]\}$. The table is a three-dimensional $\Theta(n \times n \times \log n)$ box.

5. The table has size $\Theta(n \times n \times \log n)$. The bird can give $3 \cdot n$ different answers. Hence, the running time is $\Theta(n^3 \log n)$.
6. In the original problem, the height was not fixed. To solve this problem, we could simply run the previous algorithm for each h and take the best of the resulting AVL trees. However, after running the previous algorithm once, the table already contains the cost of the best AVL for each of the possible heights h . To find the best overall AVL tree, we need only compare those listed in the table.

27.4.1 algorithm *Parsing*($\langle G, T_{start}, a_1, \dots, a_n \rangle$)

(pre-cond): G is a Chomsky normal form grammar, T_{start} is a nonterminal, and s is the string $\langle a_1, \dots, a_n \rangle$ of terminal symbols.

(post-cond): P , if possible, is a parsing that generates s starting from T_{start} using G .

begin

% Table: $optSol[h, i, j]$ would store an optimal solution for $\langle G, T_h, a_i, \dots, a_j \rangle$, namely, a parsing for $a_i, \dots, a_j$ starting with nonterminal T_h . Instead, we store only the bird's advice for the subinstance and the cost of its solution.

$table[|V|, n, n]$ *birdAdvice, cost*

% The case $s = \epsilon$ is handled separately

if $n = 0$ then

if $T_{start} \Rightarrow \epsilon$ is a rule then

$P =$ the parsing applies this one rule.

```

else
     $P = \emptyset$ 
end if
return( $P$ )
end if

```

% Base cases: The base cases are when the string to parse consists of only one character a_i .

For $i = 1$ to n

For each nonterminal T_h

If there is a rule $r_q = "A_q \Rightarrow b_q"$, where A_q is T_h and b_q is a_i , then

$birdAdvice[h, i, i] = \langle q, ? \rangle$

$cost[h, i, i] = 1$

else

$birdAdvice[h, i, i] = \langle ?, ? \rangle$

$cost[h, i, i] = 0$

end if

end loop

end loop

% General cases: Loop over subinstances in the table.

for $size = 2$ to n % length of substring $\langle a_i, \dots, a_j \rangle$

for $i = 1$ to $n - size + 1$

$j = i + size - 1$

For each nonterminal T_h , i.e., $h \in [1 \dots |V|]$

% Solve instance $\langle G, T_h, a_i, \dots, a_j \rangle$, and fill in table entry $\langle h, i, j \rangle$.

% Loop over possible bird answers.

for each rule $r_q = "A_q \Rightarrow B_q C_q"$ for which A_q is T_h

for each split in the string $k = i$ to $j - 1$

% Ask a friend if you can generate

$\langle a_i, \dots, a_k \rangle$ from B_q .

% Ask another friend if you can

generate $\langle a_{k+1}, \dots, a_j \rangle$.

from C_q

$cost_{\langle q, k \rangle} = \min(cost[B_q, i, k], cost[C_q, k + 1, j])$

end for

end for

% Take the best bird answer, i.e., one of cost one if s can be generated.

$\langle q_{min}, k_{max} \rangle = \text{a } \langle q, k \rangle \text{ that maximizes } cost_{\langle q, k \rangle}$

$birdAdvice[h, i, j] = \langle q_{min}, k_{max} \rangle$

$cost[h, i, j] = cost_{\langle q_{max}, k_{max} \rangle}$

```

        end for
    end for
end for
% Constructing the solution P
if(cost[1, 1, n] = 1) then    % i.e., if s can be generated from T_start
    P = ParsingWithAdvice( $\langle G, T_{start}, a_1, \dots, a_n \rangle$ , birdAdvice)
else
    P =  $\emptyset$ 
end if
return (P)
end algorithm

```

Time and space requirements: The running time is the number of subinstance times the number of possible bird answers, and the space is the number of subinstances. The number of subinstances indexing your table is $\Theta(|V|n^2)$, namely, $Table[h, i, j]$ for $h \in V$ and $1 \leq i \leq j \leq n$. The number of answers that the bird might give you is at most $O(mn)$, namely, $\langle q, k \rangle$ for each of the m rules r_q and the split $k \in [1 \dots n - 1]$. This gives $time = O(|V|n^2 \cdot mn)$. If the grammar G is fixed, then the time is $\Theta(n^3)$.

A tighter analysis would note that the bird would only answer q for rules $r_q = "A_q \Rightarrow B_q C_q,"$ for which the left-hand side A_q is the nonterminal T_h specified in the instance. Let m_{T_h} be the number of such rules. Then the loop over nonterminals T_h and the loop over rules r_q would not require $|V|m$ time, but $\sum_{T_h \in V} m_{T_h} = m$. This gives a total time of $\Theta(n^3 m)$.

Chapter 28. Reductions and NP-Completeness

28.2.4 The first is easy. $InstanceMap(I_1)$ simply maps each instance $I_1 \in S_1 \subseteq S_2$ of P_1 to itself, I_1 , which is a valid instance of P_2 . The second is much harder, because $InstanceMap(I_2)$ must map each instance $I_2 \in S_2$ of P_2 to some instance I_1 within the restricted set S_1 .

28.2.5 The running time $Time(Alg_{oracle})$ is measured as a function of its own input size, namely, $\Theta(2^{|I_{oracle}|^{1/3}})$. But because $|I_{oracle}| = |I_{alg}|^2$, this same time is $\Theta(2^{|I_{alg}|^{2/3}})$, in terms of Alg_{alg} 's input size. The extra $O(|I_{alg}|^3)$ time for the mappings is not substantial. Hence, Alg_{alg} 's total running time is $\Theta(2^{n^{2/3}})$.

Similarly, if Alg_{oracle} runs in polynomial time, namely, $Time(Alg_{oracle}) = \Theta(|I_{oracle}|^c)$, then so does Alg_{alg} , namely, $Time(Alg_{alg}) = \Theta(|I_{alg}|^{2c} + |I_{alg}|^3)$, but note that the polynomial is a different one.

28.3.3

Step 5: Given a graph G_{COL} that we want to color, we construct an instance $\langle G_{Ind}, N_{Ind} \rangle = Instancemap(G_{COL})$ to give to the independent set oracle as follows. As said, G_{Ind} will have a node for each pair $\langle u, c \rangle$ where u is a node of G_{COL} and c is one of the three colors. For each node $u \in G_{COL}$, we put a triangle

of edges around $\langle u, \text{red} \rangle$, $\langle u, \text{blue} \rangle$, and $\langle v, \text{green} \rangle$. For each edge $\langle u, v \rangle \in G_{COL}$, we put three parallel edges between $\langle u, c \rangle$ and $\langle v, c \rangle$, for each color c . The size of the required independent set will be the number of nodes, $N_{Ind} = |V_{COL}|$, in the graph G_{COL} .

Step 6: Given an independent-set solution S_{Ind} to G_{Ind} of size $|V_{COL}|$, we construct a coloring S_{COL} for G_{COL} by coloring u with color c if node $\langle u, c \rangle$ is in the independent set.

Step 7: We now show that if S_{Ind} is a valid independent set of size $|V_{COL}|$, then S_{COL} is a valid 3-coloring. First we show that it is impossible for a node to be given more than one color, because the edge between $\langle u, c \rangle$ and $\langle u, c' \rangle$ prevents both of these nodes being in the independent set. Because the independent set is of size $|V_{COL}|$ and no node u appears more than once, it follows that every node u appears exactly once. Hence, every node u is given a color. Finally, we show that the nodes in the edge $\langle u, v \rangle \in G_{COL}$ cannot both have the color c , because the edge between $\langle u, c \rangle$ and $\langle v, c \rangle$ prevents both of these nodes from being in the independent set.

