Ryuhei Uehara

First Course in Algorithms Through Puzzles

Springer

First Course in Algorithms Through Puzzles

Ryuhei Uehara

First Course in Algorithms Through Puzzles

Springer

Ryuhei Uehara
Japan Advanced Institute of Science
and Technology
Nomi, Ishikawa, Japan

ISBN 978-981-13-3187-9 ISBN 978-981-13-3188-6 (eBook)
https://doi.org/10.1007/978-981-13-3188-6

Library of Congress Control Number: 2018960741

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Preface

This book is an introduction to *algorithms*. What is an **algorithm**? In short, "algorithm" means "a way of solving a problem." When people encounter a problem, the approach to solving it depends on who is solving it, and then the efficiency varies accordingly. Lately, mobile devices have become quite smarter, planning your route when you provide your destination, and running an application when you talk to it. Inside this tiny device, there exists a computer that is smarter than the old large-scale computers, and it runs some neat algorithms to solve your problems. Smart algorithms use clever tricks to reduce computational time and the amount of memory needed.

One may think that "from the viewpoint of the progress of hardware devices, is an improvement such as this insignificant?" However, you would be wrong. For example, take integer programming. Without going into details, it is a general framework to solve mathematical optimization programs, and many realistic problems can be solved using this model. The running time of programs for solving integer programming has been improved 10,000,000,000 times over the last two decades.[1] Surprisingly, the contribution of hardware is 2000 times, and the contribution of software, that is, the algorithm, is 4,75,000 times. It is not easy to see, but "the improvement of the way of solving" has been much more effective than "the development of hardware."

In this book, I introduce and explain the basic algorithms and their analytical methods for undergraduate students in the Faculty of Information Science. This book starts with the basic models, and no prerequisite knowledge is required. All algorithms and methods in this book are well known and frequently used in real computing. This book aims to be self-contained; thus, it is not only a textbook, but also allows you to learn by yourself, or use as a reference book for beginners. On the other hand, I provide some smart tips for non-beginner readers.

[1] R. Bixby, Z. Gu, and Ed. Rothberg, "Presolve for Linear and Mixed-Integer Programming," RAMP Symposium, 2012, Japan.

Exercise: Exercises appear in the text and are not collected at the end of sections. If you find an exercise when you read this book, I would like to ask you to take a break, tackle the exercise, and check the answer and comments. For every exercise, an answer and comments are given in this book.

Each exercise is marked with ☕ marks, and those with many cups are more difficult than those with fewer ones. They say that "a mathematician is a machine for turning coffee into theorems" (this quotation is often attributed to Paul Erdős;[2] however, he ascribed it to Alfréd Rényi). In this regard, I hope you will enjoy them with cups of coffee.

Exercises are provided in various ways in your textbooks and reference books. In this book, all solutions for exercises are given. This is rather rare. Actually, this is not that easy from the authors' viewpoints. Some textbooks seem to be missing the answers because of the authors' laziness. Even if the author considers a problem "trivial," it is not necessarily the case for readers, and beginners tend to fail to follow in such "small" steps. Moreover, although it seems trivial at a glance, difficulties may start to appear when we try to solve some problems. For both readers and authors, it is tragic that beginners fail to learn because of authors' prejudgments.

Even if it seems easy to solve a problem, solving it by yourself enables you to learn many more things than you expect. I am hoping you will try to tackle the exercises by yourselves. If you have no time to solve them, I strongly recommend to check their solutions carefully before proceeding to the next topic.

Analyses and Proofs: In this book, I provide details of the proofs and the analyses of algorithms. Some unfamiliar readers could find some of the mathematical descriptions difficult. However, they are not beyond the range of high school mathematics although some readers may hate "mathematical induction," "exponential functions," or "logarithmic functions." Although I believe that ordinary high school students should be able to follow the main ideas of algorithms and the pleasure they bring, you may not lie down on your couch to follow the analyses and proofs. However, I want to emphasize that there exists beauty you cannot taste without such understanding. The reason why I do not omit these "tough" parts is that I would like to invite you to appreciate this deep beauty beyond the obstacles you will encounter. Each equation is described in detail, and it is not as hard once you indeed try to follow.

[2] **Paul Erdős(1913–1996)**:
He was one of the most productive mathematicians of the twentieth century, and wrote more than 1400 journal papers with more than 500 collaborators. Erdő numbers are a part of the folklore of mathematicians that indicate the distance between themselves and Erdős. Erdős himself has Erdős number 0, and his coauthors have Erdős number 1. For example, I have Erdős number 2 because I have collaborated with a mathematician who has Erdős number 1.

Algorithms are fun. Useful algorithms provide a pleasant feeling much the same as well-designed puzzles. In this book, I have included some famous real puzzles to describe the algorithms. They are quite suitable for explaining the basic techniques of algorithms, which also show us how to solve these puzzles. Through learning algorithms, I hope you will enjoy acquiring knowledge in such a pleasant way.

Nomi, Japan
October 2018

Ryuhei Uehara

Contents

Chapter 1
Preliminaries

Abstract In this chapter, we learn about (1) machine models, (2) how to measure the efficiency of algorithms, (3) data structures, (4) notations for the analysis of algorithms, (5) functions, and (6) graph theory. All of these are very basic, and we cannot omit any of them when learning about designing/analyzing algorithms. However, “basic” does not necessarily mean “easy” or “trivial.” If you are unable to understand the principle when you read the text for the first time, you do not need to be overly worried. You can revise it again when you learn concrete algorithms later. Some notions may require some concrete examples to ensure proper understanding. The contents of this section are basic tools for understanding algorithms. Although we need to know which tools we have available, we do not have to grasp all the detailed functions of these tools at first. It is not too late to only grasp the principles and usefulness of the tools when you use them.

What you will learn:

- Machine models (Turing machine model, RAM model, other models)
- Efficiency of algorithms
- Data structures (array, queue, stack)
- Big-O notation
- Polynomial, exponential, logarithmic, harmonic functions
- Basic graph theory

1.1 Machine Models

An *Algorithm* is a method for solving a given problem. Representing how to solve a problem requires you to define *basic operations* precisely. That is, an algorithm is a sequence of basic operations of some machine model you assume. First, we

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_1

introduce representative machine models: The Turing machine model, RAM model, and the other models. When you adopt a simple machine model, it is easy to consider the computational power of the model. However, because it has poor or restricted operations, it is hard to design/develop some algorithms. On the other hand, if you use a more abstract machine model, even though it is easier to describe an algorithm, you may face a gap between the algorithm and real computers when you implement it. In order to make a sense of balance, it is a good idea to review some different machine models. We have to mind the machine model when we estimate and/or evaluate the efficiency of an algorithm. Sometimes the efficiency differs depending on the machine model that is assumed.

It is worth mentioning that, currently, every "computer" is a "digital computer," where all data in it is manipulated in a digital way. That is, all data are represented by a sequence consisting of 0 and 1. More precisely, all integers, real numbers, and even irrational numbers are described by a (finite) sequence of 0 and 1 in some way. Each single unit of 0 or 1 is called a **bit**. For example, a natural number is represented by a binary system. Namely, the data 0, 1, 2, 3, 4, 5, 6, . . . are represented in the binary system as 0, 1, 10, 11, 100, 101, 110, . . .[1]

Exercise 1 ☕ Describe the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 in the binary and the hexadecimal systems. In the hexadecimal system, 10, 11, 12, 13, 14, and 15 are described by A, B, C, D, E, and F, respectively. Can you find a relationship between these two systems?

Are there non-digital computers?

An ordinary computer is a digital computer based on the binary system. In a real digital circuit, "1" is distinguished from "0" by voltage. That is, a point represents "1" if it has a voltage of, say, 5 V, and "0" if it has a ground level voltage, or 0 V. This is not inevitable; historically, there were analog computers based on analog circuits, and trinary computers based on the trinary system of minus, 0, and plus. However, in my age, all computers were already digital. Digital circuits are simple and refrain from making a noise. Moreover, "1" and "0" can be represented by other simple mechanisms, including a paper strip (with and without holes), a magnetic disk (N and S poles), and an optical cable (with and without light). As a result, non-digital computers have disappeared.

[1] **Natural number**:
Our *natural numbers* start from 0, not 1, in the area of information science. "Nonnegative integers" start from 1.

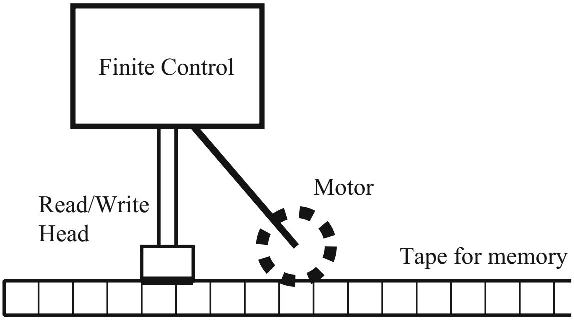


Fig. 1.1 Turing machine model

1.1.1 Turing Machine Model

A **Turing machine** is the machine model introduced by Alan Turing in 1936.[2] The structure is very simple because it is a virtual machine defined to discuss the mechanism of computation mathematically. It consists of four parts as shown in Fig. 1.1; **tape**, **finite control**, **head**, and **motor**.

The tape is one sided, that is, it has a left end, but has no right end. One of the digits 0, 1, or a blank is written in each cell of the tape. The finite control is in the "initial state," and the head indicates the leftmost cell in the tape. When the machine is turned on, it moves as follows:

(1) Read the letter c on the tape;
(2) Following the predetermined rule [movement on the state x with letter c],

- rewrite the letter in the head (or leave it);
- move the head to the right/left by one step (or stay there);
- change the state from x to y;

(3) If the new state is "halt", it halts; otherwise, go to step 1.

The mechanism is so simple that it seems to be too weak. However, in theory, every existing computer can be abstracted by this simple model. More precisely, the typical computer that runs programs from memory is known as a *von Neumann-type computer*, and it is known that the computational power of any von Neumann-type computer is the same as that of a Turing machine. (As a real computer has limited memory, indeed, we can say that it is weaker than a Turing machine!) This fact implies that any problem solvable by a real computer is also solvable by a Turing machine (if you do not mind their computation time).[3]

[2] **Alan Mathison Turing (1912–1954)**:
Turing was a British mathematician and computer scientist. The 1930s were the times before real computers, and he was the genius who developed the notion of a "computation model" that holds good even in our day. He was involved with the birth of computer, took an important part in breaking German codes during World War II, and died when he was only 42 years old from poisoning by cyanic acid. He was a dramatic celebrity in various ways.

[3] **Theory of computation**:
The **theory of computation** is a science that investigates "computation" itself. When you define a

Here we introduce an interesting fact. It is known that there exists a problem that cannot be solved by any computer.

Theorem 1 *There exists no algorithm that can determine whether any given Turing machine with its input will halt or not. That is, this problem, known as* the halting problem*, is theoretically unsolvable.*

This is one of the basic theorems in the theory of computation, and it is related to deep but interesting topics including diagonalization and Gödel's Incompleteness Theorem. However, we mention this as an interesting fact only because these topics are not the target of this book (see the references). As a corollary, you cannot design a universal debugger that determines whether any program will halt or not. Note that we consider the general case, i.e., "*any* given program." In other words, we may be able to design an *almost* universal debugger that solves the halting problem for many cases, but not all cases.

1.1.2 RAM Model

The Turing machine is a theoretical and virtual machine model, and it is useful when we discuss the theoretical limit of computational power. However, it is too restricted or too weak to consider algorithms on it. For example, it does not have basic mathematical operations such as addition and subtraction. Therefore, we usually use a **RAM (Random Access Machine) model** when we discuss practical algorithms. A RAM model consists of **finite control**, which corresponds to the **CPU (Central Processing Unit)**, and **memory** (see Fig. 1.2). Here we describe the details of these two elements.

Memory. Real computers keep their data in the cache in the CPU, memory, and external memory such as a hard disk drive or DVD. Such storage is modeled by

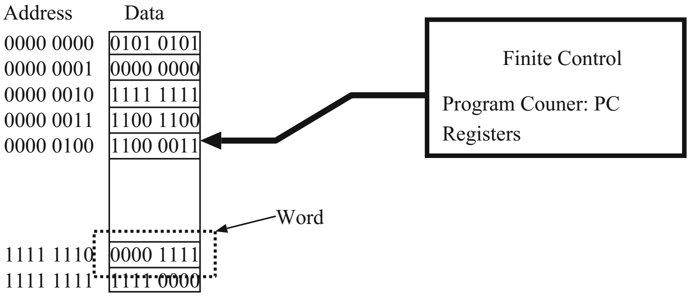


Fig. 1.2 RAM model

set of basic operations, the set of "computable" functions are determined by the operations. Then the set of "uncomputable" functions is well defined. Diagonalization is a basic proof tool in this area, and it is strongly related to Gödel's Incompleteness Theorem.

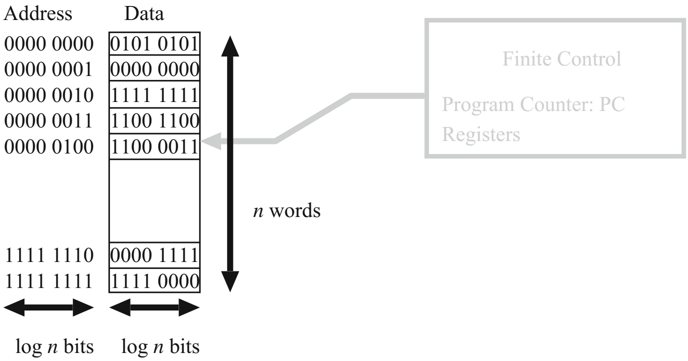


Fig. 1.3 Relationships among memory size, address, and word size of the RAM model. Let us assume that the memory can store n data. In this case, to indicate all the data, $\log n$ bits are required. Therefore, it is convenient when a word consists of $\log n$ bits. That is, a "k bit CPU" usually uses k bits as one word, which means that its memory can store 2^k words

memory in the RAM model. Each memory unit stores a **word**. Data exchange between finite control and memory occurs as one word per unit time. Finite control uses an **address** of a word to specify it. For example, finite control reads a word from the memory of address 0, and writes it into the memory of address 100.

Usually, in a CPU, the number of bits in a word is the same as the number of bits of the address in the memory (Fig. 1.3). This is because each address itself can be dealt with as data. We sometimes say that your PC contains a 64-bit CPU. This number indicates the size of a word and the size of an address. For example, in a 64-bit CPU, one word consists of 64 bits, and one address can be represented from $\underbrace{00\ldots0}_{64}$ to $\underbrace{11\ldots1}_{64}$, which indicates one of $2^{64} \approx 1.8 \times 10^{19}$ different data. That is, when we refer to a RAM model with a 64-bit CPU, it refers to a word with 64 bits on each clock, and its memory can store 2^{64} words in total.

Finite control. The finite control unit reads the word in a memory, applies an operation to the word, and writes the resulting word to the memory again (at the same or different address at which the original word was stored). A computation is the repetition of this sequence. We focus on more details relating to real CPUs. A CPU has some special memory units that consist of one **program counter** (PC), and several **registers**. When a computer is turned on, the contents of the PC and registers are initialized by 0. Then the CPU repeats the following operations:

(1) It reads the content X at the address PC to a register.
(2) It applies an operation Y according to the value of X.
(3) It increments the value of PC by 1.

The basic principle of any computer is described by these three steps. That is, the CPU reads the memory, applies a basic operation to the data, and proceeds to the next memory step by step. Typically, X is a word consisting of 64 bits. Therefore, we have 2^{64} different patterns. For each of these many patterns, some operation Y corresponds to X, which is designed on the CPU. If the CPU is available on the market, the operations are standardized and published by the manufacturer. (Incidentally, some geeks may remember coding rules in the period of 8-bit CPUs, which only has $2^8 = 256$ different patterns!). Usually, the basic operations on the RAM model can be classified into three groups as follows.

Substitution: For a given address, there is data at the address. This operation gives and takes between the data and a register. For example, when data at address A is written into a memory at address B, the CPU first reads the data at address A into a register, and next writes the data in the register into the memory at address B. Similar substitution operations are applied when the CPU has to clear some data at address A by replacing it with zero.

Calculation: This operation performs a calculation between two data values and puts the result into a register. The two data values come from a register or memory cell. Typically, the CPU reads data from a memory cell, applies some calculation (e.g., addition) to the data and some register, and writes the resulting data into another register. Although possible calculations differ according to the CPU, some simple mathematical operations can be performed.

Comparison and branch: This operation overwrites the PC if a register satisfies some condition. Typically, it overwrites the PC if the value of a specified register is zero. If the condition is not satisfied, the value of PC is not changed, and hence, the next operation will be performed in the next step. On the other hand, if the condition is satisfied, the value of the PC is changed. In this case, the CPU changes the "next address" to be read, that is, the CPU can jump to an address "far" from the current address. This mechanism allows us to use "random access memory."

In the early years, CPU allowed very limited operations. Interestingly, this was sufficient in a sense. That is, theoretically, the following theorem is known.

Theorem 2 *We can compute any function if we have the following set of operations: (1) increase the value of a register by one, (2) decrease the value of a register by one, and (3) compare the value of a register with zero, and branch if it is zero.*

Theorem 2 says that we can compute any function if we can design some reasonable algorithm for the function. In other words, the power of a computer essentially does not change regardless of the computational mechanism you use. Nevertheless, the computational model is important. If you use the machine model with the set of operations in Theorem 2, creating a program is quite difficult. You may need to write a program to perform the addition of two integers at the first step. Thus, many advanced operations are prepared as a basic set of operations in recent high-performance CPUs. Although, it seems that there is a big gap between the set of operations of the RAM model and real programming languages such as C, Java, Python, and so on. However, any computer program described by some programming language can be realized on the RAM model by combining the set of simple operations we already have. For example, we can use the notion of a "**variable**" in most programming languages (see Sect. 1.3.1 for the details). It can be realized as follows on the RAM model:

Example 1 The system assigns that "variable A is located at an address 1111 1000" when the program starts. Then, substitution to variable A is equivalent to substitution to the address 1111 1000. If the program has to read the value in variable A, it accesses the memory with the address 1111 1000.

Using this "variable", we can realize the usual substitution in a program. A typical example is as follows.

Example 2 In many programs, "add 1 to some variable A" frequently appears. In this book, we will denote it by

$$A \leftarrow A + 1;$$

When a program based on a RAM model runs this statement, it is processed as follows (Fig. 1.4): We suppose that the variable A is written at the address 1111 1000. (Of course, this is an example, which does not correspond to a concrete CPU on the market; rather, this is an example of so-called **assembly language** or **machine language**.)

(1) When it is turned on, the CPU reads the contents at the address PC (=0) in the memory. Then it finds the data "1010 1010" (Fig. 1.4(1)). Following the manual of the CPU, the word "1010 1010" means the sequence of three operations:

    (a) Read the word W at the next address (PC + 1) and read the data at address W to the register 1.
    (b) Read the word W' at the address following the next address (PC + 2) and add it to register 1.
    (c) Write the contents of register 1 to the address following the address after the next address (PC + 3).

    (Here the value of PC is incremented by 1.)

(2) The CPU reads the address PC (=1) and obtains the data (=1111 1000). Then it checks the address 1111 1000 and finds the data (=0000 1010). Thus, it copies to register 1 (Fig. 1.4(2)). (Here the value of PC is incremented by 1.)
(3) The CPU reads the address PC (=10) and obtains the data (=0000 0001). It adds the data 0000 0001 to register 1, then the contents of register 1 are changed to 0000 1011 (Fig. 1.4(3)). (Here the value of PC is incremented by 1.)
(4) The CPU reads the address PC (=11) and obtains the data (=1111 1000) (Fig. 1.4(4)). Thus, it writes the value of the register 1 (=0000 1011) to the memory at address 1111 1000 (Fig. 1.4(5)). (Here the value of PC is incremented by 1.)

The CPU performs repetitions in a similar way. That is, it increments PC by 1, by reading the data at address PC, and so on.

We note that the PC always indicates "the next address the CPU will read." Thus, when a program updates the value of PC itself, the program will "jump" to the other address. By this function, we can realize comparison and branching. That is, we can change the behavior of the program.

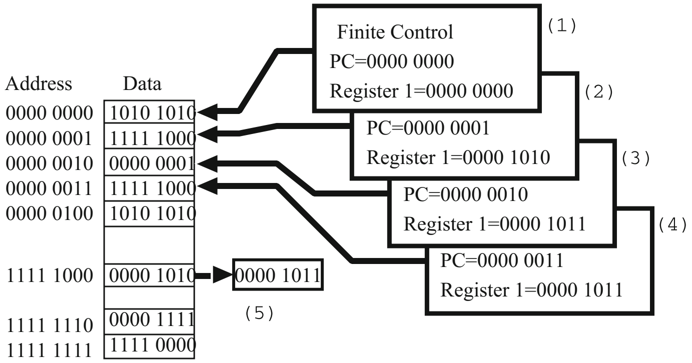


Fig. 1.4 The sequence of statements performed on the RAM model: (1) When it is turned on, (2) it reads the contents at address 1111 1000 to register 1, (3) it adds 0000 0001 to the contents of register 1, (4) it writes the contents in register 1 to the memory at address 1111 1000, and (5) The contents 0000 1010 at address 1111 1000 are replaced by 0000 1011

As already mentioned, each statement of a standard CPU is quite simple and limited. However, a combination of many simple statements can perform an algorithm described by a high-level programming languages such as C, Java, Haskell, and Prolog. An algorithm written in a high-level programming language is translated to machine language, which is a sequence of simple statements defined on the CPU, by a **compiler** of the high-level programming language.[4] On your computer, you sometimes have an "executable file" with a "source file"; the simple statements are written in the executable file, which is readable by your computer, whereas human readable statements are written in the source file (in the high-level programming language). In this book, each algorithm is written in a virtual high-level procedural programming language similar to C and Java (in a more natural language style). However, these algorithms can eventually be performed on a simple RAM model machine.

Real computers are...

In this book, we simplify the model to allow for easy understanding. In some real computers, the size of a word may differ from the size of an address. Moreover, we ignore the details of memory. For example, semiconductor memory and DVD are "memory," although we usually distinguish them according to their price, speed, size, and other properties.

[4] **Compiler**:
We usually "compile" a computer program, written in a text file, into an executable binary file on a computer. The compiler program reads the computer program in a text file, and exchanges it into the binary file that is written in a sequence of operations readable by the computer system. That is, when we write a program, the program itself is usually different from the real program that runs on your system.

1.1.3 Other Machine Models

As seen already, computers use digital data represented by binary numbers. However, users cannot recognize this at a glance. For example, you can use irrational numbers such as $\sqrt{2} = 1.4142\ldots$, $\pi = 3.141592\ldots$, and trigonometric functions such as sin and cos. You can also compute any function such as e^{π}, even if you do not know how to compute it. Moreover, your computer can process voice and movie data, which is not numerical data. There exists a big gap between such data and the binary system, and it is not clear as to how to fill this gap. In real computers, representing such complicated data in the simple binary system is very problematic, as is operating and computing between these data in the binary system. It is a challenging problem how to deal with the "computational error" resulting from this representation. However, such a **numerical computation** is beyond the scope of this book.

In theoretical computer science, there is a research area known as **computational geometry**, in which researchers consider how to solve geometric problems on a computer. In this active area, when they consider some algorithms, the representations of real numbers and the steps required for computing trigonometry are not their main issues. Therefore, such "trivial" issues are considered in a black box. That is, when they consider the complexity of their algorithms, they assume the following implicitly:

- Each data element has infinite accuracy, and it can be written and read in one step in one memory cell.
- Each "basic mathematical function" can be computed in one step with infinite accuracy.

In other words, they use the abstract machine model on one level higher than the RAM model and Turing machine model. When you implement the algorithms on this machine model onto a real computer, you have to handle computation errors because any real computer has limited memories. Because the set of basic mathematical functions covers different functions depending on the context, you sometimes have to provide for the implementation of the operations and take care of their execution time.

Lately, new computation frameworks have been introduced and investigated. For example, **quantum computer** is a computation model based on quantum mechanics, and **DNA computer** is a computation model based on the chemical reactions of DNA (or string of amino acids). In these models, a computer consists of different elements and a mechanism to solve some specific problems more efficiently than ordinary computers.

In this book, we deal with basic problems on ordinary digital computers. We adopt the standard RAM model, with each data element being an integer represented in the binary system. Usually, each data element is a natural number, and sometimes it is a negative number, and we do not deal with real numbers and irrational numbers. In other words, we do *not* consider (1) how we can represent data, (2) how we can compute a basic mathematical operation, and (3) computation errors that occur during implementation. When we describe an algorithm, we do not suppose any specific computer language, and use free format to make it easy to understand the algorithm itself.

1.2 Efficiency of Algorithm

Even for the same problem, efficiency differs considerably depending on the algorithms. Of course, efficiency depends on the programming language, coding method, and computer system in general. Moreover, the programmer's skill also has influence. However, as mentioned in the introduction, the choice of an algorithm is quite influential. Especially, the more the amount of data that is given, the more the difference becomes apparent. Now, what is "the efficiency of a computation"? The answer is the resources required to perform the computation. More precisely, time and space required to compute. These terms are known as **time complexity** and **space complexity** in the area of computational complexity. In general, time complexity tends to be considered. Time complexity is the number of steps required for a computation, but this depends on the computation model. In this section, we first observe how to evaluate the complexity of each machine model and discuss the general framework.

1.2.1 Evaluation of Algorithms on Turing Machine Model

It is simple to evaluate efficiency on the Turing machine model. As seen in Sect. 1.1.1, the Turing machine is simple and repeats basic operations. We assume that each basic operation (reading/writing a letter on the tape, moving the head, and changing its state) takes one unit of time. Therefore, for a given input, its time complexity is defined by the number of times the loop is repeated, and its space complexity if defined by the number of checked cells on the tape.

1.2.2 Evaluation of Algorithms on RAM Model

In the RAM model, its time complexity is defined by the number of times the program counter PC is updated. Here we need to mention two points.

Time to access memory. In the RAM model, one data element in any memory cell can be read/written in one unit of time. This is the reason why this machine model is known as "Random Access." In a Turing machine, if it is necessary to read a data element located far from the current head position, it takes time because the machine has to move the head one by one. On the other hand, this is not the case for the RAM model machine. When the address of the data is given, the machine can read or write to the address in one step.

One unit of memory. The RAM model deals with the data in one word in one step. If it has n words in memory, to access each of them requires $\log n$ bits to access it uniquely. (If you are not familiar with the function $\log n$, see Sect. 1.5 for the details.) It is reasonable and convenient to assume that one word uses $\log n$ bits in the RAM model, and in fact, real CPUs almost follow this rule. For example, if the CPU in your laptop is said to be a "64-bit CPU", one word consists of 64 bits, and the machine can contain 2^{64} words in its memory; thus, we adopt this definition. That is, in our RAM model, "one word" means "$\log n$ bits," where n is the number of memory cells. Our RAM model can read or write one word in one step. On the Turing machine, one bit is a unit. Therefore, some algorithms seem to run $\log n$ times faster on the RAM model than on the Turing machine model.

Exercise 2 ☕ Assume that our RAM model has n words in its memory. Then how many bits are in this memory? In that case, how many different operations can the CPU have? Calculate the number of bits in each RAM model for $n = 65536$, $n = 2^{64}$, and $n = 2^k$.

1.2.3 Evaluation of Algorithms on Other Models

When some machine model is more abstract than the Turing machine model or RAM model is adopted, you can read/write a real number in a unit time with one unit memory cell, and use mathematical operations such as power and trigonometry, for which it is not necessarily clear how to compute them efficiently. In such a model, we measure the efficiency of an algorithm by the number of "basic operations." The set of basic operations can be changed in the context; however, they usually involve reading/writing data, operations on one or two data values, and conditional branches.

When you consider some algorithms on such an abstract model, it is not a good idea to be unconcerned about what you eliminate from your model to simplify it. For example, by using the idea that you can pack data of any length into a word, you may use a word to represent complex matters to apparently build a "fast" algorithm. However, the algorithm may not run efficiently on a real computer when you implement it. Therefore, your choice of the machine model should be *reasonable* unless it warrants some special case; for example, when you are investigating the properties of the model itself.

1.2.4 Computational Complexity in the Worst Case

An algorithm computes some output from its input in general. The resources required for the computation are known as **computational complexity**. Usually, we consider two computational complexities; **time complexity** is the number of steps required for the computation, and **space complexity** is the number of words (or bits in some case) required for the computation. (To be precise, it is safe to measure the space complexity in bits, although we sometimes measure this in words to simplify the argument. In this case, we have to concern ourselves with the computation model.)

We have considered the computational complexity of a program running on a computer for some specific input x. More precisely, we first fix the machine model and a program that runs on the machine, and then give some input x. Then, its time complexity is given by the number of steps, and space complexity is given by the number of memory cells. However, such an evaluation is too detailed in general. For example, suppose we want to compare two algorithms A and B. Then, we decide which machine model to use, and implement the two algorithms on it as programs, say P_A and P_B. (Here we distinguish an algorithm from a program. An algorithm is a description of how to solve a problem, and it is independent from the machine model. In contrast, a program is written in some programming language (such as C, Java, or Python). That is, a program is a description of the algorithm and depends on the machine and the language.) However, it is not a good idea to compare them for every input x. Sometimes A runs faster than B, but B may run faster more often. Thus, we asymptotically compare the behavior of two algorithms for "general inputs." However, it is not easy to capture the behavior for infinitely many different inputs. Therefore, we measure the computational complexity of an algorithm by the worst case scenario for each length n of input. For each length n, we have 2^n different inputs from $000\dots0$ to $111\dots1$. Then we take the worst computation time and the worst computation space for all of the inputs of length n. Now, we can define two functions $t(n)$ and $s(n)$ to measure these computational complexities. More precisely, the time complexity $t(n)$ of an algorithm A is defined as follows:

Time complexity of an algorithm A:

Let P_A be a program that realizes algorithm A. Then, the time complexity $t(n)$ of the algorithm A is defined by the longest computation steps of P_A under all possible inputs x of length n.

Here we note that there is no clear distinction between algorithm A and program P_A. In this step, we consider that there are a few gaps between them from the viewpoint of time complexity. We can define the space complexity $s(n)$ of algorithm A in a similar way. Now we can consider the computational complexity of algorithm A. For example, for the time complexities of two algorithms A and B, say $t_A(n)$ and $t_B(n)$, if we have $t_A(n) < t_B(n)$ for every n, we say that algorithm A is faster than B.

The notion guarantees that it is sufficient to complete the computation even if we prepare these computational resources for the worst case. In other words, the computation will not fail for any input. Namely, this approach is based on pessimism. From the viewpoint of theoretical computer science, it is the standard approach because it is required to work well in any of the cases including the worst case. For example, another possible approach could be "average case" complexity. However, in general, this is much more difficult because we need the distribution of the input to compute the average case complexity. Of course, such a **randomization** would proceed well in some cases. Some randomized algorithms and their analyses will be discussed in Sect. 3.3.4. A typical randomized algorithm is that known as **quick sort**, which is used quite widely in real computers. The quick sort is an interesting example in the sense that it is not only used widely, but also not difficult to analyze.

1.3 Data Structures

We next turn to the topic of data structures. Data structures continue to be one of the major active research areas. In theoretical computer science, many researchers investigate and develop fine data structures to solve some problems more efficiently, smarter, and simpler. However, in this textbook, we only use the most basic and simplest data structure known as an "array." However, if we restrict ourselves to only using arrays, the descriptions of algorithms become unnecessarily complicated. Therefore, we will introduce "multi-dimensional array," "queue," and "stack" as extensions of the notion of arrays. They can be implemented by using an array, and we will describe them later.

1.3.1 Variable

In an ordinary computer, as shown in the RAM model, there is a series of memory cells. They are organized and each cell consists of a word, which is a collection of bits. Each cell is identified by a distinct **address**. However, when you write a program, it is not a good approach to identify these addresses directly. The program will not be readable, and it has no portability. Therefore, instead of identifying an address directly, we usually identify a memory cell by some readable name. This alphabetical name is known as a **variable**. That is, you can use your favorite name to identify and access your data. For example, you can use the variable A to read an *integer* 1, or another variable S to check whether it remembers the *letter* "A."

Usually, a variable has its own *type* in the form of an attribute. Typically, there are variables of the type integers, real numbers, strings, and so on. This type corresponds to the rule of binary encoding. That is, every datum is represented by a binary string consisting of a sequence of 0 and 1, and the rule of representation is defined according to the type. For example, when variable A records an *integer* 1, concretely, the binary data 00000001 is stored in the binary system, and when another variable B records a *character* 1, the binary data 00110001 is stored in it according to the ASCII encoding rule. The encoding rules are standardized by industrial standards, and hence, we can exchange data between different computers on the Web.

ASCII code and Unicode

Essentially, the integer 1 and the letter A are represented by binary data, that is, a sequence of 0 and 1. Usually, an integer is represented in the binary system; for example, a binary string 00000001 represents the integer 1, from where we count upwards as 00000001, 00000010, 00000011, 00000100, 00000101, ... That is, 00000101 in the binary system represents the integer 5. On the other hand, letters are represented by **ASCII** code, which is the standard rule. For example, the letter "A" is represented by 01101001, and "a" is represented by 10010111. The 8-bit ASCII code corresponding to 256 characters is provided in a table. Therefore, 8-bit data can represent 256 letters in total. Every letter used in the world can be represented by a new standard known as **Unicode**, which was enacted in the 1980s. However, real world is not as simple. Before Unicode, for example, we had three different coding rules in Japan, and they are still active. Such a standard is meaningless unless many people use it.

1.3.2 Array

When we refer to data processed by computer, each datum is recorded in a memory cell, and we identify it by a variable name. To do that without confusion, we have to consider different names for each data. However, it is not that easy. Suppose you are a teacher who has to process the mathematics examination scores of one hundred students. It is not realistic to name each datum in this case. What happens if you have other scores in the next semester, and what if you have another course of programming for these students? How can we compute an average score for these one hundred students? We solve these tasks by using a set of variables known as an **array**. This idea is similar to a progression, or a sequence of numbers, in mathematics. For example, we sometimes consider a progression such as $a_0 = 0, a_1 = 1, a_2 = 1, a_3 = 2, a_4 = 3$, and so on. In this case, the progression is identified by the letter a, and the ith number in it is identified by the **index** i, which is the subscript naming the progression. That is, even if we have the same name a, the numbers are different if they have different indices. Thus, the notion of an array is the same as that of a progression. We name a set of variables and identify each of them by its index. In a progression, we usually write a_1, a_2, but we denote an array as $a[1]$, $a[2]$. That is, $a[1]$ is the 1st element of array a, and $a[2]$ is the second element of array a. In a real computer (or even in the RAM model), the realization of an array is simple; the computer provides a sequence of memory cells, and names the area by the array name, with $a[i]$ indicating the ith element of this memory area. For example, we assume that the examination scores of a hundred students are stored in the array named $Score$. That is, their scores are stored from $Score[1]$ to $Score[100]$. For short, hereafter, we will say that "we assume that each examination score is stored in the array $Score[]$.”[5] Now we turn to the first algorithm in this book. The following algorithm Average computes the average of the examination scores $Score[]$ of n students. That is, calling the algorithm Average in the form of “Average($Score$, 100),” enables us to compute the average of one hundred examination scores of one hundred students, where $Score[i]$ already records the examination score of the ith student.

```
Algorithm 1: Average(S, n) computes the average score of n scores.
  Input  : S[]: array of score data, n: the number of scores
  Output: Average score
1 $x \leftarrow 0$;
2 for $i \leftarrow 1, 2, \ldots, n$ do
3   $x \leftarrow x + S[i]$;
4 end
5 output $x/n$;
```

We familiarize ourselves with the description of algorithms in this book by looking closely at this algorithm. First, every algorithm has its own name, and some **parameters** are assigned to the algorithm. In Average(S, n), “Average” is the name of this algorithm, and S and n are its parameters. When we call this algorithm in the form of “Average($Score$, 100),” the array $Score$ and an integer 100 are provided as parameters of this algorithm. Then, the parameter S indicates the array $Score$. That is, in this algorithm, when the parameter S is referred to, the array $Score$ is accessed by the reference. On the other hand, when the parameter n is referred to, the value 100 is obtained. In the next line, **Input**, we describe the type of the input, and the contents of the output are described in the line **Output**. Then the body of the algorithm follows. The numbers on the left are labels to be referred to in the text. The steps in the algorithm are usually performed from top to bottom; however, some **control statements** can change this flow. In this algorithm Average, the **for** statement from line 2 to line 4 is the control statement, and it repeats the statement at line 3 for each $i = 1, 2, \ldots .n$, after which line 5 is executed and the program halts. A semicolon (;) is used as delimiter in many programming languages and also in this book.