Step 8: Given a coloring S_{COL} for G_{COL} , we construct an independent set S_{Ind} for G_{Ind} of size $|V_{COL}|$ by putting node $\langle u, c \rangle$ in the independent set if u is colored c .

Step 9: We now show that if S_{COL} is a valid 3-coloring, then S_{Ind} is a valid independent set. We need to show that for each edge in G_{Ind} , both nodes are not in S_{Ind} . There is an edge between $\langle u, c \rangle$ and $\langle u, c' \rangle$, but u cannot have both colors c and c' . There is an edge between $\langle u, c \rangle$ and $\langle v, c \rangle$, but u and v cannot both have color c .

Chapter 34. Existential and Universal Quantifiers

- 34.0.2**
1. $\forall x \exists y x + y = 5$ is true. Let x have an arbitrary real value, and let $y = 5 - x$. Then $x + y = 5$.
 2. $\exists y \forall x x + y = 5$ is false. Let y have an arbitrary real value, and let $x = 6 - y$. Then $x + y \neq 5$.
 3. $\forall x \exists y x \cdot y = 5$ is false. Let $x = 0$. Then y must be $\frac{5}{0}$, which is impossible.
 4. $\exists y \forall x x \cdot y = 5$ is false. Let y have an arbitrary real value, and let $x = \frac{6}{y}$ if $y \neq 0$ and $x = 0$ if $y = 0$. Then $x \cdot y \neq 5$.
 5. $\forall x \exists y x \cdot y = 0$ is true. Let x have an arbitrary real value, and let $y = 0$. Then $x \cdot y = 0$.
 6. $\exists y \forall x x \cdot y = 0$ is true. Let $y = 0$, and let x have an arbitrary real value. Then $x \cdot y = 0$.
 7. $[\forall x \exists y P(x, y)] \Rightarrow [\exists y \forall x P(x, y)]$ is false. Let $P(x, y) = [x + y = 5]$. Then, as already seen, the first is true and the second is false.
 8. $[\forall x \exists y P(x, y)] \Leftarrow [\exists y \forall x P(x, y)]$ is true. Assume the right side is true. Let y_0 be the y for which $[\forall x P(x, y)]$ is true. We prove the left side as follows. Let x have an arbitrary real value, and let $y = y_0$. Then $P(x, y_0)$ is true.

9. Sorry, not provided.
10. Sorry, not provided.

Chapter 35. Time Complexity

35.1.1 The number of operations is $T = 24 \times 60 \times 60 \times 10^6 = 8.64 \times 10^{10}$. We have $n_1 = T^{1/4} = 542$, $n_2 = \log_2 T = \frac{\log T}{\log 2} = 36$.

- 35.1.4**
1. We first prove that $g + h = \Theta(\max(g, h))$ as follows. $\max(g, h) \leq g + h$, assuming that both g and h are positive and $g + h \leq 2 \max(g, h)$.
 2. One can set k to absolutely minimize $f(n, k)$ by setting f 's derivative wrt k equal to zero and solving for k . Sometimes this is hard. Because $f(n, k) = \Theta(\max(\frac{n}{\log k}, \frac{k}{\log k}))$ and we do not care about the multiplicative constant, let us instead set k in order to minimize $\max(\frac{n}{\log k}, \frac{k}{\log k})$. Observe that if k is very small, then the first term, being very big, dominates. Hence, we can make the whole expression smaller by increasing k . Similarly, if k is big, the second term dominates the expression, and we can decrease it by decreasing k . So, for the optimal solution the two terms should be roughly the same. In this case, $\frac{n}{\log k} = \frac{k}{\log k}$ gives $n = k$ and $f(n, k) = \frac{n+k}{\log k} = \frac{n}{\log n}$. This is (asymptotically) the best result, because decreasing k increases the first term and increasing k increases the second.
 3. Sorry, not included.

- 35.2.1**
1. $\exists A, \forall I, \text{Works}(\text{Sorting}, A, I)$. We know that there at least one algorithm, e.g., $A = \text{merge sort}$, that works for every input instance I .
 2. $\forall A, \exists I, \neg \text{Works}(\text{Halting}, A, I)$ We know that contrary statement is true. Every algorithm fails to work for at least one input instance I .
 3. $\exists P, \forall A, \exists I, \neg \text{Works}(P, A, I)$
 4. It says that every input has some algorithm that happens to output the right answer. It is true. Consider an arbitrary instance I . If on instance I , *Halting* happens to say yes, then let A be the algorithm that simply halts and says yes. Otherwise, let A be the algorithm that simply halts and says no. Either way, A works for this instance I .
 5. It says that every algorithm correctly solves some problem. This is not true, because some algorithms do not halt on some input instances. We prove the complement $\exists A, \forall P, \exists I, \neg \text{Works}(P, A, I)$ as follows. Let A be an algorithm that runs forever on some instance I' . Let P be an arbitrary problem. Let I be an instance I' on which A does not halt. Note that $\text{Works}(P, A, I)$ is not true.

Chapter 36. Logarithms and Exponentials

36.0.1 $a^3 \times a^5 = a^8$, $3^a \times 5^a = 15^a$, $3^a + 5^a = ?$, $2^{5 \log_4 n + 7} = [4^{0.5}]^{6 \log_4 n} \times 2^7 = [4^{\log_4 n}]^3 \cdot 128 = 128n^3$, $n^{3/\log_2 n} = [2^{\log_2 n}]^{3/\log_2 n} = 2^3 = 8$.

Chapter 37. Asymptotic Growth

37.1.4 Exercise 37.0.2 proves that $3 \log n \ll n$ for sufficiently big n . Hence, $n^2 \leq 3n^2 \log n \leq n \cdot n^2 = n^3$.

37.1.5 $3^{4n} \gg 4^{3n} \gg \frac{2^{0.001n}}{n^{100}} \gg 100n^{100} \gg \log^{1,000} n$.

37.2.2

1. $14n^9 + 5,000n^7 + 23n^2 \log n \in O(n^9)$: Let $c = 15$ and $n_0 = 100$. For all $n \geq 100$, we have $f(n) = 14n^9 + 5,000n^7 + 23n^2 \leq 14n^9 + \frac{n^2}{2}n^7 + \frac{n^6}{2}n^2 \log n \leq 14n^9 + n^9 = c \cdot g(n)$.
2. $2n^2 - 100n \in \Theta(n^2)$: Let $c_1 = 1$, $c_2 = 2$, and $n_0 = 100$. For all $n \geq 100$, $c_1 g(n) = 1n^2 = 2n^2 - n \cdot n \leq 2n^2 - 100n = f(n) \leq 2n^2 = c_2 g(n)$.
3. $14n^8 - 100n^6 \notin O(n^7)$: Let c and n_0 be arbitrary values given to us by some adversary. We then let $n = \max(10, c, n_0)$. Then we demonstrate that $f(n)$ is too big. Because $n \geq 10$, we have $100n^6 \leq n^8$. This gives $f(n) = 14n^8 - 100n^6 \geq 14n^8 - 1n^8 > n \cdot n^7 \geq c \cdot n^7$.
4. $14n^8 + 100n^6 \notin \Theta(n^9)$: Let c_1, c_2 , and n_0 be arbitrary values given to us by some adversary. Let us make $n = \max(15/c_1, 11, n_0)$. Then we demonstrate that $f(n)$ is too small: $c_1 g(n) = c_1 n^9 = n \cdot c_1 n^8 \geq \frac{15}{c_1} \cdot c_1 n^8 = 14n^8 + n^8 = 14n^8 + n^2 \cdot n^6 \geq 14n^8 + (11)^2 \cdot n^6 > 14n^8 + 100n^6 = f(n)$.
5. $2^{n+1} \in O(2^n)$: Let $c = 2$ and $n_0 = 0$. For all $n \geq 0$, $f(n) = 2^{n+1} \leq 2 \times 2^n = c \times g(n)$.
6. $2^{2n} \notin O(2^n)$: Let c and n_0 be arbitrary values. Let $n = \max(1 + \log_2 c, n_0)$. Then we have that $f(n) = 2^{2n} = 2^n \cdot 2^n > c \cdot g(n)$.