Next, we turn to the contents of this algorithm. In line 1, the algorithm performs a **substitution** operation. Here it prepares a new variable x at some address in a memory cell, and initializes this cell by 0. In line 2, a new variable i is prepared, and line 3 is performed for each case of $i = 1, i = 2, i = 3, \ldots, i = n$. In line 3, variable x is updated by the summation of variable x and the ith element of the array S. More precisely,

- First, $x = 0$ by the initialization in line 1.
- When $i = 1$, x is updated by the summation of x and $S[1]$, which makes $x = S[1]$.
- When $i = 2$, $x = S[1]$ is updated by the summation of x and $S[2]$, and hence, we obtain $x = S[1] + S[2]$.

[5] **How to write an array**:
In this book, an array name is denoted with []. That is, if a variable name is denoted with [], this is an array.

- When $i = 3$, $S[3]$ is added to $x = S[1] + S[2]$ and substituted to x, therefore, we have $x = S[1] + S[2] + S[3]$.

In the same manner, after the case $i = n$, we finally obtain $x = S[1] + S[2] + S[3] + \cdots + S[n]$. Lastly, the algorithm outputs the value divided by $n = 100$, which is the average, and halts.

Exercise 3 Consider the following algorithm SW(x, y). This algorithm performs three substitutions for two variables x and y and outputs the results. Explain what this algorithm executes. What is the purpose of this algorithm?

Algorithm 2: Algorithm SW(x, y) operates two variables x and y.

```
Input  : Two data $x$ and $y$
Output: Two processed $x$ and $y$
$x \leftarrow x + y$;
$y \leftarrow x - y$;
$x \leftarrow x - y$;
output $x$ and $y$;
```

Algorithm and its Implementation. When you try to run some algorithms in this book by writing some programming language on your system, some problems may occur, which I do not mention in this book. For example, consider the implementation of algorithm Average on a real system. When $n = 0$, the system will output an error of "division by zero" in line 5; therefore, you need to be careful of this case. The two variables i and n could be natural numbers, but the last value obtained by x/n should be computed as a real number. In this book, we do not deal with the details of such "trivial matters" from the viewpoint of an algorithm; however, when you implement an algorithm, you have to remember to consider these issues.

Reference of variables. In this textbook, we do not deal with the details of the mechanism of "parameters." We have to consider what happens if you substitute some value into the parameter $S[i]$ in the algorithm Average(S, n). This time, we have two cases; either $Score[i]$ itself is updated or not. If $S[]$ is a copy of $Score[]$, the value of $Score[i]$ is not changed even if $S[i]$ is updated. In other words, when you use the algorithm, you have two choices: one is to provide the data itself, and the other is to provide a copy of the data. In the latter case, the copy is only used in the algorithm and does not influence what happens outside of the algorithm. After executing the algorithm, the copy is discarded, and the memory area allocated to it is released. In most programming languages, we can use both of these ways, and we have to choose one of them.

Subroutines and functions. As shown for Average(S, n), in general, an algorithm usually computes some function, outputs results, and halts. However, when the algorithm becomes long, it becomes unreadable and difficult to maintain. Therefore, we divide a complex computation into some computational units according to the tasks they perform. For example, when you design a system for maintaining students' scores, "taking an average of scores" can be a computational unit, "taking a summation of small tests and examinations for a student" can be a unit, and "rearranging students according to their scores" can be another. Such a unit is referred to as a **subroutine**. For example, when we have two arrays, $A[]$ containing a hundred score data and $B[]$ a thousand, we can compute their two averages as follows:

```
Algorithm 3: PrintAve2($A$, $B$) outputs two averages of two arrays.
  Input  : Two arrays $A[]$ of 100 data and $B[]$ of 1000 data
  Output: Each average
  output "The average of the array $A$ is";
  Average($A$, 100);
  output "The average of the array $B$ is";
  Average($B$, 1000);
```

Suppose we only need the better of the two averages of $A[]$ and $B[]$. In this case, we cannot use Average(S, n) as it is. Each Average(S, n) simply outputs the average for a given array, and the algorithm cannot process these two averages at once. In this case, it is useful if a subroutine "returns" its computational result. That is, when some algorithm "calls" a subroutine, it is useful that the subroutine returns its computation result for the algorithm to use. In fact, we can use subroutines in this way, and this is a rather standard approach. We call this subroutine **function**.[6] For example, the last line of the algorithm Average

```
output $x/n$;
```

is replaced by

```
return $x/n$;
```

Then, the main algorithm calls this function, and uses the returned value by substitution. Typically, it can be written as follows.

```
Algorithm 4: PrintAve1($A$, $B$) outputs the better one of two averages of two arrays $A$ and $B$.
  Input  : Two arrays $A[]$ of 100 data and $B[]$ of 1000 data
  Output: Better average
1 $a$ ←Average($A$, 100);
2 $b$ ←Average($B$, 1000);
3 if $a > b$ then
4     output $a$;
5 else
6     output $b$;
7 end
```

The set consisting of **if**, **then**, **else**, and **end** in lines 3 to 7 are referred to as the **if-statement**. The **if**-statement is written as

[6] **Subroutine and Function**:
In current programming languages, it is popular that "everything is function." In these languages, we only call functions, and a subroutine is considered as a special function that returns nothing.

```
if (condition) then
    (statements performed when the condition is true)
else
    (statements performed when the condition is false)
end
```

In the above algorithm, the statement does the following: When the value of variable a is greater than the value of b, it outputs the value of a, and otherwise (in the case of $a \leq b$) it outputs the value of b.

It is worth mentioning that, in lines 1 and 2, the statement is written as $y = \sin(x)$ or $y = f(x)$, which are known as functions. As mentioned above, in the major programming languages, we consider every subroutine to be a function. That is, we have two kinds of functions that return a single value and another that returns no value. We note that any function returns at most one value. If you need a function that returns two or more values, you need some tricks. One technique is to prepare a special type of data that contains two or more data values; alternatively, you could use the parameters updated in the function to return the values.

1.3.3 Multi-dimensional Array

Suppose we want to draw some figures on the computer screen. Those of you who enjoy programming to create a video game, have to solve this problem. Then, for example, it is reasonable that each pixel can be represented as "a point of brightness 100 at coordinate (7, 235)." In this case, it is natural to use a two-dimensional array such as $p[7, 235] = 100$. In Sect. 1.3.2, we have learnt one-dimensional arrays. How can we extend this notion? For example, when we need a two-dimensional array of size $n \times m$, it seems to be sufficient to prepare n ordinary arrays of size m. However, it can be tough to prepare n distinct arrays and use them one by one. Therefore, we consider how we can realize a two-dimensional array of size $n \times m$ by one large array of size $n \cdot m$ by splitting it into m short arrays of the same size. In other words, we realize a two-dimensional array of size $n \times m$ by maintaining the index of one ordinary array of size $n \cdot m$.

More precisely, we consider a mapping between a two-dimensional array $p[i, j]$ $(0 \leq i \leq n-1, 0 \leq j \leq m-1)$ of size $n \times m$ and an ordinary array $a[k]$ $(0 \leq k \leq n \times m - 1)$ of size $n \cdot m$.[7] It seems natural to divide the array $a[]$ into m blocks, where each block consists of n items. Based on this idea, we can design a mapping between $p[i, j]$ and $a[k]$ such that $k = j \times n + i$. It may be easy to understand by considering the integer k on the n-ary system, with the upper "digit" being j, and the lower "digit" i.

[7] **Index of an array**:
For humans, it is easy to see that the index has the values 1 to n. However, as with this mapping, it is sometimes reasonable to use the values 0 to $n-1$. In this book, we will use both on a case-by-case basis.

Exercise 4 ☕☕ Show that the corresponding $k = j \times n + i$ results in a one-to-one mapping between $p[i, j]$ and $a[k]$.

Therefore, when you access the virtual two-dimensional array $p[i, j]$, access $a[j \times n + i]$ automatically instead. Some readers may consider that such an "automation" should be done by computer. It is true. Most major programming languages support such multi-dimensional arrays in their standards, and they process these arrays using ordinary arrays in the background in this manner. Hereafter, we will use multi-dimensional arrays when they are necessary.

Exercise 5 ☕☕ How can we represent a three-dimensional array $q[i, j, k]$ ($0 \le i < n, 0 \le j < m, 0 \le k < \ell$) by an ordinary array $a[]$?

Tips for implementation. Most programming languages support multi-dimensional arrays, which are translated into one-dimensional (or ordinary) arrays. On the other hand, in real computers, memory is used more efficiently when consecutive memory cells are accessed. This phenomenon mainly comes from a technique known as **caching**. In general, it takes time to access a "large" amount of memory; therefore, the computer prepares "small and fast" (and expensive) memory, and pre-reads some blocks from the main large memory. This is known as caching, and this small and fast memory is known as cache memory. This is one of the major techniques to construct a faster computer. By this mechanism, when a multi-dimensional array is realized by an ordinary array, the efficiency of the program can change the ordering in which the array is accessed. Oppositely, accessing scattered data in the large memory could slow down the program because the computer updates its cache every time. A typical example is the initialization of a two-dimensional array $p[i, j]$. We consider the following two implementations:

Algorithm 5: Initialization Init1(p) of two-dimensional array

```
Input  : Two-dimensional array $p[]$ of size $n \times m$.
Output: $p[]$ where $p[i, j] = 0$ for all $i$ and $j$
for $i \leftarrow 0, 1, \dots n - 1$ do
    for $j \leftarrow 0, 1, \dots, m - 1$ do
        $p[i, j] \leftarrow 0$ ;                    /* $i$ will be changed after $j$ */
    end
end
```

and

Algorithm 6: Initialization Init2(p) of two-dimensional array

```
Input  : Two-dimensional array $p[]$ of size $n \times m$.
Output: $p[]$ where $p[i, j] = 0$ for all $i$ and $j$
for $j \leftarrow 0, 1, \dots, m - 1$ do
    for $i \leftarrow 0, 1, \dots n - 1$ do
        $p[i, j] \leftarrow 0$ ;                    /* $j$ will be changed after $i$ */
    end
end
```

It is easy to see that they have the same efficiency from the viewpoint of a theoretical model. However, their running times differ on some real computers. (By the way, I added some comments between /* and */ in the algorithms. They are simply comments to aid the reader and have no meaning in the algorithms.) When you try programming, know-how such as this is useful. Recent compilers are so smart that such know-how is sometimes managed by the compilers, which refer to this as **optimization**. In that case, both algorithms require the same amount of time to run (or the resulting binary executable files written in machine language are the same), and your system is praiseworthy!

1.3.4 Queues and Stacks

Let us imagine that we are forming a line to buy tickets for a museum. In front of the ticket stand, many people form a line. In this line, the first person is the first one to receive service. This mechanism is known as **queuing**. A queue is one of the basic data structures. The operations required by a queue are listed as follows:

- Add new item into Q,
- Take the first element in Q and remove it from Q, and
- Count the number of elements in Q.

Note that if these three operations are realized for Q, the way they are implemented does not matter for users.

Next, let us again imagine that you accumulate unprocessed documents on your desk (I do not need to imagine this because the situation is real). The documents are piled up on your desk. The bunch of paper crumble to the touch; if you remove one from the middle of the pile, they will tumble down, and a great deal of damage will be done. Therefore, you definitely have to take one from the top. Now we consider this situation from the viewpoint of sheets, in which case the service will be provided to the last (or latest) sheet first. That is, the process occurs in the order opposite to that in which they arrived. When data is processed by a computer, sometimes this opposite ordering (or LIFO) is more useful than queuing (or FIFO).[8] Thus, this idea has also been adopted as a basic structure. This structure is known as a **stack**. Similarly, the operations required by a stack S are as follows:

- Add new item into S,
- Take the last element in S, and remove it from S, and
- Count the number of elements in S.

The operations are the same as for a queue, with the only different point being the ordering.

[8] **FIFO**:
The property determining that the first data will be processed first is known as FIFO, which stands for First In, First Out. We also show the other notion of a “stack,” which is the opposite of a queue, and is known as LIFO or FILO. These stand for Last In, First Out and First In, Last Out.

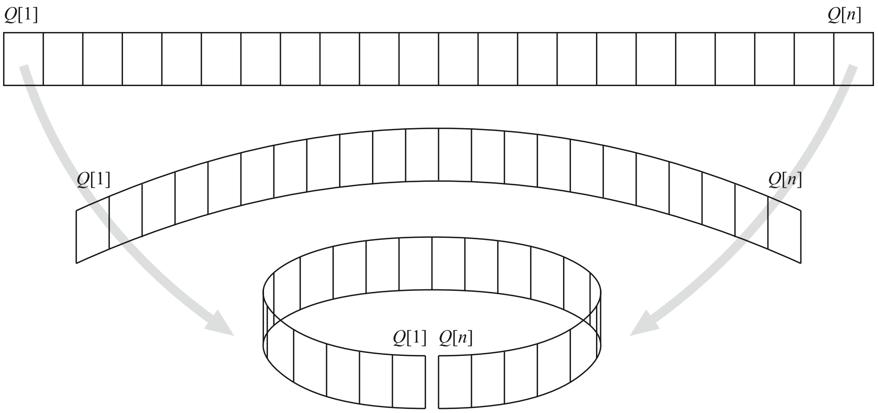


Fig. 1.5 Joining both endpoints of the array, we obtain a loop data structure

Both queues and stacks are frequently used as basic data structures. In fact, they are easy to implement by using an array with a few auxiliary variables. Here we give the details of the implementation. First, we consider the implementation of a queue Q. We use an array Q ($Q[1], \ldots, Q[n]$) of size n and two variables s and t. The variable s indicates the starting point of the queue in Q, and the variable t is the index such that the next item is stored at $Q[t]$. That is, $Q[t]$ is empty, $Q[t-1]$ contains the last item. We initialize s and t as $s = 1$ and $t = 1$. Based on this idea, a naive implementation would be as follows.

Add a new element x **into** Q: For given x, substitute $Q[t] \leftarrow x$, and update $t \leftarrow t + 1$.
Take the first element in Q: Return $Q[s]$, and update $s \leftarrow s + 1$.
Count the number of elements in Q: The number is given by $t - s$.

It is not difficult to see that these processes work well in general situations. However, this naive implementation contains some bugs (errors). That is, they do not work well in some special cases. Let us consider how we can fix them. There are three points to consider.

The first is that the algorithm tries to take an element from an empty Q. In this case, the system should output an error. To check this, when this operation is applied, check the number of elements in Q beforehand, and output an error if Q contains the zero element.

The next case to consider is that $s > n$ or $t > n$. If these cases occur, more precisely, if s or t become $n + 1$, we can update the value by 1 instead of $n + 1$. Intuitively, by joining two endpoints, we regard the linear array Q as an array of loop structure (Fig. 1.5). However, we face another problem in this case: we cannot guarantee the relationship $s \leq t$ between the head and the tail of the queue any more. That is, when some data are packed into $Q[s], Q[s + 1], \ldots, Q[n], Q[1], \ldots, Q[t]$ through the ends of the array, the ordinary relationship $s < t$ turns around and we have

$t < s$. We can decide whether this occurs by checking the sign of $t - s$. Namely, if the number of elements of Q satisfies $t - s < 0$, it means the data are packed through the ends of the array. In this case, the number of elements of Q is given by $t - s + n$.

Exercise 6 ☕ When $t - s < 0$, show that the number of elements in the queue is equal to $t - s + n$.

The last case to consider is that there are too many data for the queue. The extreme case is that all $Q[i]$s contain data. In this case, "the next index" t indicates "the top index" s, or $t = s$. This situation is ambiguous because we cannot distinguish it from that in which Q is empty. We simplify this by deciding that the capacity of this queue is $n - 1$. That is, we decide that $t = s$ means the number of elements in Q is zero, thereby preventing the nth data element from being packed into Q. The last element of Q is useless, but we do not mind it for simplicity. To be sure, before adding a new item x into Q, the algorithm checks the number of elements in Q. If x is the nth item, return "error" at this time, and do not add it. (This error is called **overflow**.)

Refining our queue with these case analyses enables us to safely use the queue as a basic data structure. Here we provide an example of the implementation of our queue. The first subroutine is initialization:

```
Algorithm 7: Initialization of queue InitQ($n, t, s$)
  Input  : The size $n$ of queue and two variables $t$ and $s$
  Output: An array $Q$ as a queue
  $s \leftarrow 1$;
  $t \leftarrow 1$;
  Allocate an array $Q[1], \ldots, Q[n]$ of size $n$;
```

Next we turn to the function "sizeof," which returns the number of elements in Q. By considering Exercise 6, $|t - s| < n$ always holds. Moreover, if $t - s \geq 0$, it provides the number of elements. On the other hand, if $t - s < 0$, the queue is stored in $Q[s], Q[s + 1], \ldots, Q[n], Q[1], \ldots, Q[t]$. In this case, the number of elements is given by $t - s + n$. Therefore, we obtain the following implementation:

```
Algorithm 8: Function sizeof($Q$) that returns the number of elements in $Q$.
  Input  : Queue $Q$
  Output: The number of elements in $Q$
  if $t - s \geq 0$ then
      return $t - s$;
  else
      return $t - s + n$;
  end
```

When a new element x is added into Q, we have to determine whether Q is full. This is implemented as follows:

Algorithm 9: Function push(Q, x) that adds the new element x into Q.

```
Input  : Queue $Q$ and element $x$
Output: None
if sizeof($Q$)$= n - 1$ then
    output “overflow” and halt;
else
    $Q[t] \leftarrow x$;
    $t \leftarrow t + 1$;
    if $t = n + 1$ then $t \leftarrow 1$;
end
```

Similarly, when we take an element from Q, we have to check whether Q is empty beforehand. The implementation is shown below:

Algorithm 10: Function pop(Q) that takes the first element from Q and returns it.

```
  Input  : Queue $Q$
  Output: The first element $y$ in $Q$
1 if sizeof($Q$)$= 0$ then
2     output “Queue is empty” and halt;
3 else
4     $q \leftarrow Q[s]$;
5     $s \leftarrow s + 1$;
6     if $s = n + 1$ then $s \leftarrow 1$;
7     return $q$;
8 end
```

Note: In many programming languages, when a function F returns a value, the control is also returned to the procedure that calls F. In pop(Q), it seems that we can return the value $Q[s]$ at line 4; however, then we cannot update s in lines 5 and 6. Therefore, we temporally use a new variable q and return it in the last step.

It is easier to implement stack than queue. When using queue, we have to manage the top element s and the tail element t; however, the top element of a stack is fixed, thus we do not need to manage it by a variable. More precisely, we can assume that $S[1]$ is always the top element in the stack S. Therefore, the cyclic structure in Fig. 1.5 is no longer used. Based on this observation, it is not difficult to implement stack.

Exercise 7 ☕☕ Implement the stack S.

Some readers may consider these data structure to be almost the same as the original one-dimensional array. However, this is not correct. These abstract models clarify the data properties. In Chap. 4, we will see a good example for using such data structure.

Next data structure of array?