37.2.8

1. $x = 7y^3(\log_2 y)^{18} \in y^{3+o(1)} = [y^3, y^{3+\epsilon}]$. Solving this gives $y = x^{1/(3+o(1))}$.
2. Substituting in $y = x^{1/(3+o(1))}$ gives $x = 7y^3(\log_2 y)^{18} = 7y^3(\log_2 x^{1/(3+o(1))})^{18} = \Theta(y^3(\log_2 x)^{18})$. Solving this gives $y = \Theta(x^{1/3}/(\log x)^6)$.

Chapter 38. Adding-Made-Easy Approximations

38.2.1 His first such step takes him half an hour, his second a quarter, his third an eighth, ... for a total of only $\sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1$ hour. Given that he travels one kilometer at one kilometer an hour, this is reasonable.

38.2.4

1. The function $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$ is $2^{\Omega(n)}$, because it is bounded between 2^n and 2^{2n} . Let $n = 2^k$. Then $\lceil \log_2 n \rceil = k$ and $f(n) = 2^{2^k} = 2^n$, but for $n' = n + 1$ we have $\lceil \log_2 n' \rceil = k + 1$ and $f(n') = 2^{2^{k+1}} = 2^{2n}$.
2. We show $\sum_{i=1}^n f(i) \neq \Theta(f(n))$ as follows. With $n = 2^k$, both functions are more or less constant from $f(\frac{n}{2})$ to $f(n)$. Because they behave like arithmetic functions within this range, the adding-made-easy techniques give that $\sum_{i=1}^n f(i) = \Theta(n \cdot f(n))$ and not $\Theta(f(n))$.

3. We show that $f(n) = 2^{\lfloor \log \log n \rfloor} \in n^{\Theta(1)-1}$ by bounding it between $\sqrt{n}$ and n . Let $n = 2^{2^k}$. Then $\lfloor \log \log n \rfloor = k$ and $f(n) = 2^{2^k} = n$, but for $n' = n - 1$ we have $\lfloor \log \log n' \rfloor = k - 1$ and $f(n') = 2^{2^{k-1}} = \sqrt{n}$.
4. We show that $\sum_{i=1}^n f(i) \notin \Theta(n \cdot f(n))$ as follows. Again let $n = 2^{2^k}$, so that $f(n) = n$, yet every previous term is most $\sqrt{n}$. The total of $\sum_{i=1}^n f(i)$ then is at most $(n - 1) \cdot \sqrt{n} + n$. Because the last term $f(n)$ is so much bigger than the previous ones, the total is not $\Theta(n \cdot f(n))$, which is $\Theta(n^2)$.
5. The function $f(n) = 2^{\lfloor \frac{1}{2} \cos(\pi \log_2 n) + 1.5 \rfloor \cdot n}$ is a geometric counterexample squeezed between 2^n and 2^{2n} , just as $f(n) = 2^{\lfloor \log_2 n \rfloor}$ is:

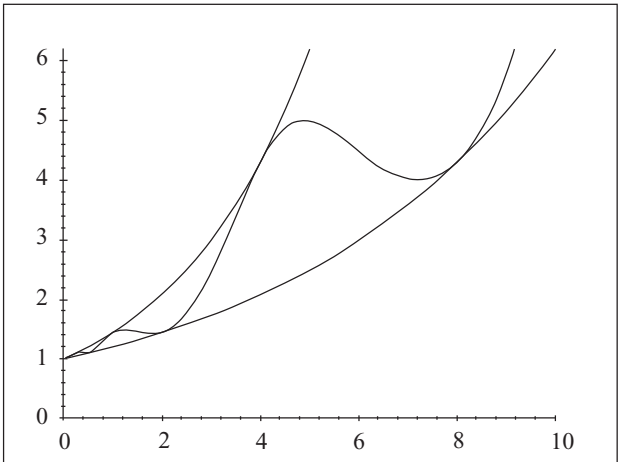


Fig. E.9

Chapter 39. Recurrence Relations

39.1.1 Examples:

$T(n)$	$\frac{\log a}{\log b}$	c	$d \text{ vs } -1$	Dom.	Rule	Solution
$2T(\frac{n}{2}) + n$	$\frac{\log_2 2}{\log_2 2} = 1$	1	$0 >$	All	$\Theta(f(n) \log n)$	$\Theta(n \log n)$
$2T(\frac{n}{2}) + 1$	$\frac{\log_2 2}{\log_2 2} = 1$	0		Base	$\Theta(n^{\log a / \log b})$	$\Theta(n)$
$4T(\frac{n}{3}) + \Theta(n^3 / \log^3 n)$	$\frac{\log_2 4}{\log_2 3} = 2$	3		Top	$\Theta(f(n))$	$\Theta(n^3 / \log^3 n)$
$32T(\frac{n}{4}) + \Theta(\log n)$	$\frac{\log_2 32}{\log_2 4} = \frac{5}{2}$	0		Base	$\Theta(n^{\log a / \log b})$	$\Theta(n^{2.5})$
$27T(\frac{n}{3}) + \Theta(n^3 \log^4 n)$	$\frac{\log_3 27}{\log_3 3} = \frac{3}{1}$	3	$4 >$	All	$\Theta(f(n) \log(n))$	$\Theta(n^3 \log^5 n)$
$8T(\frac{n}{4}) + \Theta((n / \log n)^{1.5})$	$\frac{\log_2 8}{\log_2 4} = \frac{3}{2}$	$\frac{3}{2}$	$-1.5 <$	Base	$\Theta(n^{\log a / \log b})$	$\Theta(n^{1.5})$
$4T(\frac{n}{2}) + \Theta(n^2 / \log n)$	$\frac{\log_2 4}{\log_2 2} = 2$	2	$-1 =$	Exercise 27.2.4 gives $\Theta(n^c \cdot \log \log n) = \Theta(n^2 \log \log n)$		

39.2.1 Plugging $Fib(n) = \alpha^n$ into $Fib(n) = Fib(n - 1) + Fib(n - 2)$ gives that $\alpha^n = \alpha^{n-1} + \alpha^{n-2}$. Dividing through by α^{n-2} gives $\alpha^2 = \alpha + 1$. Solving this gives that either $\alpha = \frac{1+\sqrt{5}}{2}$ or $\alpha = \frac{1-\sqrt{5}}{2}$. Any linear combination of these two solutions will also be a valid solution, namely $Fib(n) = c_1 \cdot (\alpha_1)^n + c_2 \cdot (\alpha_2)^n$.