The next aspect of an array is called a "pointer." If you understand arrays and pointers properly, you have learnt the basics of a data structure. You can learn any other data structure by yourself. If you have a good understanding of an array and do not know the pointer, I strongly recommend to learn it. Actually, a pointer corresponds to an address of the RAM machine model. In other words, a pointer *points* to some other data in a memory cell. If you can imagine the notion of the RAM model, it is not difficult to understand the notion of a pointer.

1.4 The Big-O Notation and Related Notations

As discussed in Sect. 1.2, the computational complexity of an algorithm A is defined by using a function of the length n as input. Then we consider the worst-case analysis for many variants of inputs (there are 2^n different inputs for length n). Even so, in general, the estimation of an algorithm is difficult. For example, even if we have an algorithm A whose time complexity $f_A(n)$ is $f_A(n) = 503n^3 + 30n^2 + 3$ if n is odd, and $f_A(n) = 501n^3 + n + 1000$ if n is even, what can we conclude from them? When we estimate and compare the efficiency of algorithms, we have to say that these detailed numbers have little meaning. Suppose we have another algorithm B for the same problem that runs in $f_B(n) = 800n + 12345$. In this situation, for these two algorithms A and B, which is better? Most readers agree that B is faster in some way. To discuss this intuition more precisely, D. E. Knuth proposed using the big-O notation. We will adopt this idea in this book. The key of the big-O notation is that we concentrate on the main factor of a function by ignoring nonessential details. By this notion, we can discuss and compare the efficiency of algorithms independent from machine models. We focus on the asymptotic behavior of algorithms for sufficiently large n. This enables us to discuss the approximate behavior of an algorithm with confidence.

1.4.1 Warning

I dare to give a warning before stating the definition. The big-O notation is a notation intended as the upper bound of a function. That is, if an algorithm A solves a problem P in $O(n^2)$ time, intuitively, "the time complexity of algorithm A is bounded above by n^2 within a constant factor for any input of length n." We need to consider two points in regard to this claim.

- There may exist a faster algorithm than A. That is, to solve problem P, while algorithm A solves it in $O(n^2)$ time, it may not capture the difficulty of P. Possibly, P can be solved much easier by some smart ideas.

- This $O(n^2)$ provides an upper bound of the time complexity. We obtain the upper bound $O(n^2)$ by analysis of the program that represents A in some way. However, the program may actually run in time proportional to n. In this time, our analysis of the algorithm has not been appropriate.

The first point is not difficult to understand. There may be a gap between the essential difficulty of the problem and our algorithm, because of the lack of understanding of some properties of the problem. We provide such an example in Sect. 2.2. Of course, there exist some "optimal" algorithms that cannot be improved any further in a sense. In Sect. 3.3.5, we introduce such interesting algorithms.

The second point is a situation that requires more consideration. For example, when you have $f(n) = O(n^2)$, even some textbooks may write this as "$f(n)$ is proportional to n^2." However, this is wrong. Even if you have an algorithm A that runs in $O(n^3)$, and another algorithm B that runs in $O(n^2)$, A can be faster than B when you implement and perform these algorithms. As we will see later, the notation "$O(n^3)$" indicates that the running time of A can be bounded above by n^3 in a constant factor. In other words, this is simply an upper bound of the running time that we can prove. For the same reason, for one algorithm, without modification of the algorithm itself, its running time can be "improved" from $O(n^3)$ to $O(n^2)$ by refining its analysis.

When you prefer to say "a function $f(n)$ is proportional to n^2" with accuracy, you should use the notation $\Theta(n^2)$. In this book, to make the difference clear, we will introduce O notation, Ω notation, and Θ notation. We also have o notation and ω notation, but we did not introduce them in this book. However, the O notation is mainly used. That is, usually we need to know the upper bound of the computational resources in the worst case. For a novice reader, it is sufficient to learn the O notation. However, it is worth remembering that the O notation only provides the upper bound of a function.

*1.4.2 Big-*O *Notation*

We first give a formal definition of the O notation. Let $g(n)$ be a function on natural numbers n. Then $O(g(n))$ is the set of functions defined as follows.

O notation:

$$O(g(n)) = \{f(n) \mid \text{ there exist some positive constants } c \text{ and } n_0, \\ \text{for all } n > n_0, 0 \le f(n) \le cg(n) \text{ holds.}\}$$

If a function $f(n)$ on natural numbers n belongs to $O(g(n))$, we denote by $f(n) = O(g(n))$.

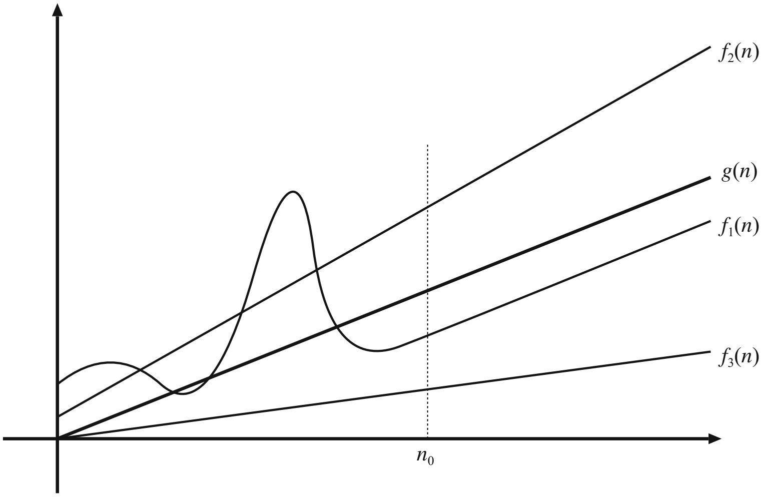


Fig. 1.6 Functions and the big O notation

In this definition, we have some points to consider. First of all, $O(g(n))$ is a set of functions. Some readers familiar with mathematics may think "then, should it be denoted by $f(n) \in O(g(n))$?" This is correct. Some people denote it in this way; however, they are (unfortunately) the minority. In this textbook, following the standard manner in this society, I adopt the style $f(n) = O(g(n))$. That is, this "=" is not an equal to symbol in a strict sense. Although we write $3n^2 = O(n^2)$, we never write $O(n^2) = 3n^2$. This "=" is not symmetrical! This is not a good manner, which may throw beginners into confusion. However, in this textbook, we adopt the standard manner that is used in this society.

The next point is that we use n_0 to avoid finite exceptions. Intuitively, this n_0 means that "we do not take care of finite exceptions up to n_0." For example, the function $f_1(n)$ in Fig. 1.6 varies drastically when n is small, but stabilizes when n increases. In such a case, our interest focuses on the asymptotic behavior for sufficiently large n. Therefore, using the constant n_0, we ignore the narrow area bounded by n_0. If the behavior of $f_1(n)$ can be bounded above by $g(n)$ for sufficiently large n, we denote it by $f_1(n) = O(g(n))$.

We cannot ignore the constant c. By doing this, we can say that "we do not take care of the constant factor." For example, the function $f_2(n)$ is obtained from $g(n)$ by multiplying by a constant, and adding another constant. The philosophy of the big-O notation is that these "differences" are not essential for the behavior of functions and ignorable. This notion can be approximate, in fact, $O(g(n))$ contains quite a smaller function than $g(n)$ such as $f_3(n)$ in Fig. 1.6.

As learnt in Sect. 1.1, an algorithm consists of a sequence of basic operations, and the set of basic operations depends on its machine model. Even in a real computer, this set differs depending on its CPU. That is, each CPU has its own set of basic operations designed by the product manufacturer. Therefore, when you implement an algorithm, its running time depends on your machine. Typically, when you buy a new computer, you may be surprised that the new computer is much faster than the old one. To measure, compare, and discuss the efficiency of algorithms in this situation, we require some framework that is approximate in terms of "finite exceptions" and "constant factors."

Exercise 8 ☕☕☕ For each of the following, prove it if it is correct, or disprove it otherwise.

(1) $2n + 5 = O(n)$.
(2) $n = O(2n + 5)$.
(3) $n^2 = O(n^3)$.
(4) $n^3 = O(n^2)$.
(5) $O(n^2) = O(n^3)$.
(6) $5n^2 + 3 = O(2^n)$.
(7) $f(n) = O(n^2)$ implies $f(n) = O(n^3)$.
(8) $f(n) = O(n^3)$ implies $f(n) = O(n^2)$.

How to pronounce "$f(n) = O(n)$"

This "O" is simply pronounced "Oh"; thus, the equation can be pronounced as "ef of en is equal to big-Oh of en." However, some people also pronounce it as "ef of en is equal to order en."

1.4.3 Other Notations Related to the Big-O Notation

We next turn to the other related notations Ω and Θ, which are less frequently used than O. (We also have the other notations o and ω, which are omitted in this book. See the references for the details.)

In the broad sense, we call these O, Ω, Θ, o, and ω notations "the big-O notations." Namely, the most frequently used is the big-O notation. In this book, we will use "the big-O notation" that is used both in the broad sense and in the narrow sense depending on the context.

Ω **notation**. Let $g(n)$ be a function on natural numbers n. Then $\Omega(g(n))$ is the set of functions defined as follows (it is pronounced as "big-omega" or simply "omega").

Ω **notation**:

$$\Omega(g(n)) = \{f(n) \mid \text{ there exist some positive integers } c \text{ and } n_0, \text{ for all } n > n_0, 0 \le cg(n) \le f(n) \text{ holds. }\}$$

If a function $f(n)$ on natural numbers n belongs to $\Omega(g(n))$, we denote by $f(n) = \Omega(g(n))$.

Similar to the O notation, "=" is not a symmetric symbol.

Comparing this with the O notation, the meaning of this Ω notation becomes clearer. Two of their common properties are summarized as follows:

- We do not take care of finite exceptions under n_0.
- We do not mind a constant factor of c.

The difference is $cg(n) \le f(n)$, that is, the target function $f(n)$ is bounded from below by $g(n)$.

That is, whereas the O notation is used to provide the upper bound of a function, the Ω notation is used to provide the lower bound. Therefore, in the context of computational complexity, the Ω notation is used to show a lower bound of some resources required to perform an algorithm. Here we give a trivial but important example. For some problem P with an input of length n, all input data should be read to solve the problem. Let an algorithm A solve the problem P in $t_A(n)$ time. Then we have $t_A(n) = \Omega(n)$. That is, essentially, $t_A(n)$ cannot be less than n.[9]

In Chap. 3, we will see a more meaningful example of the Ω notation.

Θ **notation**. Let $g(n)$ be a function on natural numbers n. Then $\Theta(g(n))$ is the set of functions defined as follows (it is pronounced as "theta").

Θ **notation**:

$$\Theta(g(n)) = \{f(n) \mid \text{ there exist some positive constants } c_1, c_2, \text{ and } n_0, \text{ for all } n > n_0, 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ holds.}\}$$

If a function $f(n)$ on natural numbers n belongs to $\Theta(g(n))$, we denote it by $f(n) = \Theta(g(n))$.

Considering $O(g(n))$, $\Omega(g(n))$, and $\Theta(g(n))$ as sets of functions, we can observe the following relationship among them:

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

That is, $\Theta(g(n))$ is an intersection of $O(g(n))$ and $\Omega(g(n))$.

[9] **Trivial lower bound**:
To make sure, the assumption that "all data should be read to solve a problem" is quite a natural assumption for most problems. This lower bound is sometimes referred to as a **trivial lower bound**.

Exercise 9 ☕☕ Prove the above equation $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Checking the O and Ω notations, let us consider the meaning of the equation $f(n) = \Theta(g(n))$. The following two margins are again available:

- We do not take care of finite exceptions under n_0.
- We do not mind a constant factor of c.

Within these margins, $f(n)$ is bounded by $g(n)$ both above and below. That is, we can consider $f(n)$ to be proportional to $g(n)$. In other words, $f(n)$ is essentially the same function as $g(n)$ within a constant factor with finite exceptions.

There are some algorithms of which the behavior has been completely analyzed, and their running times are already known. For these algorithms, their efficiency can be represented in Θ notation with their accuracy independent of machine models. Some examples are provided in Chap. 3.

1.5 Polynomial, Exponential, and Logarithmic Functions

Let A be an algorithm for some problem, and assume that its running time $f_A(n)$ is $f_A(n) = 501n^3 + n + 1000$ for any input of length n. When we measure the efficiency of an algorithm, discussion about the detailed coefficients is not productive in general. Even if we "improve" the algorithm and obtain a new complicated algorithm A' that runs in $f_{A'}(n) = 300n^3 + 5n$, it may be better to update the computer and perform A on it. The real running time of A may vary depending on the programmer's skill and the machine model.

Therefore, we usually estimate the running time of an algorithm A by using the O notation introduced in Sect. 1.4 as $f_A(n) = O(n^3)$. Once you become familiar with algorithm design, it may not be rare that you discover some redundant process, refine it, and obtain an algorithm that is, say, n times faster. Therefore, from this viewpoint, even if you improve your algorithm from $501n^3$ to $300n^3$, it is unlikely that this essentially improves the algorithm.

When we evaluate the computational complexity of some problems, we frequently take a global view of the difficulty. If you like to solve some specific problem, in most cases, the problem has some solution, and moreover, even choices are also given. In this case, how about the idea of "checking all possibilities"? That is, from the initial state, checking all possible choices, we eventually reach the solution, don't we? In most cases, this naive intuition is theoretically correct. However, *theoretically* it is true. For example, consider a well-known popular game such as Shogi. The board is fixed. Its size is 9×9, that is, we have at most 81 choices at each step. The number and kinds of pieces are also fixed. The rules of the game are also strictly designed. Then, theoretically, if both players play with their best possible strategies, the game is decided by one of three outcomes: the first player always wins, the second player always wins, or the game is tie. That is, the end of the game has already been decided, even though we do not yet know which is the true end. We would be able to reach the true outcome by checking all possible choices. A program for checking all choices would not seem difficult to a skillful programmer. However, so far, nobody knows the true outcome for a game of chess, Shogi, Go, and so on. This is because such a program never completes its computation. Even in the case of the board consisting of only 9×9 squares, the number of choices runs into astronomical numbers, and the computation cannot be completed in a realistic time.

Here we again consider the fact that the Shogi board *only* has a size of 81. When we consider current computers, which have gigabytes of memory and run at gigahertz, the number 81 seems to be quite small compared to these specifications. Nevertheless, why does the algorithm never complete its computation? The key is a phenomenon known as "exponential explosion." To understand this phenomenon with related ideas, we can partition the functions in this book into three categories: polynomial functions, exponential functions, and logarithmic functions. When we consider the efficiency of an algorithm (from the viewpoints of both time and memory), at the first step, it is quite important to have a clear view as to the category of the running time and memory used. We will describe these three categories of functions with some intuitive image.

Polynomial function. For example, a quadratic function is a typical polynomial function. It is the most natural "function" people imagine when they say function. In general, a polynomial function is defined as follows.

Polynomial function:

A polynomial function is given in the following form:

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_3 x^3 + a_2 x^2 + a_1 x + a_0,$$

where $a_0, a_1, \ldots, a_d$ are constant numbers independent of x, and d is a natural number. More precisely, d is the maximum number with $a_d \neq 0$, and d is referred to as the **degree** of this function $f(x)$.

In terms of computational complexity, if any computational resource is not included in this class, it is said to be **intractable**. That is, even if the problem has a simple solution (e.g., 0/1 or Yes/No), and we can build a program for it (e.g., by exhaustive check of all combinations), the intractable problem cannot be solved in practical time. Here we note that this sentence contains a double negative; that is, any problem not included in this class cannot be solved in practical time. We never say that all problems in this class are tractable. This does not aim to confuse readers. There is a gray zone between tractable and intractable. For example, suppose you succeed in developing an algorithm A that requires $f_A(n) = \Theta(n^d)$ to run. Even if $f_A(n)$ is a polynomial function, it can be quite difficult to obtain a solution in practical time when d is quite large, for example, 1000 or 20000. From a practical viewpoint, a reasonable value of d is not so large, say, $d = 2, 3$ or up to 10. Of course, this "reasonable" is strongly related to the value of n, the specification of the computer, the skill of programmer, and so on. However, at least, $f_A(n)$ cannot be bounded above by some polynomial function, $f_A(n)$ increases drastically when n becomes large, and the program never halts in a practical time.

Exercise 10 ☕☕ Let the running time $f_A(n)$ of an algorithm A be n^{50} steps. That is, the algorithm A performs n^{50} basic operations for any input of size n to compute the answer. Suppose that the processor of your computer executes instructions at a clock speed of 10GHz. That is, your computer can perform 10×10^9 basic operations per second. Compute the value of n such that your computer performs the algorithm A for an input of size n in a practical time.

Exercise 11 ☕ Let us consider algorithms for computing the following polynomial function:

$$f(x) = a_0 + a_1 x + a_2 x^2 + + \cdots + a_{d-1} x^{d-1} + a_d x^d$$

Suppose that an array $a[]$ stores each coefficient with $a[i] = a_i$, and we want to compute $f(x)$ for any given x. We first consider a naive algorithm that computes in the way of $a_0 + a_1 \times x + a_2 \times x \times x + \cdots + a_{d-1} \times x \times \cdots \times x + a_d \times x \times \cdots \times x$ straightforwardly. Then, how many additions and multiplications does the naive algorithm perform? Next, we transform the equation and use the second algorithm to perform the following computation:

$$f(x) = a_0 + x \times (a_1 + x \times (a_2 + x \times (a_3 + \cdots + x \times (a_{d-1} + x \times a_d) \cdots))).$$

Then, how many additions and multiplications does the second algorithm perform?

Exponential function. Typically, an exponential function is the xth power of a constant such as 2^x. We introduce a more general form as follows:

Exponential function:

An exponential function is $f(x) = c^x$ for some constant c with $c > 1$.

When we consider the resources of algorithms, as we only consider increasing functions, we may have $c > 1$ in general. When $c > 1$, any exponential function grows quite rapidly, which is counterintuitive. This phenomenon is known as **exponential explosion**. For example, consider a board game such as Shogi, which allows us some choices. If we have two choices in every turn, if you try to read possible situations after ten turns, you will have to check $2^{10} = 2 \times 2 \times \cdots \times 2 = 1024$ possible cases. (Of course, different choices can imply the same situation, and we have more choices in general; hence, we have to give more careful consideration to the real situation.) It may seem that this number of cases is an easy task for current computers. Anyway, try the following exercise.

Exercise 12 ☕☕ Take a sheet of newspaper. First, fold it in half. Next, fold it in half again. Repeat the folding until you cannot fold it anymore. How many times can you fold it? Can you fold it beyond ten times? Let the thickness of the newspaper be 0.1 mm. How many times do you have to fold it in half until the thickness (or height) of the sheet exceeds that of Mt. Fuji (3776 m)? Moreover, how many times do you have to fold it to go beyond the distance between the moon and Earth (around 38×10^4 km)?

After solving this exercise, you will find that you can go to the moon just by folding a sheet of paper relatively "few times". This "strange incompatibility" with your intuition is the reason why it is difficult to realize the tremendousness of exponential functions.

When we design an algorithm for some problem, it is not difficult to develop an exponential time algorithm in general. The naive idea of "checking all possible cases" tends to lead us to an algorithm requiring exponential time. It is a very important key point to develop a polynomial time algorithm in some way. That requires us to understand the problem deeply, and, by identifying the nontrivial properties of the problem one by one enables us to reach more efficient, faster, and elegant algorithms.

Logarithmic function. We have already seen that exponential functions grow expansively. When you plot an exponential function $y = f(x)$ on the usual Cartesian coordinate system, the curve rises rapidly, immediately rising beyond any graph paper. A logarithmic function is obtained by folding or reflecting some exponential curve along the 45° line $y = x$.

Logarithmic function:

For an exponential function $f(x) = c^x$, its inverse function $g(x) = \log_c x$ is a logarithmic function. We sometimes omit the base and write it as $\log x$ when the base c is 2. When the base c is $e = 2.718\cdots$, the natural logarithmic function $\log_e x$ is denoted by $\ln x$.

Because a logarithmic function is an inverse function of some exponential function, the logarithmic function $y = f(x)$ grows very slowly even if x becomes quite large.

For example, when you use 8 bits, you can distinguish $2^8 = 256$ cases, and 16 bits lead us to $2^{16} = 65536$ cases. An increase of 1 bit causes the number of distinguishable cases to double. This rapid increase is the property of an exponential function. For logarithmic functions, the situation is inverse; even if 65536 cases were to rapidly increase up to twice as much, i.e., 131072, it is sufficient to add only 1 bit to the 16-bit information. That is, for the same reason that an exponential function grows with counterintuitive speed, a logarithmic function grows with counterintuitive slowness.

Exercise 13 ☕☕☕ Draw three graphs of $f(x) = 2^x$, $f(x) = x$, and $f(x) = \log x$ for $1 \leq x \leq 20$.

l'Hospital's rule

In this book, we conclude that any exponential function $f(n) = c^n$ with $c > 1$ grows much faster than any polynomial function $g(n)$ for sufficiently large n with intuitive explanation. Precisely, we do not give any formal proof of the fact $g(n) = O(f(n))$. For example, we can say that $n^{100} = O(1.01^n)$. However, unfortunately, its formal and mathematical proof is not so easy. A simple proof can be performed by using **l'Hospital's rule** that is a theorem about differential functions. By l'Hospital's rule, for two functions $f(n)$ and $g(n)$ (with some omitted conditions here), we have $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f'(n)/g'(n)$. In our context, we can differentiate any number of times. We can observe that any polynomial function $g(n) = O(n^d)$ for some positive integer d becomes a constant after differentiating d times. On the other hand, any exponential function $f(n)$ will still be exponential function after d differentiations. (Typically, differentiating $f(n) = e^n$ becomes $f'(n) = e^n$ again.) Therefore, for example, after 100 differentiations of $n^{100}/1.01^n$, we have $\lim_{n\to\infty} n^{100}/1.01^n = 0$, which implies $n^{100} = O(1.01^n)$.

1.5.1 Harmonic Number

Analyses of algorithms sometimes lead us to impressive equations. In this section, we demonstrate one of them known as **harmonic number**. The definition of the nth harmonic number $H(n)$ is as follows.[10]

The nth harmonic number $H(n)$:

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

That is, the summation of the first n elements in the **harmonic series** $\sum_{i=1}^{\infty} \frac{1}{i}$ is known as the nth harmonic number, which is denoted by $H(n)$. The harmonic number has quite a natural form, and it has the following interesting property.

$$\lim_{n\to\infty} (H(n) - \ln n) = \gamma,$$

where γ is an irrational number $\gamma = 0.57721\ldots$ known as **Euler's constant**. That is, $H(n)$ is very close to $\ln n$, with quite a small error.

[10] **Harmonic number**:
In the wide sense, the harmonic number is defined by the summation of the inverse of the sequence of numbers with a common difference (or the denominators are a sequence of numbers with a common difference), in the narrow sense, the simplest sequence $\sum_{i=1}^{\infty} \frac{1}{i}$ is used to define the harmonic number.

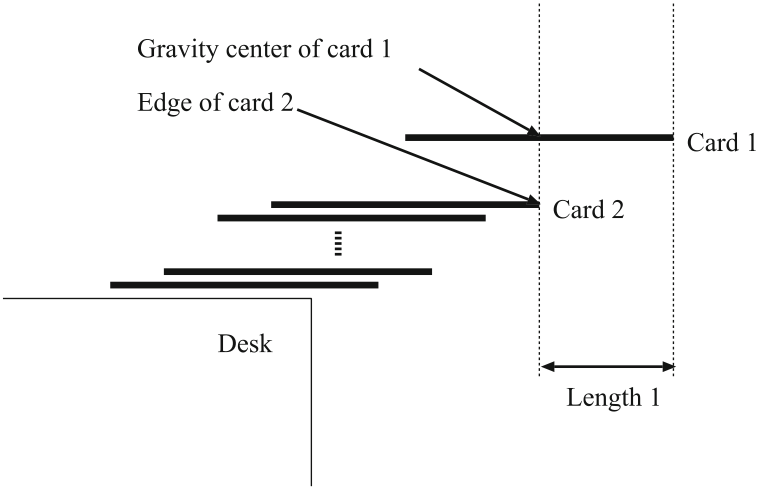


Fig. 1.7 Card stack problem

We do not pursue this interesting issue in any detail; rather, we simply use the useful fact that **the nth harmonic number is almost equal to** $\ln n$. The harmonic number is in natural form, and hence, it sometimes appears in the analysis of an algorithm. We will encounter it a few times. Intuitively, the harmonic number grows quite slowly as n increases because it is almost $\ln n$.

Card stack problem. Here we will see a typical situation in which the harmonic number appears. You have quite a number of cards. You would like to stack the cards on your desk such that the card at the top protrudes beyond the edge of the desk. Then, how far can you position the card? Intuitively, it seems impossible to position the topmost card such that the entire card protrudes beyond the desk. Are you sure about this?

In reality, you have to stack the cards from bottom to top, of course. However, to simplify the explanation, we reverse the time and first consider the last card. (We might consider slipping in a card at the bottom of the stack at each step.) Without loss of generality, our card has the width 2 and weight 1. Cards are identified by numbering them as $1, 2, 3, \ldots$ from the top.

Let us consider the card at the top, card 1. We position this card as far as the edge of the next card, card 2 (Fig. 1.7). At the limit, the center of card 1 is on the edge of card 2. That is, card 1 can be moved beyond half of its length (or length 1) from card 2. The next step is for positioning card 2. Let x_2 be the length between the center of gravity of these two cards and the right edge of card 2 (Fig. 1.8). Then, at this point the centers of cards 1 and 2 are balanced in a seesaw manner (distance × weight), that is, they are balanced at the right and left sides. Therefore, $(1 - x_2) \times 1$ for the left card 2, and $x_2 \times 1$ for the right card 1 are balanced. That is, we have

$$(1 - x_2) \times 1 = x_2 \times 1,$$

and hence $x_2 = 1/2$.

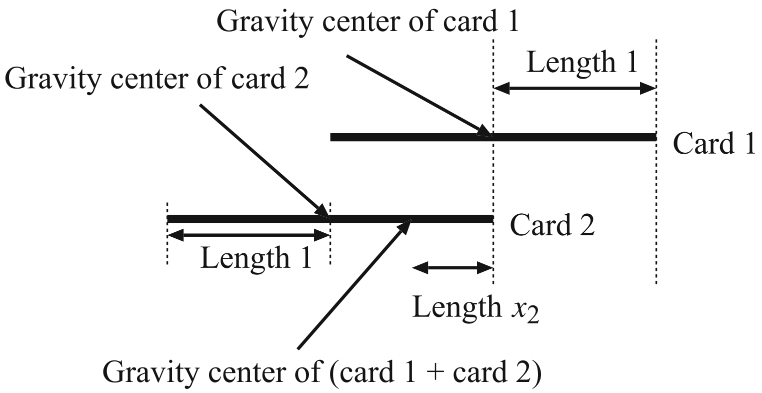


Fig. 1.8 The center of gravity of the top two cards 1 and 2

We have to put our heart into the next step for card 3. The center of gravity of the two cards 1 and 2 lies at $1/2$ from the right side of card 2. We consider the fulcrum that maintains a balance between the right edge of card 3, on top of which cards 1 and 2 are placed, and the center of card 3. Let x_3 be the distance of this balance point from the right side of card 3. Then, similarly, we have

$$(1 - x_3) \times 1 = x_3 \times 2.$$

Note that we have ×2 on the right hand because we have two cards on the right side. Solving this, we obtain $x_3 = 1/3$.

We can repeat this process; for the card k, we have $k - 1$ cards on it. We consider that the gravity center of these $k - 1$ cards occurs on the right edge of card k, and we balance it with the center of card k at the point distance x_k from the right side of card k. Then we have

$$(1 - x_k) \times 1 = x_k \times (k - 1),$$

and obtain $x_k = 1/k$.

Summarizing, when we stack n cards in this way, the distance of the right edge of card 1 from the edge of the desk is given by the following equation.

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$$

We can observe that $1 + 1/2 + 1/3 = 11/6 < 2 < 1 + 1/2 + 1/3 + 1/4 = 25/12$, which means that if we stack only four cards neatly, the first card protrudes beyond the edge of the desk. That is, beneath card 1, we have nothing but cards. Actually, it is not that easy; however, it is worth trying just for fun (Fig. 8.3).

Now we consider the last function $\sum_{i=1}^{n} \frac{1}{i}$. This is exactly the harmonic number $H(n)$. As we have already seen, the harmonic number $H(n)$ is almost equal to $\ln n$.

Moreover, the function $\ln n$ is a monotonically increasing function for n. Therefore, when you pile up many cards, you can position card 1 as far from the edge of the desk as you prefer. We must agree that this is counterintuitive. Then, how many cards do we have to pile up to make the card protrude farther?

Exercise 14 ☕☕ We have already observed that we need four cards to obtain the distance 2, which is the width of one card. Then how many cards do we need to increase this distance to 4, which is the width of two cards. How many cards do we have to prepare to increase this distance to 10 for five cards?

Solving Exercise 14, we realize that it is not as easy to achieve a large distance. That is, to increase a logarithmic function $f(n)$ to become large, we have to substitute a very large value into n. This is essentially the same as the exponential explosion. As an exponential function $g(n)$ increases quite rapidly as n increases, to increase the size of a logarithmic function $f(n)$, we have to increase the value of n considerably.

Have curiousness!

Theoretically, we can position the first card as far as you prefer by using the stacking method described in the text. However, if you prefer to obtain a large distance, we need a very large number of cards, as we have already seen. In fact, the above discussion with the result $H(n)$ is itself a well-known classic result in the society of recreational mathematics and puzzles. Then, can we reduce the number of cards to obtain some distance? In other words, is the above method the optimal way to ensure the card at the top protrudes farther? When you consider the method locally, as you put every card on the edge of balance in each step, at a glance it seems impossible to improve.

Surprisingly, however, in 2006, it was shown that this method is **not** optimal, and we can reduce the number of cards in general![a] That is, we have algorithms that are more efficient for stacking cards. Honestly, as I never imagined that this method might be improved, I was very surprised when I heard the presentation about the improved algorithm. Aside from the details of the algorithm, I had to take my hat off to their idea that the classic method might not be optimal.

[a] Mike Paterson and Uri Zwick, "Overhang," *Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm*, pp. 231–240, ACM, 2006.

1.6 Graph

Graph theory is a research area in modern mathematics. It is relatively new topic in the long history of mathematics, and it currently continues to be quite an active research area. However, the notion of a "graph" itself is quite a simple model. We have a point known as a **vertex**, and two vertices may join by a line known as an

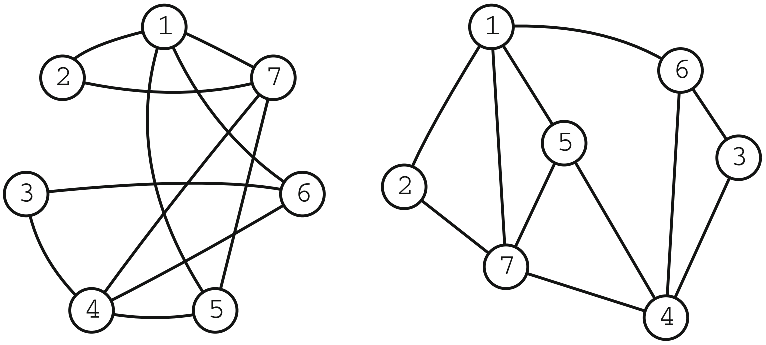


Fig. 1.9 Example of a graph

edge. Two vertices joined by an edge are said to be **adjacent**. A **graph** consists of some vertices joined by some edges. Some reader may be thinking “that’s it?”, but that’s it. A graph is a simple model to represent the connections between some objects. For example, on the World Wide Web, each Web page is represented by a vertex, and each hyperlink corresponds to an edge. In this example, we note that each hyperlink has its direction. Even if a page A refers to another page B by hyperlink, the page B does not necessarily indicate the page A by another hyperlink. Such a graph with directed edges is known as a **directed graph**, and if edges have no direction, the graph is known as an **undirected graph**. In this book, we assume that an edge joins two different vertices, and for each pair of vertices, only one edge is allowed between them. (Such a graph is referred to as a **simple graph** in technical terms in graph theory.)

In this book, we also suppose that the vertices in a graph are uniquely numbered by consecutive positive integers. That is, vertices in a graph with n vertices can be identified by $1, 2, \ldots, n$. In an undirected graph, each edge can be represented by a set of two vertices. That is, if vertices 1 and 50 are joined by an edge, it is represented by $\{1, 50\}$. In a directed graph, because the edge has a direction, we represent it by an ordered pair; hence, the edge is represented by $(1, 50)$. We note that {} represents a set and () represents an ordered pair. That is, $\{1, 50\}$ and $\{50, 1\}$ both indicate the same (unique) edge joining two vertices 1 and 50, whereas $(1, 50)$ and $(50, 1)$ are different edges (from 1 to 50 and from 50 to 1) on a directed graph. This may confuse some readers; however, this subtle difference is not such a serious issue in the description of an algorithm.

Exercise 15 In Fig. 1.9, there are two graphs. Enumerate all edges in each graph and compare these two graphs.

A graph is a very simple but strong model and has many interesting properties. Here we introduce three theorems that show useful properties for the analyses of algorithms, with two proofs for them.

The first theorem is about the degrees of an undirected graph. The **degree** $d(i)$ of a vertex i in an undirected graph is the number of vertices that are adjacent to vertex

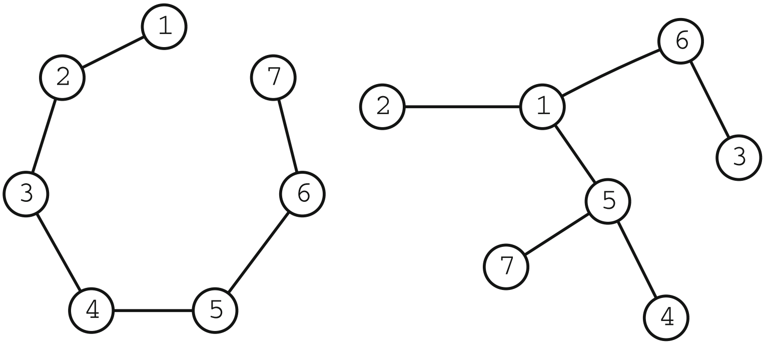


Fig. 1.10 An example of a tree

i. In other words, $d(i)$ is the number of edges joined to the vertex i. For the degrees of a graph, the following theorem is well known.

Theorem 3 *In an undirected graph, the summation of the degrees of the vertices is equal to twice the number of edges.*

Exercise 16 ☕ Compare the summation of the degrees of the vertices in the graph in Fig. 1.9 with the number of edges.

Theorem 3 seems to be strange, but can be proved elegantly by changing the viewpoint.

Proof Observe the operation "add $d(i)$ for each vertex i" from the viewpoint of an edge. Each edge $\{i, j\}$ is counted from vertex i and vertex j. That is, each edge is counted twice. Therefore, in total, the summation becomes twice the number of edges. □

Exercise 17 ☕☕ For a vertex i in a directed graph, the number of edges from the vertex i is referred to as **out-degree**, and the number of edges to the vertex i is referred to as **in-degree**. Then, develop an analogy of Theorem 3 for directed graphs.

Next, we introduce a special class of graphs known as **trees**. On an undirected graph, if any pair of vertices is joined by some sequence of edges, this graph is considered to be **connected**. For a connected graph, if any pair of vertices has a unique route between them, this graph termed a **tree**. Intuitively, a tree has no cycle; otherwise, we would have some pair of vertices with two or more routes between them. Formally, a graph is a tree if and only if it is connected and acyclic. We here show two examples of trees in Fig. 1.10. There is an interesting theorem for trees.

Theorem 4 *Any tree with n vertices has exactly $n - 1$ edges. Oppositely, any connected graph with $n - 1$ edges is a tree.*

There are exponentially many trees with n vertices. Nevertheless, every tree has the same number of edges if the number of vertices is fixed. It is an impressive fact. In fact, two trees in Fig. 1.10 have seven vertices with six edges. From the viewpoint of algorithm design, as we will see, the intuition that "any tree with n vertices has almost n edges" is useful. The proof of Theorem 4 is not simple, and we omit from this book. However, this theorem is one of the basic and important theorems in graph theory, and it is easy to find the proof in standard textbooks on graph theory.

Each vertex of degree 1 in a tree is known as a **leaf**. The following theorem is simple and useful.

Theorem 5 *Every tree with at least two vertices has a leaf.*

Proof It is trivial that a tree with two vertices consists of two leaves. We consider any tree T that has three or more vertices. Suppose that this T has no leaves. Because T is connected, every vertex has a degree of at least 2. Therefore, by Theorem 3, for m number of edges, we have $2m = \sum_{\text{each vertex } v} d(v) \geq 2 \times n = 2n$, which implies $m \geq n$. However, this contradicts the claim $m = n - 1$ in Theorem 4. Therefore, any tree has a vertex of degree one. □

Examining the two trees in Fig. 1.10, the left tree has two leaves (vertices 1 and 7), and the right tree has four leaves (vertices 2, 3, 4, and 7). In fact, Theorem 5 can be strengthened to "every tree with at least two vertices has at least two leaves." However, in this book, we only need the fact that every nontrivial tree has a leaf.