Using the fact that $Fib(0) = 0$ and $Fib(1) = 1$ and solving for c_1 and c_2 gives that

$$Fib(n) = \frac{1}{\sqrt{5}} \left[\left[\frac{1 + \sqrt{5}}{2} \right]^n - \left[\frac{1 - \sqrt{5}}{2} \right]^n \right].$$

39.2.2 Unwinding:

1. $T(n) = T(n-1) + n$: Because $a = 1$ and $c > 1$, the table give $T(n) = \Theta(n \cdot f(n)) = \Theta(n^2)$. Unwinding gives $T(n) = n + T(n-1) = n + (n-1) + T(n-2) = n + (n-1) + (n-2) + T(n-3) = n + (n-1) + (n-2) + \dots + (n-i+1) + T(n-i) = n + (n-1) + (n-2) + \dots + 1 = \Theta(n^2)$.
2. $T(n) = 2 \cdot T(n-1) + 1$: Because $a = 2$, the table gives $T(n) = \Theta(a^{\frac{n}{b}}) = \Theta(2^n)$. Unwinding gives $T(n) = 1 + 2T(n-1) = 1 + 2 + 4T(n-2) = 1 + 2 + 4 + \dots + 2^{i-1} + 2^i T(n-i) = 1 + 2 + 4 + \dots + 2^{n-1} + 2^n = \Theta(2^n)$.

39.2.4 Suppose $f(n) = n^c \cdot \log^d n$ and $c = \frac{\log a}{\log b}$; then $T(n) = \Theta(\sum_{i=0}^h a^i f(n/b^i)) = \Theta(\sum_{i=0}^h a^i (n/b^i)^c \log^d(n/b^i)) = \Theta(n^c \cdot \sum_{i=0}^h (1)^i [\log(n/b^i)]^d)$. The expression n/b^i takes on the values $n, n/b, n/b^2, \dots, 1$. Reversing this order gives $T(n) = \Theta(n^c \cdot \sum_{j=0}^h [\log(b^j)]^d) = \Theta(n^c \cdot \sum_{j=0}^h [j \log b]^d)$. Here $[\log b]^d$ is a constant that we can hide in the Theta. This gives $T(n) = \Theta(n^c \cdot \sum_{j=0}^h j^d)$. The adding-made-easy approximations state that this sum is arithmetic as long as $d > -1$. In this case, the total is $T(n) = \Theta(n^c \cdot h \cdot h^d) = \Theta(n^c \cdot \log^{d+1} n) = \Theta(f(n) \log n)$. If $d = -1$, then we get the harmonic sum $T(n) = \Theta(n^c \cdot \sum_{j=0}^h \frac{1}{j}) = \Theta(n^c \cdot \log h) = \Theta(n^c \log \log n)$. If $d < -1$, then the sum has a bounded tail, giving $T(n) = \Theta(n^c \cdot \Theta(1)) = \Theta(n^{\log a / \log b})$.

Conclusion

The overall goal of this entire text has been to teach skills in abstract thinking. I hope that it has been fruitful for you. Good luck at applying these skills to new problems that arise in other courses and in the workplace.

Index

- abstract data type (ADT), 1, 47
 - functions vs., 47
 - merging with queue, 61
 - specifications/implementations, 47
 - AVL tree, 53
 - binary search tree, 53
 - dictionary, 51
 - graphs, 51
 - link list implementation, 55
 - list, 47
 - orders, 53
 - parsing with stack, 62
 - priority queue, 50
 - queue, 49
 - recursive definition of tree, 169
 - set, 50
 - set system, 50
 - simple types, 48
 - stack, 48
 - trees, 52
 - union-find set system, 53
- Ackermann's function, 162
 - algorithm, 162
 - crashing, 163
 - recurrence relation, 162
 - running time, 163
 - solving, 162
- adding-made-easy approximations, 537
 - examples, 537
 - exercise solutions, 586
 - proofs, 538
 - analytical functions, simple, 538
 - arithmetic sums, 536
 - close to harmonic, 537
 - functions, without basic form, 537
 - geometric sums, simple, 534
 - harmonic sum, 537
 - ratio between terms, 534
 - solution classes, 530
 - technique, 530
- ADT, *See* abstract data type
- algorithms, *See also* recursive algorithms
 - Ackermann's function, 162
 - best AVL tree bird-and-friend algorithm, 396
 - bipartite matching using network flow, 419
 - brute force algorithm, 69, 294
 - defined, 1
 - Dijkstra's shortest-weighted path algorithm, 253
 - dynamic programming algorithms and examples, 338
 - all pairs, matrix multiplication, 397
 - best AVL tree, 397
 - chains of matrix multiplications, 390
 - context-free grammar parsing, 398
 - dynamic programming algorithms. *via* reductions, 338
 - longest increasing contiguous subsequence example, 369
 - longest increasing subsequence example, 370
 - longest-common-sequence problem, 364
 - shortest weight path, directed leveled graph example, 337
 - weighted job/event scheduling problem, 371
 - Euclid's greatest common divisor (GCD) algorithm, 80
 - graph algorithms
 - expander graphs, 428
 - max cut problem, 428
 - minimum spanning tree, 314
 - network flows, 268
 - shortest weight path, directed leveled graph example, 337
 - 3-Colouring, 408, 419
 - graph search algorithms, 243
 - breadth-first search, shortest path, 248, 250
 - depth-first search, 259
 - generic search algorithm, 243
 - partial order linear ordering, 265
 - depth-first search algorithm, 265
 - easy but slow algorithm, 265
 - recursive depth-first search, 263
 - greedy algorithms, 294, 330, 367, 395
 - hill-climbing algorithm, 290
 - iterative algorithms, 9, 13, 21

- iterative sorting algorithms, 74, 75, 78
- looking forward vs. backward, recursive algorithms, 134
- meta-algorithms, xiii, 2
- parsing algorithm specifications, 199
- randomized algorithms, 428
- sorting and selection algorithms, 423
- steepest-ascent hill-climbing algorithm, 149
- Strassen's matrix multiplication, 161, 284
- time/space complexity, 83, 508, 518, 579, 582
- analytical functions, simple, 538
- arithmetic sums, 536 *See also* adding-made-easy approximations,
- asymptotic growth
 - asymptotic growth rate, 518
 - asymptotic notation, 88, 518, 519
 - BigOh definition, 88, 525
 - Little Oh definition, 525
 - Little Omega definition, 525
 - loop invariant for lower bounds, 80
 - Theta definition, 88, 518
 - exercise solutions, 586
 - function classification, 518
 - growth rates classes, 518
 - purpose, 518
- AVL tree, 53
 - best AVL tree, 395
 - best AVL tree problem, 396
- binary search, 24
 - for cost in optimization problems, 407
 - narrowing the search space example, 22
 - returning index example, 91
 - returning yes/no, 93
 - trees, 53, 64
 - ADT, 50
 - balanced, 53
 - basic steps, 64
 - best binary search tree problem, 395
 - coding/implementation details, 65
 - ending, 65
 - establishing loop invariant, 64
 - exit condition, 65
 - is tree a binary tree example, 177
 - loop invariant, 64
 - main steps, 65
 - maintain loop invariant, 65
 - make progress, 65
 - measure of progress, 64
 - nodes in binary search tree example, 174
 - running time, 65
 - specifications, 64
 - typical errors, 31
- binary trees, recursion, 209
- BigOh definition, 526
- bipartite edges, 261
- bipartite matching using network flow, 419
- breadth-first search, 248
 - shortest path, 248, 249
 - code, 251
 - exiting loop, 252
 - initial code, 252
 - loop body, 244
 - loop invariant, 245
 - maintaining loop invariant, 245
 - optimization problem, 248
 - shortest path problem, 248
 - shortest path proof, 251
- bucket sort by hand, 74
 - basic steps, 74
 - exit condition, 75
 - loop invariant, 75
 - maintain loop invariant, 75
 - specifications, 74
- chains of matrix multiplications, 390
 - failed dynamic programming algorithm, 391
 - failed greedy algorithm, 391
 - little bird question, 391
 - optimal solution construction, 394
 - optimal solution cost, cost for subinstances, 392
 - recursive structure, 392
 - reduced to subinstance, 392
 - set of subinstances called, 392
 - table fill order, 392
 - table indexed by subinstances construction, 392
 - time/space requirements, 394
- coloring the plane, 33
 - basic steps, 33
 - coding/implementation details, 34
 - ending, 34
 - establishing loop invariant, 34
 - exit condition, 34
 - loop invariant, 34
 - main steps, 34
 - maintain loop invariant, 34
 - measure of progress, 34
 - running time, 34
 - special cases, 34
 - specifications, 33
- computational complexity, 1
 - asymptotic notations, 88, 518, 525
 - formal proof of correctness, 549
 - nondeterministic polynomial-time decision problems (NP), 408
 - time/space complexity, 83, 425, 508, 519, 548, 584
- context-free grammar parsing, 198
 - abstract data type parsing, 199
 - correctness proof, 205
 - dynamic programming example
 - abstract data type parsing, 399
 - base cases, 400
 - help from friend, 400