Theorem 5 is useful when we design an algorithm that functions on a tree structure. Suppose we would like to solve some problem P defined on a tree T. Then, by Theorem 5, T has to have a leaf. Thus, we first solve P on the leaf vertex. Then, we remove the useless (or processed) vertex with its associated edge. This leaf is of degree one; hence, the tree is still connected after removing the leaf. Moreover, the condition in Theorem 4 still holds because we have removed one vertex and one edge. That is, after removing the leaf the graph continues to be a tree, we again have another unprocessed leaf. In this way, we can repeatedly apply the same algorithm to each leaf, reducing the graph one by one, and finally, we have a trivial graph with one vertex. Even problems that are difficult to solve on a general graph, can sometimes be solved efficiently on a tree in this way.

1.6.1 Representations of a Graph

When we process a graph by computer, we have to represent the graph in memory in some ways. There are two typical approaches for representing a general graph. Mainly, the first approach is used when the algorithm is concerned with the neighbors of each specified vertex, and the second approach is used when the algorithm processes the whole graph structure at once. Both ways have their advantages; thus, we choose one of them (or the other specified way) according to the main operations in the algorithm.

Representation by adjacency set. The vertices adjacent to a vertex i are represented by an adjacency set. That is, the neighbor set $N(i)$ for a vertex i in an undirected graph is defined by

$$N(i) = \{j \mid \text{ there is an edge } \{i, j\} \text{ in the graph}\},$$

and the corresponding set $N(i)$ for a vertex i in a directed graph is defined by

$$N(i) = \{j \mid \text{ there is an edge } (i, j) \text{ in the graph}\}.$$

That is, in each case, $N(i)$ denotes the set of vertices that can be reached from vertex i in one step.

This relationship can be represented by a two-dimensional array. That is, we prepare a sufficiently large two-dimensional array $A[i, j]$, and each element in the neighbor set $N(i)$ of the vertex i is stored in $A[i, 1], A[i, 2], \ldots$. In our notation, the naming of vertices starts from 1. Therefore, we can specify that $A[i, j] = 0$ means "empty." Using this delimiter, we can determine the end of data for each $N(i)$. For example, when the neighboring set $N(1)$ of the vertex 1 is $N(1) = \{2, 3, 5, 10\}$, they are represented by $A[1, 1] = 2$, $A[1, 2] = 3$, $A[1, 3] = 5$, $A[1, 4] = 10$, and $A[1, 5] = 0$. This representation is useful when an algorithm examines "the entire neighbors j for each vertex i." More precisely, it is useful to search for some property in a given graph.

Example 3 As an example, let us construct the representation of the graph given in Fig. 1.9 by the adjacency set. According to the answer to Exercise 15, the set of edges is $\{\{1, 2\}, \{1, 5\}, \{1, 6\}, \{1, 7\}, \{2, 7\}, \{3, 4\}, \{3, 6\}, \{4, 5\}, \{4, 6\}, \{4, 7\}, \{5, 7\}\}$. We represent it in an array $A[]$. In the array $A[i, j]$, $A[i, *]$ consists of the vertices adjacent to i. Note that $A[i, *]$ does not contain i itself. Moreover, as a delimiter, we have to pack 0 at the tail of $A[i, j]$ Therefore, we need six entries with one delimiter in $A[i, *]$ in the worst case. In this example, we prepare $A[1, 1]$ to $A[7, 7]$. (In this example, in fact, $A[i, 7]$ is never used for each i since we have no vertex of degree 6.) The edges including 1 are $\{1, 2\}$, $\{1, 5\}$, $\{1, 6\}$, and $\{1, 7\}$. Therefore, we have $A[1, 1] = 2$, $A[1, 2] = 5$, $A[1, 3] = 6$, $A[1, 4] = 7$, and $A[1, 5] = 0$. Similarly, vertex 2 appears in $\{1, 2\}$ and $\{2, 7\}$ (do not forget $\{1, 2\}$). Thus, we have $A[2, 1] = 1$, $A[2, 2] = 7$, and $A[2, 3] = 0$. In this way, we obtain the array $A[]$ as follows

$$A = \begin{pmatrix} 2 & 5 & 6 & 7 & 0 & 0 & 0 \\ 1 & 7 & 0 & 0 & 0 & 0 & 0 \\ 4 & 6 & 0 & 0 & 0 & 0 & 0 \\ 3 & 5 & 6 & 7 & 0 & 0 & 0 \\ 1 & 4 & 7 & 0 & 0 & 0 & 0 \\ 1 & 3 & 4 & 0 & 0 & 0 & 0 \\ 1 & 2 & 4 & 5 & 0 & 0 & 0 \end{pmatrix}$$

In this representation, the vertices in a row are in increasing order. However, of course, we have other equivalent representations. For example, the following array $A'[]$ is also a valid representation of the same adjacent set.

$$A' = \begin{pmatrix} 2\ 7\ 6\ 5\ 0\ 0\ 0 \\ 7\ 1\ 0\ 0\ 0\ 0\ 0 \\ 4\ 6\ 0\ 0\ 0\ 0\ 0 \\ 5\ 6\ 3\ 7\ 0\ 0\ 0 \\ 7\ 4\ 1\ 0\ 0\ 0\ 0 \\ 3\ 4\ 1\ 0\ 0\ 0\ 0 \\ 5\ 4\ 2\ 1\ 0\ 0\ 0 \end{pmatrix}$$

Some readers may be concerned about the latter part of arrays $A[i, j]$ and $A'[i, j]$ for a large j, which is wasted. This is not problematic if the degrees of the vertices are almost equal; instead, there would be reason for concern if they were extremely varied such as the graph representing Web pages. In such a case, the use of pointers (which is beyond the scope of this book) instead of an array, would enable us to construct a more efficient data structure.

Representation by adjacency array. In an adjacency array representation, we again use a two-dimensional array $A[i, j]$; however, element $[i, j]$ directly indicates edge (i, j) or $\{i, j\}$. That is, if $A[i, j] = 0$ we have no edge between two vertices i and j, and $A[i, j] = 1$ means there is an edge joining the vertices i and j. This representation is not efficient for a large graph with a few edges from the viewpoint of memory usage. On the other hand, some algorithms run fast because they are able to determine whether the graph contains the edge $\{i, j\}$ in one step. Moreover, this representation is extensible to graphs with weighted edges; in this case, $A[i, j]$ provides the weight of the edge $\{i, j\}$. When we consider an undirected graph, we have $A[i, j] = A[j, i]$. Usually, the diagonal elements $A[i, i]$ are set to be 0.

Example 4 Here we again consider the graph in Fig. 1.9 and represent it by an adjacency array. This time, we write 1 if the graph has the corresponding edge, and 0 if not. For example, for the edge {1, 2}, we write 1s for both of the (1, 2) element and the (2, 1) element in the array. However, for the pair {3, 5}, we have no edge between vertices 3 and 5; hence, we write 0s at both for the (3, 5) and (5, 3) elements. We also define (i, i) as 0 for every i, then the resulting adjacency array $A''[]$ for the edge set {{1, 2}, {1, 5}, {1, 6}, {1, 7}, {2, 7}, {3, 4}, {3, 6}, {4, 5}, {4, 6}, {4, 7}, {5, 7}} is represented as follows:

$$A'' = \begin{pmatrix} 0\ 1\ 0\ 0\ 1\ 1\ 1 \\ 1\ 0\ 0\ 0\ 0\ 0\ 1 \\ 0\ 0\ 0\ 1\ 0\ 1\ 0 \\ 0\ 0\ 1\ 0\ 1\ 1\ 1 \\ 1\ 0\ 0\ 1\ 0\ 0\ 1 \\ 1\ 0\ 1\ 1\ 0\ 0\ 0 \\ 1\ 1\ 0\ 1\ 1\ 0\ 0 \end{pmatrix}$$

Note that we have a unique A'' unlike an adjacency set because the indices of the vertices are fixed. In an undirected graph, we always have $A''[i, j] = A''[j, i]$, that is, A'' is a symmetric matrix. In this representation, each entry is either 0 or 1.

Link structure of WWW

It is known that the graph representing the link structure of the WWW (World Wide Web) has a property referred to as **scale free**. In a scale-free graph, most vertices have quite small sets of neighbors, whereas few vertices have quite large sets of neighbors. Considering real WWW pages, many WWW pages contain a few links that are maintained by individual people; however, as for a search engine, few WWW pages contain a large number of links. In this book, we introduce two different representations for graphs. Unfortunately, none of them would be able to efficiently (from the viewpoint of memory usage) maintain a huge scale-free graph. This would require a more sophisticated data structure.

Chapter 2
Recursive Call

Abstract Recursive call may appear difficult when you initially approach, but it cannot be overlooked when it comes to studying algorithms. It bears a close connection with mathematical induction, and those who once had a frustrating experience with mathematical induction in the past may feel intimidated, but there is nothing to fear. It is my opinion that anyone who understands grammatically correct your native language and knows how to count natural numbers can master the use of recursive calls. In this chapter, we will attempt to understand recursive calls and their correct usage by considering two themes: the "Tower of Hanoi " and "Fibonacci numbers.".

What you will learn:

- Recursive call
- Tower of Hanoi
- Fibonacci numbers
- Divide-and-conquer
- Dynamic programming

2.1 Tower of Hanoi

Under a dome that marks the center of the world in a great temple in Benares, India, there are three rods. In the beginning of the world, God piled 64 discs of pure gold onto one of the rods in descending order of size from bottom to top. Monks spend days and nights transferring discs from rod number 1 to rod number 3 following a certain rule. The world is supposed to end as soon as the monks finish moving all the discs. The rules for moving the discs around are quite simple. In the first place, only one disc can be moved at a time. Moreover, a disc cannot be placed upon a smaller one. How can the monks move the disks in an efficient manner? Moreover, how many times would the monks have to move the discs around to finish moving all discs?

This puzzle is known as the Tower of Hanoi. It was originally proposed in 1883 by French mathematician E. Lucas. An example of the Tower of Hanoi available in

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_2

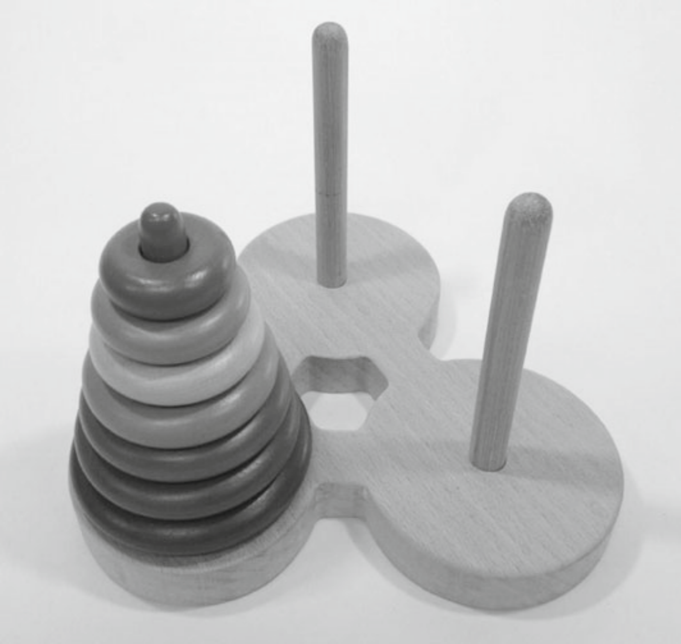


Fig. 2.1 Tower of Hanoi with 7 discs

the market is shown in Fig. 2.1. This example has seven discs. In this chapter, we will consider algorithms to move discs and the number of moves $H(n)$, where n is the number of discs.

We simplify descriptions by denoting the transfer of disc i from rod j to rod k as $(i; j \to k)$. Consider that the discs are enumerated in ascending order of size, starting from the smallest one. Let us start with a simple case, which is a convenient way to familiarize ourselves with the operation. The simplest case is of course for $n = 1$. If there is only one disc, it is straightforward to perform operation $(1; 1 \to 3)$, which consists of simply moving the disc from rod 1 to rod 3; thus, $H(1) = 1$. Next, let us consider the case $n = 2$. In the beginning, the only operations that are possible are $(1; 1 \to 3)$ or $(1; 1 \to 2)$. As we eventually want to execute $(2; 1 \to 3)$, we choose the latter alternative. With a little thought, it is not difficult to conclude that the best operations are $(1; 1 \to 2)$, $(2; 1 \to 3)$, and $(1; 2 \to 3)$. Therefore, $H(2) = 3$. At this point, we start getting confused for values of $n = 3$ or greater. After some thought, we notice the following three points:

- We necessarily have to perform operation $(3; 1 \to 3)$.
- Operation $(3; 1 \to 3)$ requires all discs except 3 to be moved from rod 1 to rod 2.
- After performing $(3; 1 \to 3)$, all discs already moved to rod 2 must be transferred to rod 3.

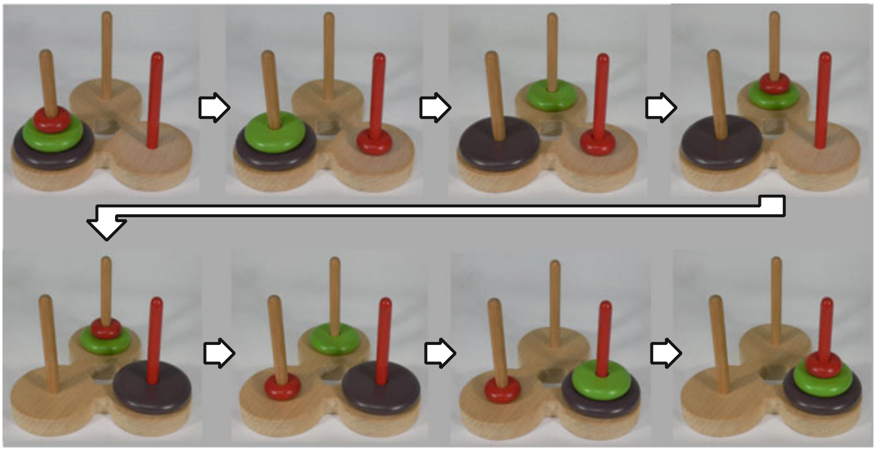


Fig. 2.2 View of the transitions of the Tower of Hanoi for $n = 3$

Regarding the "evacuation" operation of "moving all discs from rod 1 to rod 2," we can reuse the procedure previously used for $n = 2$. In addition, the same procedure can be used for "moving all discs from rod 2 to rod 3." In other words, the following procedure will work:

- Move discs 1, 2 from rod 1 to rod 2: $(1; 1 \to 3)$, $(2; 1 \to 2)$, $(1; 3 \to 2)$
- Move disc 3 to the target: $(3; 1 \to 3)$
- Move discs 1, 2 from rod 2 to rod 3: $(1; 2 \to 1)$, $(2; 2 \to 3)$, $(1; 1 \to 3)$

Therefore, we have $H(3) = 7$ (Fig. 2.2).

Here, let us further look into the rationale behind the $n = 3$ case. We can see that it can be generalized to the general case of moving k discs, i.e.,

- It is always true that we have to execute operation $(k; 1 \to 3)$.
- Executing operation $(k; 1 \to 3)$ requires all $k - 1$ discs to be moved from disc 1 to disc 2.
- After executing operation $(k; 1 \to 3)$, all discs sent to rod 2 must be moved to rod 3.

An important point here is the hypothesis that "the method for moving $k - 1$ discs is already known." In other words, if we **hypothesize** that the problem of moving $k - 1$ discs is already solved, it is possible to move k discs. (By contrast, as it is not possible to move k discs without moving $k - 1$ discs, this is also a necessary condition.) This rationale is the core of a recursive call, that is, it is a technique that can be used when the two following conditions are met:

(1) Upon solving a given problem, the algorithm for solving its own partial problem can be reused;
(2) The solution to a sufficiently small partial problem is already known. In the case of the Tower of Hanoi in question, the solution to the $k = 1$ case is trivial.

If the above is written as an algorithm, the following algorithm for solving the Tower of Hanoi problem takes shape. By calling this algorithm as Hanoi(64, 1, 3), in principle the steps for moving the discs will be output.

```
Algorithm 11: Algorithm Hanoi($n$, $i$, $j$) for solving the Tower of Hanoi problem
  Input  : number of discs $n$, rods $i$ and $j$
  Output: procedure for moving discs around
if $n = 1$ then
    output "move disc 1 from rod $i$ to rod $j$"
else
    let $k$ be the other rod than $i$, $j$;
    Hanoi($n - 1$, $i$, $k$);
    output "move disc $n$ from rod $i$ to $j$";
    Hanoi($n - 1$, $k$, $j$);
end
```

We can see that the above can be written in a straightforward manner, with no unnecessary moves. Here, a reader who is not familiarized with recursive calls may feel somewhat fooled. In particular, the behavior of variable n and that of variables i, j may seem a little odd. At first sight, variable n seems to have multiple meanings. For example, when Hanoi(3, 1, 2) is executed, $n = 3$ in the beginning, but as the execution of Hanoi($n - 1$, i, k) proceeds, we require $n = 2$ inside the call. Further into the process, a function call takes place with $n = 1$, and the sequence of recursive calls stops. An example of a Hanoi(3, 1, 2) execution is shown in Fig. 2.3. Within this sequence of calls, "variable n" represents different roles under the same name. This may seem strange to the reader. This is useful for those who are familiarized, but may confuse those who are not. Readers who are not particularly interested may proceed, but unconvinced ones are invited to refer to Sect. 2.1.2, "Mechanism of Recursive Calls."

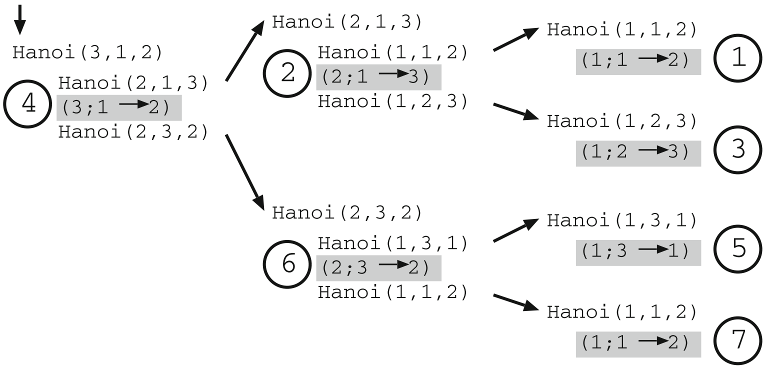


Fig. 2.3 Execution flow of Hanoi(3, 1, 2). The parts displayed against a gray background are outputs, and the circled numbers denote the order of output

Exercise 18 ☕☕☕ Write Hanoi$(4, 1, 3)$ down by hand. What is the value of $H(4)$? Predict the value of the general case $H(n)$.