- little bird question, 399
 - not look ahead one, 399
 - number of parsings, 399
 - optimal solution construction, 400
 - parsing problem, 399
 - set of subinstances, 400
 - table fill order, 400
 - table indexed by subinstances construction, 400
- exercise solutions, 578
- expression parsing, 199
- GetExp* code, 201
- GetExp* expressions, 201
- GetExp*, *GetTerm*, *GetFact* examples, 201
- GetFact* code, 203
- GetTerm* code, 203
- grammar, 198
- look ahead one, 198, 205
- nonterminals, 199
- parsing algorithm specifications, 199
- rules, 199
- running time, 204
- semantics and, 198
- string derivation, 198
- syntax and, 198
- terminals, 199
- tree of stack frames, 204
- correctness, 2
 - formal proof of correctness, 549
 - for context-free grammar parsing, 199
 - for depth-first search, 265
 - for dynamic programming, 353
 - for greedy algorithms, 321
 - for recursive algorithms, by strong induction, 147
- counting sort, 75
 - basic steps, 76
 - code, 77
 - establishing loop invariant, 77
 - exit condition, 77
 - loop invariant, 77
 - main step, 77
 - maintain loop invariant, 77
 - running time, 77
 - specifications, 75
- cyclic edges, 261
- Davis–Putnam, 331
- depth-first search, 247, 259
 - code, 259
 - edges classification, 261
 - establish/maintain loop invariant, 260
 - generic search algorithm changes, 259
 - loop invariants, 259
 - recursive, 263
 - time stamping, 262
- deterministic finite automation (DFA), 35
 - addition example, 39
 - applications, 35
 - calculator example, 40
 - compiling iterative program into DFA, 37
 - division example, 40
 - dynamic programming, 41
 - longest block of ones example, 41
 - longest increasing contiguous subsequence, 42
 - longest increasing subsequence, 42
- dictionary ADT, trees, data structure, 52–53
- Dijkstra’s shortest-weighted-path algorithm, 253
 - code, 255
 - exiting loop, 258
 - initial code, 258
 - loop body, 255
 - loop invariant, 255
 - maintaining LI1, 256
 - maintaining LI2, 256
 - problem specifications, 253
 - shortest distance approximation, 253
 - shortest path proof, 251
- dynamic programming, 336, 444–453
 - algorithm using trusted bird and friend, 445
 - can’t reduce, 453
 - cutting task into two independent tasks, 451–452
 - exercise solutions, 585, 586
 - G as the graph of life, 450
 - nodes of G are subinstances, 449–450
 - recursive backtracking, 336, 449
 - running time, 340
 - recursive backtracking algorithm, 445
 - reduction of any dynamic programming algorithm to *LeveledGraph*, 448
 - running time, 453
 - specifications, 445
 - steps in developing, 336
 - base cases, 344
 - code, 345
 - count subinstances, 343
 - final solution, 344
 - redundancy, 341
 - running time, 346
 - set of subinstances, 341
 - solution from subsolutions, 343
 - table fill order, 344
 - table indexed by subinstances construction, 343
 - subtle points, 346
- dynamic programming algorithms *via* reductions, 375
 - best path similarity, 375
 - bigger-is-smarter elephant problem, 379
 - event scheduling problem, 375
 - graph instance formation, 376
 - mapping back algorithm, 377
- dynamic programming algorithms and examples

- all pairs, matrix multiplication, 397
- best AVL tree, 395
 - best AVL tree problem, 396
 - best binary search tree, 395
- chains of matrix multiplications, 390
- context-free grammar parsing, 398
- dynamic programming algorithms. *via*
 - reductions, 375
- longest increasing contiguous subsequence
 - example, 369
- longest increasing subsequence example, 370
 - time/space requirements, 368
- longest-common-sequence problem, 364
 - base cases, 365
 - code, 366
 - greedy algorithm, 364
 - information about subinstance, 367
 - little bird possible answers, 364
 - little bird question, 365
 - longest common sequence, 364
 - optimal solution construction, 367
 - set of subinstances, 365
 - table fill order, 366
 - table indexed by subindexes construction, 365
 - time/space requirements, 368
- shortest weight path, directed leveled graph
 - example, 337
- weighted job/event scheduling problem, 371
 - failed algorithms, 371
 - greedy dynamic programming, 372
- edges classification, 261
 - back edges, 261
 - bipartite edges, 261
 - cross edges, 261
 - cyclic edges, 261
 - forward edges, 261
 - tree edges, 261
- Euclid's GCD algorithm, *See* [greatest common divisor algorithm](#)
 - bug, 86
 - code, 86
 - elementary school algorithm, 84
 - ending with postcondition, 85
 - Ethiopians, 87
 - have *code_{loop}*, 85
 - have (*loop-invariant*), 85
 - head in the right direction & return to path, 85
 - high school algorithm, 86, 87
 - LHS vs. RHS, 85, 86
 - loop invariant, 84
 - maintaining the loop invariant, 85, 86
 - make progress, 86
 - preconditions, 85
 - running time, 85, 86
- Euler cycle, 44
- event scheduling, 307
 - weighted job/event scheduling problem, 371
- existential/universal quantifiers, 500, 513
 - bound variables definition, 500
 - combining quantifiers, 500
 - exercise solutions, 585
 - expressions building, 501
 - free variables definition, 500
 - Loves* example, 499
 - negation, 502
 - quantifiers order, 502
 - relation definition, 501
 - representations, 499
 - variable domain, 502
- exponentials, 515
 - base, 517
 - exercise solutions, 585
 - ratio, 517
 - rules, 515
 - uses, 515
- fast Fourier transformation, 161
- Fast Fourier Transformations (FFT), 163–168
 - code, 165
 - $FFT(FFT(F(t))) = n \cdot F(t)$, 167
 - interpolations, 164
 - inverse sin-cos-FFT, 167
 - multiplying, 167
 - my input, 164
 - my output, 164
 - n values of x , 164
 - n th roots of unity, 167
 - n th roots of unity are special, 165
 - radians, 167
 - recursion, 164
 - rotating the plane, 166
 - special values, 164
 - splitting the input, 164
 - t^{th} value of x and t^{th} term/frequency, 167
- FFT, *see* [Fast Fourier Transformations](#)
- find-max two-finger algorithm example, 11
- forward/cross edges, 261
- friends level of abstraction, recursive algorithms, 135
 - base cases, 137
 - general input, 135
 - generalizing problem, 136
 - link to techniques, iterative algorithms, 137
 - minimizing number of cases, 136
 - running time, 137
 - size, 135
 - specifications, 135
- functions, *See also* [Ackermann's function](#)
 - abstract data types vs., 47
 - analytical functions, simple, 538
 - linear function, 523
 - quadratic function, 523
 - time/space complexity as, 508

- game of life, 380–389, *see also* graph of life
 - graph G from computation, 380
 - sequence of actions vs. a sequence of assertions, 5
- GCD, *see* Euclid's GCD algorithm
- GCD algorithm, *See* greatest common divisor (GCD) algorithm
- geometric sums, simple, 534
- GetExp* code, 201
- GetExp* expressions, 199
- GetExp*, *GetTerm*, *GetFact* examples, 201
- GetExp* reasoning, 201
- GetFact* code, 203
- GetFact* reasoning, 203
- GetTerm* code, 203
- global vs. local considerations, 270, 274, 291, 295, 322, 347
- graph algorithms
 - expander graphs, 428
 - max cut problem, 428
 - minimum spanning tree, 314
 - network flows, 268
 - shortest weight path, directed leveled graph
 - example, 337
 - 3-Colouring, 408, 415
- graph of life, 382–389, *see also* game of life
 - actions vs. landmarks, 382
 - algorithm 1 – maximize cash on day t , 387
 - algorithm 2 – maximize stock s at time t , 388
 - algorithm 3 – two types of states and atomic actions, 388
 - assertions, 382
 - buy stock, 389
 - cutting a task into two independent subinstances, 383
 - edge cost, 384
 - edge length, 384
 - examples, 385–389
 - finding optimal path, 384
 - forgetting, 383
 - G as the graph of life, 450
 - goal to find optimal path, 384
 - knapsack, 385
 - little bird, 384
 - longest common subsequence, 386
 - path of life, 382
 - printing neatly, 387
 - produce the solution *Sol*, 383
 - pseudocode execution, 382
 - reading the input I , 383
 - resource, 383
 - running time, 384
 - sell stock, 389
 - sitting on cash, 388
 - sitting on stock, 389
 - states and possible actions, 383
 - stock problem, 387
 - TM & DFA, 382
- graph search algorithms, 243
 - breadth-first search, shortest path, 247, 248
 - depth-first search, 259
 - Dijkstra's shortest-weighted-path algorithm, 253
 - exercise solutions, 575
 - generic search algorithm, 243
 - basic steps, 243
 - code, 245
 - exit condition, 246
 - exiting loop, 246
 - handling nodes order, 247
 - initial code, 246
 - loop body, 244
 - loop invariant, 244
 - maintaining loop invariant, 245
 - measure of progress, 245
 - reachability problem, 243
 - running time, 247
 - partial order linear ordering, 265
 - recursive depth-first search, 263
- graph theory problems, 243
- greatest common divisor (GCD) algorithm, 80
 - code, 82
 - ending, 80, 82
 - establishing loop invariant, 80
 - example, 82
 - exercise solutions, 570
 - exit condition, 82
 - iteration on general instance, 80, 81
 - loop invariant, 80
 - lower bound, 83
 - making progress, 81
 - recursive, 160
 - running time, 80
 - special cases, 81
 - specifications, 80
 - termination, 82
- greatest common divisor algorithm (GCD), *see* Euclid's GCD algorithm
- greedy algorithms, 294, 439–444
 - alternative proof, 303
 - brute force algorithm, 294
 - correctness proof, 299
 - using loop invariants, 296
 - definition of correct, 303
 - does not go down, 303
 - does not go up, 303
 - examples
 - game show, 295
 - interval cover problem, 310
 - job/event scheduling problem, 307
 - minimum spanning tree problem, 314
 - exercise solutions, 575
 - exiting loop, 440, 444

- first order logic, 304
- fix problems, 301
- fixity vs. adaptive priority, 304
- game, 304
- game as proof, 304
- globally optimal step, 303
- greedy choice, 295, 439
- greedy game, 304
- head in the right direction & return to path, 301
- initially $((pre \Rightarrow LI))$, 300
- instructions for modifying, 300
- locally optimal step, 303
- loop invariant, 297, 441
- loop invariants, types, 296
 - specifications, 294
- make S_I consistent, 301
- making change example, 300
- many cases, 301
- postcondition, 294
- $potentialVal(A_I)$, 303
- proof, 303
- proof globally optimal, 304
- proving, 304
- proving that she has a witness, 302
- specifications, 439
- taking a step, 300
- theorem, 303
- three players, 300
- tree of options, 301
- using, 304
- what we know, 300
- harmonic sum, 537
 - close to harmonic, 537
- heap sort/priority queues, 180
 - array implementation, balanced binary tree, 180
 - common mistakes, 186
 - completely balanced binary tree, 180
 - heap definition, 180
 - heapify problem, 181
 - code, 181
 - iterative algorithm, 182
 - recursive algorithm, 181
 - running time, 182, 183
 - specifications, 181
- heapsort problem, 184
 - algorithm, 184
 - array implementation, 184
 - code, 185
 - specification, 184
- makeheap problem, 183
 - iterative algorithm, 183
 - recursive algorithms, 183
 - running time, 184
 - specifications, 183
- priority queues, 186
- hill-climbing algorithm, *See also* [primal-dual hill climbing method](#); [steepest-ascent hill-climbing algorithm](#), 290
 - small local maximum, 270
 - algorithm, faulty, 272
 - algorithm fixing, 274
 - augmentation graph, faulty, 272
 - basic ideas, 270
 - counterexample, 274
 - local maximum, 274
- image drawing, *See* [recursive image drawing](#)
- information hiding, 47
- information theoretic lower bounds, 92, 93
- iterative algorithms, 9, 13
 - basic steps, 14
 - code structure, 9
 - coding/implementation details, 20
 - conservation of energy, 26
 - correctness proof, 10
 - differential equations, 27
 - ending, 19
 - establishing loop invariant, 18
 - exit condition, 19
 - find-max two finger algorithm example, 11
 - formal proof, 20
 - loop invariants, 28
 - loop invariants for, 9, 14, 16
 - main steps, 17
 - maintain loop invariant, 17
 - make progress, 17
 - measure of progress, 14
 - running time, 10, 20
 - search space narrowing, 22
 - binary search narrowing example, 24
 - special cases, 20
 - specification, 13
 - types of iterative algorithms, 21
 - case analysis, 23
 - insertion sort example, 24
 - more of input, 22
 - more of output, 22
 - selection sort example, 23
 - work done, 23
 - bubble sort example, 26
- iterative sorting algorithms
 - bucket sort by hand, 74
 - counting sort, 75
 - radix sort, 78
- job/event scheduling, *See* [event scheduling](#)
- k th smallest element example, 152
- Las Vegas model, 424
- linear function, 531
- linear programming, *See also* [network flow](#), 288
 - Euclidean space interpretation, 289