2.1.1 Analysis of the Tower of Hanoi

If we actually compute $H(4)$ of Exercise 18, we are surprised by the unexpectedly large number obtained. In the present section, we will consider the value of $H(n)$ in the general case. In terms of expressions, the analysis above can be formulated as the following two expressions:

$$\begin{aligned} &H(1) = 1 \\ &H(n) = H(n-1) + 1 + H(n-1) = 2H(n-1) + 1 \qquad (for\ n > 1) \end{aligned}$$

Recursive calls are intimately related to mathematical induction. This is not surprising, considering that they almost represent the two sides of the same coin. However, we will try the easiest way. First, let us add 1 to both sides of the equation $H(n) = 2H(n-1) + 1$, obtaining $H(n) + 1 = 2H(n-1) + 2 = 2(H(n-1) + 1)$. Denoting $H'(n) = H(n) + 1$, we obtain $H'(1) = H(1) + 1 = 2$, and we can further write $H'(n) = 2H'(n-1)$. By further expanding the right-hand side for a general n, we obtain:

$$H'(n) = 2H'(n-1) = 2(2(H'(n-2))) = \cdots = 2^{n-1}H'(1) = 2^n$$

returning to the original equation, we have:

$$H(n) = H'(n) - 1 = 2^n - 1.$$

In other words, roughly speaking, adding one disc increases the number of moves by approximately two times. As we have seen in Sect. 1.5, this is an exponential function and the number is known to grow explosively.

Exercise 19 ☕☕☕ How many moves $H(n)$ are required to move 64 discs? Considering that it takes 1 second to move a single disc, what is the approximate time left before the end of the world?

2.1.2 Recurrent Call Mechanism

If we think about it, we realize that the mechanism of recurrent calls is a strange one. Why do we attach different meanings to variables with the same name and manage to control the process without confusion? It is worth taking an alternative approach to carefully consider this. For example, let us look at the following Duplicate algorithm, which results from a slight modification of the Tower of Hanoi algorithm:

```
Algorithm 12: Duplicate($n$)
  Input  : Natural number $n$
  Output: if $n > 1$ then display $n$ twice each time, with a recurrent call in between
if $n = 1$ then
    output the value of $n$
else
    output the value of $n$;
    Duplicate($n - 1$);
    output the value of $n$;
    Duplicate($n - 1$);
end
```

The algorithm has an impressive name, but is quite simple. It works as follows when actually executed. Because it basically repeats the action "output n and attach the output up to $n - 1$" twice, it is easy to consider the sequences that are output by starting with a small value of n:

- Duplicate(1) produces 1 as the output.
- Duplicate(2) produces 2121 as the output.
- Duplicate(3) produces 3212132121 as the output.
- Duplicate(4) produces 4321213212143212132121 as the output.

The sequences of numbers that are output seem to have a meaning, but in fact, they do not. It is worth noting "variable n" here. It is nested upon each recursive call and controlled in separate memory areas under the same name n. Let us present an aspect of this recursive call in the form of a diagram (Fig. 2.4, where the second half is omitted). Roughly speaking, the vertical axis shows the process flow, which represents time evolution. The issue is the horizontal axis. The horizontal axis shows the "level" of recursive calls. The level becomes deeper and deeper as we proceed rightwards. When the subroutine named Duplicate is called with an argument n as in Duplicate(n), the computer first allocates local memory space for argument n and stores this value. When the recursive call ends, this memory area is released/cleared by deleting its contents.

To further clarify this problem, let us denote the memory area allocated upon the i-th call to Duplicate(n) as n_i. When Duplicate(4) is called, memory for n_1 is first allocated, and when Duplicate(3) is executed in between, memory n_2 is allocated, and so on. When n_i is no longer necessary, it is released. Figure 2.4 also shows aspects of memory allocation and clearance. If n_i is correctly managed, recursive calls are executed correctly. On the other hand, if any kind of confusion occurs, it is not possible to correctly identify the variables. Here we denote the event "allocated variable n_i" as $\underset{i}{[}$, and the event "released variable n_i" as $\underset{i}{]}$. Using this notation, if we place the events involving variable allocation and release in Fig. 2.4 in chronological order, we obtain the following sequence of symbols. The figure shows only the first

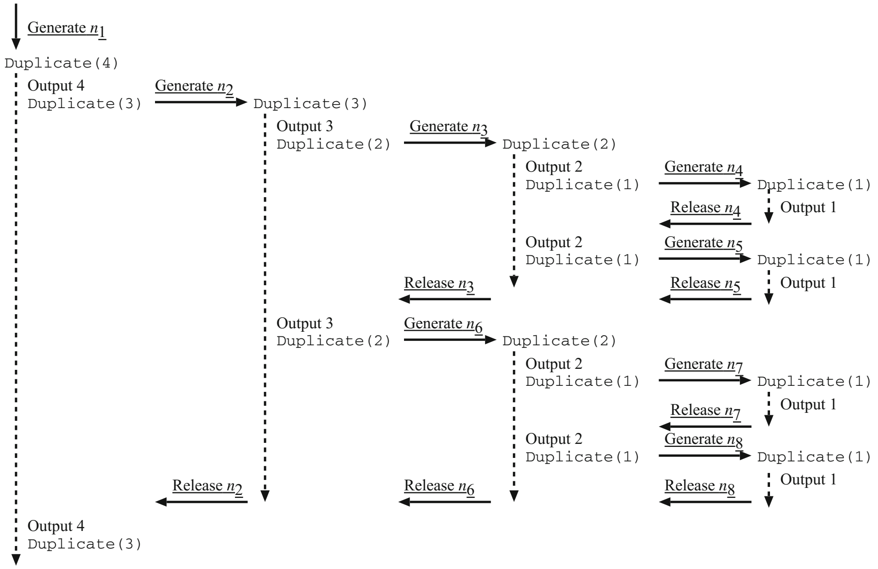


Fig. 2.4 Flow of Duplicate(4). (Here the first half with output 432121321214 is shown. The second half with output 3212132121 is omitted)

half, but the second half has a similar shape. The following sequence of symbols also includes the second half.

$$\underset{1}{[}\,\underset{2}{[}\,\underset{3}{[}\,\underset{4}{[}\,\underset{4}{]}\,\underset{5}{[}\,\underset{5}{]}\,\underset{3}{]}\,\underset{6}{[}\,\underset{7}{[}\,\underset{7}{]}\,\underset{8}{[}\,\underset{8}{]}\,\underset{6}{]}\,\underset{2}{]}\,\underset{9}{[}\,\underset{10}{[}\,\underset{11}{[}\,\underset{11}{]}\,\underset{12}{[}\,\underset{12}{]}\,\underset{10}{]}\,\underset{13}{[}\,\underset{14}{[}\,\underset{14}{]}\,\underset{15}{[}\,\underset{15}{]}\,\underset{13}{]}\,\underset{9}{]}\,\underset{1}{]}$$

If we look at it carefully, we realize that in fact the "$_i$" part of the notation **is not needed**. In other words, even without the index "$_i$", "[" and "]" are correctly related to each other. Furthermore, two differing indices may be an indication of nesting such as $\underset{i}{[}\,\underset{j}{[}\,\underset{j}{]}\,\underset{i}{]}$, or an independent case such as $\underset{i}{[}\,\underset{i}{]}\,\underset{j}{[}\,\underset{j}{]}$. An "entangled relation" such as $\underset{i}{[}\,\underset{j}{[}\,\underset{i}{]}\,\underset{j}{]}$ never occurs. In other words, variables n_i are released in the opposite order in which they are allocated. Does this sound familiar? Yes, that is the "**stack**." The variables of a recursive call can be managed using a stack. In concrete terms, whenever Duplicate(n) is called, a new memory location is allocated on the stack, and this location can be referenced when n needs to be accessed. Moreover, when calling Duplicate(n) has finished, the last element allocated to the stack can be accessed and deleted. Thus, the management of variables in a recursive call is exactly the same as for the stack structure.

When realizing/implementing a recursive call, it is inevitable to use the same variable name for different meanings. This "trick" is not restricted to recursive calls and can be used as a standard in several programming languages. These variables are called "**local variables**" and are in fact managed by means of the stack structure.

On the other hand, it is also useful to have variables that "always mean the same thing, anywhere." These are called "**global variables**." Managing "global variables" is not difficult. However, excessive use of global variables is not a smart practice for a programmer. The best practice is to divide the problem into local problems, and then encapsulate and solve them within their ranges.

2.2 Fibonacci Numbers

Fibonacci numbers denote a sequence of numbers that go under the name of Italian mathematician Leonardo Fibonacci.[1] More precisely, these numbers are known as Fibonacci sequence, and the numbers that belong to this are called Fibonacci numbers. The sequence actually has quite a simple definition, namely:

$$\begin{aligned} F(1) &= 1, \\ F(2) &= 1, \\ F(n) &= F(n-1) + F(n-2) \quad (\text{for}\, n > 2). \end{aligned}$$

In concrete terms, the sequence can be enumerated as $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$. The Fibonacci sequence $F(n)$ is known to attract widespread interest for frequently appearing as a numerical model when we try to build models representing natural phenomena, such as the arrangement of seeds in sunflowers, the number of rabbit siblings, and others. Let us deepen our understanding of recursive calls using this sequence. At the same time, let us discover the limitations of recursive calls and introduce further refinements.

2.2.1 *Computing Fibonacci Numbers $F(n)$ Arising from Recursive Calls*

There is one particular aspect that must be considered carefully when working with recursive calls. We have seen that, in recursive calls, the algorithm capable of solving its own partial problem is reused. It is worth noting to what extent this "partial problem" is necessary. In the Tower of Hanoi problem, we reused the algorithm for solving the partial problem of size $n - 1$ to solve a problem of size n. Considering the definition of Fibonacci numbers $F(n)$, the values of both $F(n-1)$ and $F(n-2)$ are required. From this fact, we learn that it does not suffice to show a clear solution for the small partial problem in $n = 1$, but the solutions for $n = 1$ and $n = 2$ are

[1] **Leonardo Fibonacci:1170?–1250?**:
Italian mathematician. His real name was Leonardo da Pisa, which means "Leonardo from Pisa." "Leonardo Fibonacci" means "Leonardo, son of Bonacci." In fact, he did not invent Fibonacci numbers himself. They borrowed his name due to the popularity gained after he mentioned them in his book "Liber Abaci" (book on abacus).

also needed. If we construct the algorithm according to its definition while paying attention to this point, we obtain the algorithm below. Using this algorithm, the n-th Fibonacci number $F(n)$ can be obtained by calling Fibr(n), which produces the returned value as its output.

Algorithm 13: recursive algorithm Fibr(n) to compute the n-th Fibonacci number $F(n)$

```
Input  : $n$
Output: the $n$-th Fibonacci number $F(n)$
if $n = 1$ then return 1;
if $n = 2$ then return 1;
return Fibr($n - 1$) + Fibr($n - 2$);
```

Exercise 20 ☕☕☕ Implement and execute the algorithm above. How will the program behave when n is made slightly larger (by a few tens)?

2.2.2 Execution Time of Fibr Based on Recursive Calls

When we implement and execute Fibr(n), execution clearly slows down for values around $n = 40$ and above. Why does this occur? Let us consider an execution time $t_F(n)$. When considering execution times, examining the recursive equation is inevitable. In concrete terms, we have:

- For $n = 1$: a constant time $t_F(1) = c_1$
- For $n = 2$: a constant time $t_F(2) = c_2$
- For $n > 2$: for a given constant c_3, $t_F(n) = t_F(n-1) + t_F(n-2) + c_3$

Here, the value of the constant itself has no meaning, and therefore, we can simply denote them as c. By adding c to both sides of the equation, for $n > 2$ we can write:

$$t_F(n) + c = (t_F(n-1) + c) + (t_F(n-2) + c)$$

In other words, if we introduce another function $t'_F(n) = t_F(n) + c$, we can write:

$$t'_F(n) = t'_F(n-1) + t'_F(n-2),$$

which is exactly equivalent to the definition of Fibonacci numbers. In other words, the execution time $t_F(n)$ of Fibr(n) can be considered proportional to the value of Fibonacci numbers. If we anticipate the contents of Sect. 2.2.4 below, it is known that Fibonacci numbers $F(n)$ will be $F(n) \sim \Theta(1.618^n)$. In other words, Fibonacci numbers constitute an exponential function. Therefore, if we compute Fibonacci numbers according to their definition, the computation time tends to increase exponentially, that is, if n becomes larger, the computation time of a naïve implementation is excessively large.

2.2.3 Fast Method for Computing Fibonacci Numbers

We learnt that, according to the definition, the time required to compute Fibonacci numbers increases exponentially. However, some readers may feel intrigued by that. For instance, to compute $F(9)$, only $F(8)$ to $F(1)$ suffice. First, let us solve the question why the time required for computing Fibonacci numbers increases exponentially. For example, $F(9)$ can be computed if $F(8)$ and $F(7)$ are available. Computing $F(8)$ only requires knowing $F(7)$ and $F(6)$, and computing $F(7)$ only requires $F(6)$ and $F(5)$, and so forth. This calling mechanism is illustrated in Fig. 2.5.

What we need to note here is "who is calling who." For instance, if we pay attention to $F(7)$, this value is called twice, that is, upon the computation of both $F(9)$ and $F(8)$. That is, $F(7)$ is nested twice to compute $F(9)$. Likewise, $F(6)$ is called from $F(8)$ and $F(7)$, and $F(7)$ is called twice. This structure becomes more evident as we approach the leaves of the tree structure. $F(2)$ and $F(3)$ appear in Fig. 2.5 remarkably often. Thus, the Fibr(n) algorithm based on recursive calls computes the same value over and over again in a wasteful fashion.

Once the problem becomes clear at this level, the solution naturally becomes evident. The first idea that pops up is to use arrays. Using an array, we can start computing from the smallest value and refer to this value when necessary. Concretely speaking, we can use the following algorithm:

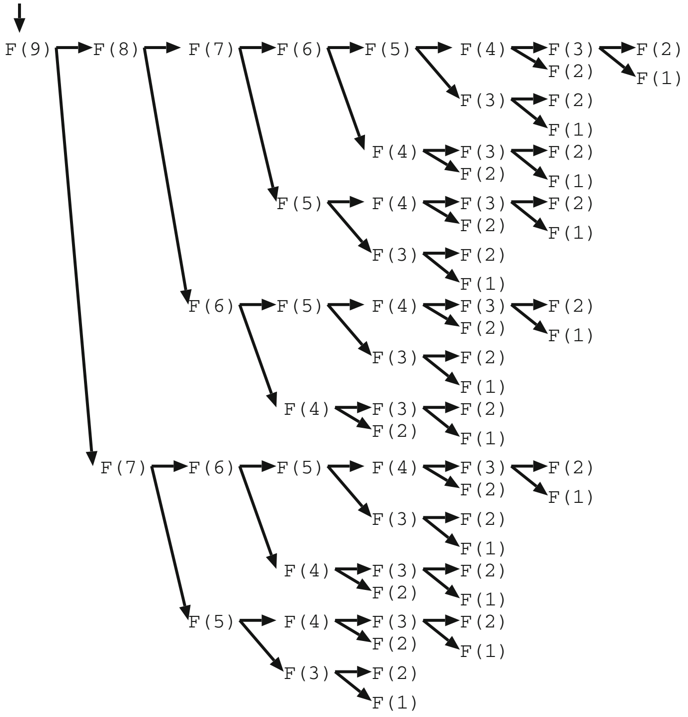


Fig. 2.5 Aspect of the computation of the Fibonacci number $F(9)$

Algorithm 14: Algorithm Fiba(n) to compute the n-th Fibonacci number $F(n)$ using array $Fa[]$

```
Input  : $n$
Output: $n$-th Fibonacci number $F(n)$
$Fa[1] \leftarrow 1$;
$Fa[2] \leftarrow 1$;
for $i \leftarrow 3, 4, \ldots, n$ do
    $Fa[i] \leftarrow Fa[i-1] + Fa[i-2]$;
end
output $Fa[n]$;
```

First, $F(1)$ and $F(2)$ are directly computed when $i = 1, 2$. Then, for $i > 2$ it is quite clear that for computing $Fa[i]$ we already have the correct values of $Fa[i-1]$ and $Fa[i-2]$. These two observations indicate to us that Fibonacci numbers can be correctly computed by this algorithm. Therefore, the computation time of the Fiba(n) algorithm is linear, that is, proportional to n, which means it is extremely fast.

Let us stop for a while to think ahead. For $i > 2$, what is necessary for computing $F(i)$ is $F(i-1)$ and $F(i-2)$, and elements further behind are not needed. In other words, provided that we keep in mind the last two elements, the array itself is unnecessary. With that perception in mind, we can write an even more efficient algorithm (from the viewpoint of memory space).

Algorithm 15: Algorithm Fib2(n) to compute the n-th Fibonacci number $F(n)$ without using arrays

```
Input  : $n$
Output: $n$-th Fibonacci number $F(n)$
if $n < 3$ then
    output "1";
else
    $Fa1 \leftarrow 1$ ;                                  /* Memorize $F(1)$ */
    $Fa2 \leftarrow 1$ ;                                  /* Memorize $F(2)$ */
    for $i \leftarrow 3, 4, \ldots, n$ do
        $Fa \leftarrow Fa2 + Fa1$ ; /* Compute/memorize $F(i) = F(i-1) + F(i-2)$ */
        $Fa1 \leftarrow Fa2$ ;                  /* Update $F(i-2)$ with $F(i-1)$ */
        $Fa2 \leftarrow Fa$ ;                     /* Update $F(i-1)$ with $F(i)$ */
    end
    output $Fa$;
end
```

Exercise 21 Implement and execute the algorithm above. Confirm that the answer is output immediately even for very large values of n.

2.2.4 Extremely Fast Method to Compute Fibonacci Numbers

Fibonacci numbers $F(n)$ have fascinated mathematicians since ancient times due to their intriguing and interesting properties. In this section, let us borrow some of the results of such mathematical research. In fact, it is known that the general term of Fibonacci numbers $F(n)$ can be expressed by the following expression:

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

It is fascinating that even though $\sqrt{5}$ appears several times in the equation, the result $F(n)$ is always a natural number for any input n that is a natural number. If we implement an algorithm that computes this equation, the computation time of a Fibonacci number $F(n)$ ends in a constant time. (Considering the errors involved in the computation of $\sqrt{5}$, there might be cases in which the algorithm that allows computation in linear time of the previous section is preferable.)