- example, 289
- formal specification, 289
- hill-climbing algorithm, 290
- matrix representation, 289
- network flows, 289
- running time, 292
- small local maximum, 291
- link list implementation, 55
 - adding node to end, 57
 - adding node to front, 56
 - deleting node, 60
 - hidden invariants, 56
 - initialize walk, 58
 - notation, 56
 - removing node from end, 57
 - removing node from front, 57
 - testing whether empty, 57
 - walking down linked list, 57
- Little Oh definition, 525, 527
- Little Omega definition, 525, 527
- logarithms, 515
 - base, 517
 - exercise solutions, 584
 - ratio, 517
 - rules, 515
 - uses, 515
- longest block of ones example, 41
- longest increasing contiguous subsequence
 - example, 369
- longest increasing subsequence example, 370
- longest-common-sequence problem, 241, 364
 - base cases, 365
 - code, 366
 - greedy algorithm, 364
 - little bird possible answers, 364
 - little bird question, 365
 - longest common sequence, 364
 - optimal solution construction, 367
 - set of subinstances, 365
 - table fill order, 366
 - table indexed by subindexes construction, 365
 - time/space requirements, 368
- looking forward vs. backward, recursive
 - algorithms, 134
 - algorithm, 134
- loop invariant for lower bounds, 88
 - asymptotic notation, 88
 - binary search returning index example, 91
 - binary search returning yes/no, 93
 - dynamic algorithms, 92
 - exercise solutions, 565
 - flipping a bit, 92
 - loop invariant argument, 89
 - lower bounds proof, state of art, 94
 - multiplexer example, 93
 - parity example, 92, 93
 - sorting example, 90
 - time complexity, 88
 - upper bound, algorithm, 88
- loop invariants, 97–101
 - code from loop invariants, 28
 - defining, 97
 - exit condition, 99
 - greedy algorithms, 297
 - initializing the system, 101
 - iterative algorithms, 28
 - maintaining the loop invariant, 98
 - maintaining the system invariants, 100
 - obtaining the postcondition, 99, 101
 - operation specification, 101
 - private invariants, 100
 - public invariants, 100
 - relationship to system invariants, 100
 - running time, 99
 - specification, 97
 - step, 97
- loop invariants for iterative algorithms, 9, 14, 16
- lower bounds, *see also* loop invariant for lower bounds, 83, 88, 94, 334, 402, 565, 577
 - for GCD, 83
 - loop invariant for lower bounds, 88
 - lower bounds proof, state of art, 94
 - reductions, 402
- machine learning, 431–438
 - abstract thinking, 432
 - algorithm, 434
 - blind, 433
 - cat, 432
 - coding, 431
 - compression, 434
 - convolution and recurrent layer, 438
 - correlation of vectors, 436
 - error, 433
 - error surface, 433
 - evolution, 431
 - generalizing, 434
 - gradient descent, 433
 - hopeful applications, 431
 - linear and non-linear regression, 432, 433
 - linear layer = matrix multiplication, 436
 - machine, 432
 - neural networks, 433
 - non-linear layer, 437
 - overfitting, 434
 - regularization, 435
 - scary applications, 431
 - singularity, 438
 - smooth/differentiable, 434
 - supervised training data, 432
 - theory: Learnable Probably Approximately Correct (PAC), 435
 - underfitting, 434

- vectors, 436
- magic sevens, 66
 - basic steps, 67
 - establishing loop invariant, 67
 - exit condition, 67
 - loop invariant, 67
 - maintain loop invariant, 67
 - running time, 67
 - specifications, 66
- matrix multiplication
 - all pairs, 397
 - chains of, solution as tree, 390
 - Strassen's matrix multiplication example, 161
- measures of progress, loop invariants, 5
- merge sort example, 149
- merging with queue, 61
- meta-algorithms, xiii, 2
- min cut specification, 269
- Monte Carlo model, 424
- more of input, 22
- more-of-the-output loop invariant algorithms, 22
 - Euler cycle example, 44
 - selection sort example, 23
- more-of-the-input iterative loop invariant
 - algorithms, 22, 34, 70, 136, 296, 368, 562
 - coloring the plane example, 33
 - deterministic finite automation, 35
 - exercise solutions, 562
 - in dynamic programming
 - longest increasing contiguous subsequence example, 369
 - longest increasing subsequence example, 370
 - in greedy algorithms, 296
 - in VLSI chip testing example, 68
 - more-of-the-input vs. more-of-the-output, 42
 - tournament example, 42
 - recursive algorithms, link to, 137
- multiplexer example, 92
- narrowing the search space, 22, 24, 64, 89, 137, 297, 565
 - binary search example, 24
 - binary search trees, 64
 - exercise solutions, 565
 - magic sevens, 66
 - VLSI chip testing, 68
- network flows/linear programming, 268
 - bipartite matching using network flow, 419
 - exercise solutions, 573
 - hill-climbing algorithm, small local maximum, 270
 - linear programming, 288
 - min cut specification, 269
 - primal-dual hill-climbing method, 276
 - specification, 268
 - steepest-ascent hill-climbing algorithm, 284
- nondeterministic polynomial-time decision
 - problems (NP) completeness, 407
 - bipartite matching, network flow algorithm, 402, 419, 420
 - classifying problems, 403
 - exercise solutions, 582
 - lower bounds, 402
 - NP completeness proof steps, 407, 409
 - nondeterministic polynomial-time decision problems (NP), 407
 - reduction P_{1poly} P_2 , 402
 - reverse reductions, 403
 - satisfiability vs. optimization
 - Alg for optimization problem, 406
 - CIR-SAT, 404
 - optimization problems, 406
 - 3 coloring example, 403
 - upper bounds, 402
 - why reduce, 402
- operations on integers, 157
 - b^N example, 157
 - Strassen's matrix multiplication example, 161
 - xyz example, 159
- optimization problems, 241
 - examples, 241
 - airplane, 242
 - course scheduling, 242
 - longest common sequence, 241
 - network flow, 268
 - problem specification, 241
- parity example, 92, 93
- parsing with stack, *see also* context-free parsing, 62
 - code, 62
 - ending, 62
 - example, 62
 - initial conditions, 62
 - loop invariant, 62
 - maintaining loop invariant, 62
 - parsing only, 63
 - parsing with context-free grammar, 63
 - specifications, 62
- partial order linear ordering, 264
 - depth-first search algorithm, 265
 - easy but slow algorithm, 265
 - partial order definition, 265
 - shortest weight path, DAG, 267
 - topological sort problem specifications, 265
 - total order definition, 264
- postconditions, 1
- preconditions, 1
- primal-dual hill-climbing method, 276
 - algorithm, 276
 - ending, 278
 - max-flow-min-cut duality principle, 282
 - running time, 283

- primality testing, randomized, 425
- printing neatly example, 346
- private invariants, 100
- pseudocode
 - graph of life, 382
 - iterative algorithms, 18
- public invariants, 100
- quadratic function, 523
- quantifiers, *See* existential/universal quantifiers
- quick sort
 - example, 150
 - randomized, 425
- radix sort, 78
 - basic steps, 78
 - ending, 79
 - establishing loop invariant, 79
 - loop invariant, 79
 - maintain loop invariant, 79
 - specification, 78
- randomized algorithms, 423
 - deterministic worst case model, 424, 425
 - game show problem, 425
 - hiding worst cases from adversary, 423
 - Las Vegas model, 424
 - Monte Carlo model, 424
 - optimization problems with random structure, 427
 - expander graphs, 428
 - max cut problem, 428
 - quick sort, 425
 - randomized counting, 426
 - randomized primality testing, 425
 - VLSI chip testing, 72
- randomly generating maze, 195
 - initial conditions, 195
 - postcondition, 195
 - precondition, 195
 - running time, 196
 - searching maze, 197
 - subinstances, 196
- recurrence relations, 540
 - exercise solutions, 586
 - in recursive backtracking algorithms, 328
 - proofs, 543
 - recursive programs timing, 540
 - solving recursive relations, 541
- recursion, 133, 208–210
 - binary trees, 209
 - Fast Fourier Transformations (FFT), 164
 - formal proof, 209
 - friends level of abstraction, 135
 - on trees
 - exercise solutions, 570
 - generalizing problem solved, 179
 - heap sort/priority queues, 180
 - recursive definition of tree, 169
 - representing expressions with trees, 187
 - simple examples, 177
 - running time, 209
 - steps, 208
- recursion on trees
 - data structure, 171
 - many children, 172
 - running time, 171
- recursive algorithms, *See also* recursive backtracking
 - checklist for, 139
 - code structure, 139
 - specifications, 140
 - tasks to complete, 142
 - variables, 140
- correctness proof, with strong induction, 146
- examples
 - Ackermann's function, 162
 - exercise solutions, 567
 - operations on integers, 157
 - b^N example, 157
 - Strassen's matrix multiplication example, 161
 - xyz example, 159
- exercise solutions, 567
 - base cases, 137
 - general input, 135
 - generalizing the problem, 136
 - link to techniques, iterative algorithms, 137
 - minimizing number of cases, 136
 - running time, 137
 - size, 135
 - specifications, 135
- looking forward vs. backward, 134
- solving, 162
- sorting/selecting algorithms, 149
 - choosing the pivot, 152
 - finding the k th smallest example, 152
 - general recursive sorting algorithm, 149
 - merge sort example, 149
 - partitioning according to pivot element, 155
 - quick sort example, 150
- stack frame, 144
- strong induction, 146
- tower of Hanoi, 138
- recursive backtracking algorithms, 321, 576
 - as sequence of decisions, 321
 - best animal searching example, 323
 - maze searching example, 322
 - developing steps, 325
 - exercise solutions, 576
 - pruning branches, 329
 - greedy algorithms, 330
 - queens problem example, 325
 - satisfiability, 331
 - code, 333
 - Davis-Putnam, 332

- instances/subinstances, 332
 - pruning, 332
 - running time, 334
 - satisfiability problem, 331
 - solutions iterating, 331
- recursive backtracking, dynamic programming, 445
- recursive depth-first search, 263
 - achieving postcondition, 264
 - code, 263
 - example, 264
 - running time, 264
- recursive image specification, 192
- recursive images, 192
 - exercise solutions, 571
 - fixed recursive/base case image, 192
 - base case, 192
 - birthday cake, 193
 - examples, 193
 - fractal, 194
 - image drawing, 192
 - man recursively framed, 193
 - recurring, 192
 - rotating square, 193
 - randomly generating maze, 195
- reduction, 402
 - dynamic programming algorithms via, 375
 - best path similarity, 375
 - bigger-is-smarter elephant problem, 379
 - event scheduling problem, 375
 - graph instance formation, 376
 - mapping back algorithm, 377
 - lower bound, 402
 - optimization problems reduction, 406
 - reverse reduction, 403
 - upper bound, 402
 - use in classifying problems, 403
- running time, *See* time/space complexity, 2
 - Ackermann's function, 163
 - best binary search tree, 396
 - binary search, 24
 - binary search trees, 65
 - coloring the plane, 34
 - context-free grammar parsing, 204
 - counting sort, 77
 - dynamic programming, 346
 - GCD algorithm, 82
 - generic search algorithm, 247
 - heapify, 182
 - heapsort, 186
 - interval cover, 314
 - iterative algorithms, 10, 20
 - job/event scheduling, 310
 - linear programming, 292
 - magic sevens, 67
 - makeheap, 183
 - merge sort, 150
 - minimum spanning tree, 316
 - nodes in binary tree, 171
 - primal-dual hill-climbing method, 283
 - queens problem, 329
 - quick sort, 151
 - randomly generating maze, 196
 - recursive algorithms, 137
 - recursive backtracking, 340
 - satisfiability, 334
 - steepest-ascent hill-climbing algorithm, 285
 - towers of Hanoi, 139
 - union find set system, 54
 - VLSI chip testing, 72
- sequence of actions vs. a sequence of assertions
 - check each computation path, 6
 - code vs. math assertions, 7
 - correctness, 9
 - differentiating between values, 7
 - game of life, 9
 - loops, 8
 - one step at a time, 7
 - proof by case, 8
 - proof by transitivity, 8
 - proof of correctness of each step, 8
 - proof of correctness of the algorithm, 6
- shortest weight path, directed leveled graph
 - example, 337
- solution is a tree, all pairs using matrix
 - multiplication, 397
- sorting and selection algorithms, 149
 - bubble sort example, 26
 - finding the k th smallest example, 152
 - choosing the pivot, 152
 - iterative sorting algorithms
 - bucket sort by hand, 74
 - counting sort, 75
 - radix sort, 78
 - merge sort, 149
 - quick sort, 150
 - randomized quick sort, 425
 - recursive sorting algorithm, general, 149
- stack frame, 144
 - memory, 145
 - stack of stack frames, 145
 - tree of stack frames, 133, 144, 171, 204
 - using, 146
- steepest-ascent hill-climbing algorithm, 284
 - augmentation path, 284
 - running time, 285
- Strassen's matrix multiplication example, 161
- system invariants, 99–101
 - goal, 99
 - private invariants, 100
 - public invariants, 100
 - relationship to loop invariants, 99

- time/space complexity, 83, 84, 403, 425, 513, 518, 548, 584
 - examples, 510
 - exercise solutions, 584
 - formal definition, 513
 - as functions, 508
 - operation definition, 510
 - purpose, 508
 - size definition, 509
 - time complexity of problem, 510
- tournament example, 42
- towers of Hanoi, 138
 - code, 139
 - divide and conquer, 138
 - running time, 139
 - specification, 138, 139
 - subinstance, 139
- tree edges, 261
- trees, *See also* binary search trees; recursion on trees, 52, *see also* recursion on trees
 - AVL tree, 53
 - best AVL tree problem, 396
 - best binary search tree, 395
 - binary search tree, 53
 - chains of matrix multiplications, solution as tree example, 390
 - dictionary ADT, data structure, 52–53
 - is a tree a binary search tree example, 174
 - nodes in binary tree example, 170
 - recursive definition of, 169
 - representing expressions with
 - differentiate expression example, 189
 - evaluate expression example, 188
 - recursive definition of expression, 188
 - simplify expression example, 189
 - tree data structure, 188
 - of stack frames, 133, 144, 171, 204, 294
 - traversals, 174
- universal quantifiers, *See* existential/universal quantifiers
- upper bound, 88
 - cuts as upper bound, 279
 - for hill-climbing, 292
- VLSI chip testing, 68
 - brute force algorithm, 69
 - data structure, 69
 - exiting loop, 72
 - extending the algorithm, 72
 - faster algorithm, 70
 - initial code, 72
 - loop invariant design, 70
 - maintaining loop invariant, 71
 - measure of progress, 72
 - randomized algorithm, 72
 - running time, 72
 - specification, 69
- weighted job/event scheduling problem, 371
 - failed algorithms, 371
 - greedy dynamic programming, 372
 - weighted event scheduling problem, 371
- work done-bubble sort example, 26