In the present Section, let us prove that the Fibonacci number $F(n)$ approximately follows $\Theta(1.618^n)$. First, consider the two terms $\left(\frac{1+\sqrt{5}}{2}\right)^n$ and $\left(\frac{1-\sqrt{5}}{2}\right)^n$. If we actually compute them, we obtain $\left(\frac{1+\sqrt{5}}{2}\right) = 1.61803\ldots$ and $\left(\frac{1-\sqrt{5}}{2}\right) = -0.61803\ldots$. Thus, the absolute value of the former is considerably larger than 1, whereas the absolute value of the latter is considerably smaller than 1. Therefore, as n increases the value of $\left(\frac{1+\sqrt{5}}{2}\right)^n$ increases very fast, whereas the value of $\left(\frac{1-\sqrt{5}}{2}\right)^n$ becomes small very quickly. Thus, the value of $F(n)$ is approximately $1.618^n/\sqrt{5}$, which is of the order $\Theta(1.618^n)$, i.e., an exponential function.

Exercise 22 ☕☕☕☕ Prove by means of mathematical induction that the general term equation for Fibonacci numbers $F(n)$ actually holds.

Exercise 23 ☕☕☕☕ Investigate the relation between Fibonacci numbers $F(n)$ and the golden ratio ϕ. The **golden ratio** ϕ is a constant defined as $\phi = \frac{1+\sqrt{5}}{2}$, and its value is approximately 1.618.

2.3 Divide-and-Conquer and Dynamic Programming

When tackling a large problem using a computer, approaching the problem directly as a whole often does not lead to good results. In such cases, we must first consider whether it is possible to split the large problem into small partial problems. In many cases, a large problem that cannot be tackled as it is can be split into sufficiently small partial problems that can be appropriately handled individually. Solving these individual partial problems and combining their solutions often permits the original large problem to be solved. This method is known as "**divide and conquer**." The idea is that a large target must be divided to allow for proper management. Political units such as countries, prefectures, cities, towns, and villages also constitute examples of this idea. The divide-and-conquer method can be considered a "top-down" approach.

The Tower of Hanoi problem was successfully solved by splitting the original problem consisting of moving n discs into the following partial problems:

- Partial problem where the number of discs is $n - 1$
- Partial problem of moving only 1 disc (the largest one)

This is an example of divide-and-conquer. Regarding the computation of Fibonacci numbers, using a recursive definition for computation can be considered a divide-and-conquer method. However, in the case of Fibonacci numbers, we found a computation method that is much more efficient than the divide-and-conquer method. The limitation of the "divide-and-conquer" method for Fibonacci numbers is that the solution to the problems that had already been solved is forgotten every time. This is a characteristic of top-down approaches that is difficult to avoid, a fatal limitation in the case of Fibonacci numbers. This condition resembles the situation of vertical administration systems where similar organizations exist in several places.

Dynamic programming is a method conceived to address such problems. In this method, minutely sliced problems are solved in a bottom-up approach. When optimal solutions to partial problems are found, such that no further improvement is possible, only the necessary information is retained and all the unnecessary information is forgotten. In the Fibonacci numbers example, an important point is that computations are carried out in a determined order, starting from the small values. Because only the values of the Fibonacci numbers immediately preceding $F(i - 1)$ and $F(i - 2)$ are needed, values further back can be forgotten. In other words, for Fibonacci number computation, it suffices to retain only the previous two values.

In the case of the Tower of Hanoi problem, because all discs must be moved every time, the problem does not fit well in the dynamic programming framework. For this reason, the solution is limited to the divide-and-conquer method.

Chapter 3
Algorithms for Searching and Sorting

Abstract Searching and sorting are two basic problems that contain many valuable elements for algorithms. They are still being investigated in many related areas. The series of books entitled "The Art of Computer Programming" by D. E. Knuth is said to be "the Bible" of algorithmic theory, and the third volume has the subtitle "Sorting and Searching." We will enjoy and understand these deep problems in this chapter.

What you will learn:

- Linear search
- Sentinel
- Block search
- Binary search
- Hash
- Bubble sort
- Merge sort
- Quick sort
- Lower bound for sorting
- Bucket sort
- Spaghetti sort

Donald Ervin Knuth (1938–):
He is a professor emeritus of computer science at Stanford University. The series of books mentioned in the text is called "TAOCP." It was planned to publish the series in seven volumes, starting in the 1960s. Now, some preliminary drafts of the fourth volume are public on the Web. It is surprising that, to write this series, Knuth first started with designing and programming the well-known TEX system for typesetting, and the METAFONT system for designing typesetting fonts. See Chap. 7 for more details.

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_3

3.1 Searching

We first consider the search problem that is formalized as

Searching:

Input: An array $a[1], a[2], \ldots, a[n]$ and data x.
Output: The index i such that $a[i] = x$.

There are some variants of the problem according to the conditions, e.g.,

- It is guaranteed that there exists an i such that $a[i] = x$ for every x or not.
- The array $a[]$ may contain the same data two or more times, that is, $a[i] = a[j]$ for some $i \neq j$ or not.
- The elements in $a[]$ are preprocessed following some rule or not. One typical assumption is that $a[1] \leq a[2] \leq a[3] \leq \cdots \leq a[n]$. In real data, such as dictionaries and address books, in general, massive data are organized in some order.

For the moment, we have no special assumption on the ordering of data.

3.1.1 Linear Search and Its Running Time

A natural straightforward search method is the linear search; it checks the data from $a[1]$ to the last data item or until finding x. The algorithm's description is simple:

Algorithm 16: LinearSearch(a, x)

```
  Input  : An array $a[]$ and data $x$
  Output: The index $i$ if there is an $i$ with $a[i] = x$; otherwise, "Not found."
1 $i \leftarrow 1$;
2 while $i \leq n$ and $a[i] \neq x$ do
3   | $i \leftarrow i + 1$;
4 end
5 if $i \leq n$ then
6   | output $i$;
7 else
8   | output "Not found";
9 end
```

We now explain the while statement, which appears here for the first time. The while statement is one of the control statements. It means "repeating some statements while given condition." In this linear search, after initializing the variable i by 1 at line 1, the statement "$i \leftarrow i + 1$" is repeated while the condition "$i \leq n$ and $a[i] \neq x$" holds. In other words, when we have "$i > n$ or $a[i] = x$," the loop halts and the program goes to the next step. If we look at the manner in which the while statement works, the correctness of the algorithm is easy to see. The running time is $\Theta(n)$, which is also easily seen. When there is no i such that $a[i] = x$, the algorithm halts in $\Theta(n)$ time, and if there is an i with $a[i] = x$, the algorithm finds it after checking $n/2$ elements in the average case. (Of course, this "average case" depends on the distribution of the data.)

Sentinel. We now introduce a useful technique for linear searching. This technique, which is a type of programming technique rather than an algorithmic one, is called *sentinel*. One may feel that the linear search algorithm described above is a bit more complicated than necessary. The reason is that, in the while statement, we simultaneously check two essentially different conditions:

- Whether the array $a[i]$ ends or not, and
- Whether i satisfies $a[i] = x$ or not.

Moreover, the algorithm checks whether $i \leq n$ in line 5, but this check has already been performed once in the last while statement. This seems to be redundant. Although the basic idea of the algorithm is simple, and even though the idea of the underlying algorithm is also simple, when we consider the boundary conditions in the implementation, it becomes complicated. This is often the case. How can we resolve it?

One of the techniques for resolving this complexity is called **sentinel**. This is a neat idea for unifying these two conditions "check the end of the array $a[]$" and "check the index i with $a[i] = x$." To apply this idea, we first prepare an additional element $a[n+1]$ and initialize it by x. Then, of course, there does exist an index i with $a[i] = x$ in the interval $1 \leq i \leq n+1$. In this case, we find i with $a[i] = x$ in the original array if and only if $1 \leq i \leq n$. Therefore, the algorithm can decide to output i only if $i < n+1$. This element $a[i+1]$ is called the *sentinel*. Using this technique, the algorithm can be rewritten in a smarter style.

```
Algorithm 17: LinearSearchSentinel($a$, $x$)
  Input  : An array $a[]$ and data $x$
  Output: The index $i$ if there is an $i$ with $a[i] = x$; otherwise, "Not found."
$i \leftarrow 1$;
$a[n+1] \leftarrow x$; ;                              /* This is the sentinel. */
while $a[i] \neq x$ do
    $i \leftarrow i + 1$;
end
if $i < n + 1$ then
    output $i$;
else
    output "Not found";
end
```

It may seem a trifling improvement, but the conditions in while are checked every time, and this 50% reduction is unexpectedly efficient. Although it can be applied only if we can use an additional element in the array, it is a noteworthy technique.

3.1.2 Searching in Pre-sorted Data

In the linear search method, we assume nothing about the ordering of the data. Thus, we cannot guess where x can be in the array at all. In this case, we have a few tricks for improving the linear search method, but it seems to be impossible to achieve a better running time than $\Theta(n)$. Now, we turn to the significant case where the data are ordered. That is, our next problem is the following.

Searching in ordered data:

Input: An array $a[1], a[2], \ldots, a[n]$ and data x. We know that $a[1] \le a[2] \le \cdots \le a[n]$.
Output: The index i such that $a[i] = x$.

We first mention that this assumption is reasonable. In dictionaries and address books, where one searches real data, the data are usually already ordered to make searching easier. In the case of repeated searching, such a pre-ordering is the standard technique for making the search efficient. (Imagine what would happen if you had to linear search the Oxford English Dictionary if it was not pre-ordered!) Of course, we need to consider carefully whether or not this pretreatment will pay off. At any rate, we introduce fast algorithms that use the assumption that the given data is pre-ordered.

When you search a thick dictionary, how do you find the entry? Maybe you first open the page "around there," narrow the search area, and look at the entries one by one. In this book, we call this strategy **block search** and investigate its properties. That is, a block search consists of three phases: the algorithm first introduces a *block*, which is a bunch of data, finds the block that contains your data, and checks each data item in the block one by one. (Actually, this "block search" is not used in a computer program in this form. The reason will appear later. This is why this method has no common name. That is, this "block search" is available only in this book to explain the other better algorithm.)

3.1.3 Block Search and Its Analysis

To describe the block search algorithm clearly, we introduce a new variable d. The algorithm repeats, skipping $(dn - 1)$, to check each data item in all the data. We set $0 < d < 1$; for example, when $d = 0.1$, the algorithm checks each 10% of the data. That is, the size of each block is dn; in other words, each block contains dn data (see Fig. 3.1). To simplify, the following two assumptions are set.

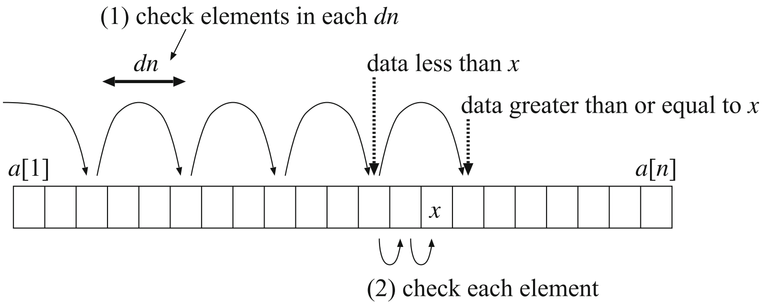

(1) check elements in each dn
dn
data less than x
data greater than or equal to x
$a[1]$
$a[n]$
x
(2) check each element

Fig. 3.1 Basic idea of block search

- Data are stored in $a[1], a[2], \ldots, a[n-1]$, and $a[n]$ can be used for the sentinel. As the sentinel, it is better to assume that $a[n-1] < a[n] = \infty$ rather than $a[n] = x$, as we shall see.[1]
- The number dn can divide n. In other words, if the algorithm checks $a[dn]$, $a[2dn]$, $a[3dn]$, and so on, eventually it certainly hits $a[kdn] = a[n]$ for some integer k, and compares x with $a[n]$, the sentinel, and obtains $x < a[n]$.

Based on the assumptions, the algorithm can be described as follows.

```
Algorithm 18: BlockSearch($a$, $x$)
  Input  : An ordered array $a[1], a[2], \ldots, a[n]$ and data $x$
  Output: The index $i$ if there is an $i$ with $a[i] = x$; otherwise, "Not found."
 1 $i \leftarrow dn$;
 2 while $a[i] < x$ do
 3     $i \leftarrow i + dn$;
 4 end
   /* We can assert that $a[i - dn] < x \leq a[i]$ holds at this point.  */
 5 $i \leftarrow i - dn + 1$;
 6 while $a[i] < x$ do
 7     $i \leftarrow i + 1$;
 8 end
 9 if $a[i] = x$ then
10     output $i$;
11 else
12     output "Not found";
13 end
```

In the while loop from lines 2–4, the algorithm searches the smallest index i with $a[i] \geq x$ for each dn data item. Therefore, when it exits this while loop, we have $a[i - dn] < x \leq a[i]$. Since x can exist from $a[i - dn + 1]$ to $a[i]$, the algorithm

[1] **Implementation of sentinel**:
In the real implementation, we cannot use "the value" ∞, and hence, we have to allow it to be a sufficiently large integer. Several programming languages have such a special value. If your language does not have such a value, you can set the maximum number of the system. We sometimes use the same idea in this book.

checks whether x exists in the data elements in this interval from $a[i - dn + 1]$ one by one in the while loop from lines 6–8. (We can skip the $(i - dn)$th element because we already know that $a[i - dn] < x$.) When the while loop from lines 6–8 is complete, we know that $x \geq a[i]$. This time, in line 9, we can determine whether $a[i]$ contains x for some i with i itself by checking whether $x = a[i]$ or not. By virtue of the sentinel $a[n] = \infty$, it is guaranteed that there exists an i that satisfies $x < a[i]$, which makes the second while loop simple.

Now, we estimate the running time of block search and compare it with that of linear search. How about the first narrowing step? In the worst case, the search checks $n/(dn) = 1/d$ elements in the array $a[i]$ for each dn element. In the average case, we can consider the running time to be $1/(2d)$. The next step is essentially the same as in a linear search. However, this time we have only dn data. Therefore, it runs in dn time in the worst case and $dn/2$ in the average case. Summing up, the algorithm runs in $\Theta(1/d + dn)$ time in both the worst and average case. If we let d be larger, the time of the steps in the former half decreases, but that in the latter half increases. On the other hand, when d is set smaller, the algorithm runs more slowly in the former half. With careful analysis, these two factors become well-balanced when $d = 1/\sqrt{n}$, and then, the running time becomes $\Theta(\sqrt{n})$, which is the minimum.

Exercise 24 Explain that for the function $\Theta(1/d + dn)$, $d = 1/\sqrt{n}$ gives us the minimum function $\Theta(\sqrt{n})$.

Recalling that the linear search runs in $\Theta(n)$, it can be seen that the running time $\Theta(\sqrt{n})$ is a great improvement. For example, when $n = 1000000$, the block search runs 1000 times faster. Moreover, as n becomes larger, the improvement ratio also increases.

3.1.4 From Recursion to Binary Search

If we use a block search instead of a linear search, the running time is drastically improved from $\Theta(n)$ to $\Theta(\sqrt{n})$. Nevertheless, this is not sufficient. How can we improve it further?

In a block search, the algorithm first makes a selection in each interval to narrow the area and then performs a linear search. Why do we not improve the latter half? Then, we can use the idea of recursion described in Chap. 2. To begin with, we introduce a block search to improve the linear search. Nevertheless, we again use the linear search in the latter half in the block search to solve the subproblem. Then, how about the idea of using a block search in this part recursively? It seems to improve the latter half. Letting d_1 be the first parameter to adjust in the first block search, its running time is $\Theta(1/d_1 + d_1 n)$. The factor $d_1 n$ indicates the running time for solving the subproblem by using a linear search. This part can be improved by replacing it with a block search. Let d_2 be the second parameter for the second block search. Then, the running time is improved to $\Theta(1/d_1 + 1/d_2 + d_1 d_2 n)$. However, the factor $d_1 d_2 n$ is

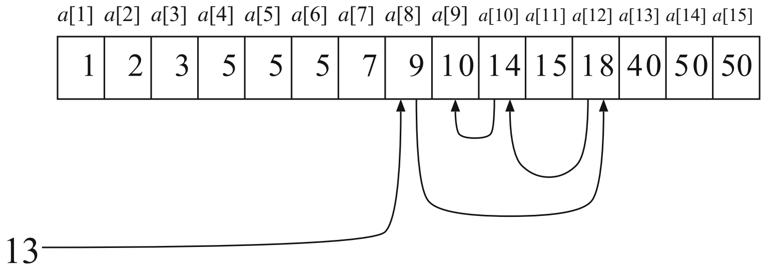


Fig. 3.2 Illustration for binary search

solved by a linear search again, and therefore, we can again apply a block search with parameter d_3 and obtain a shorter running time, $\Theta(1/d_1 + 1/d_2 + 1/d_3 + d_1 d_2 d_3 n)$. We can repeat this recursive process until we have only one element in the array. If the array has only one element $a[i]$, the algorithm outputs i if $a[i] = x$; otherwise, "Not found."

Now, we turn to consider the parameters $d_1, d_2, d_3, \ldots$. We omit the detailed analysis of this part since it is above the level of this book, but it is known that it is best to choose $d_1 = d_2 = \cdots = 1/2$, in other words, to check the item at the 50% point in the range at every step. Namely, it is most effective to check the central item in the range. If the algorithm hits the desired item at the center, it halts. Otherwise, you can repeat the same process for the former or the latter half. To summarize, the process of the "recursive block search" is as follows.

(1) Compare x with the $a[i]$, which is the central element in a given array.
(2) If $a[i] = x$, then output i and halt.
(3) If the current array consists of one element, output "Not found" and halt.
(4) If $a[i] > x$, make a recursive call for the former half.
(5) If $a[i] < x$, make a recursive call for the latter half.

This method is usually called **binary search** since it divides the target array into two halves at each repeat.

Example 5 For the array given in Fig. 3.2, let us perform a binary search with $x = 13$. The number of elements in $a[]$ is 15; thus, we check whether $a[8] = 9$ is equal to $x = 13$ or not. Then, we have $x > a[8]$. Thus, the next range of the search is $a[9], \ldots, a[15]$. Here, we have seven elements; therefore, we check whether $a[12] = 18$ is equal to x and obtain $a[12] > x$. Thus, the range is now $a[9], a[10], a[11]$. Comparing $a[10] = 14$ to $x = 13$, we have $a[10] > x$. Next, we check $a[9] = 10$ and $x = 13$ and conclude that the array $a[]$ does not contain x as a data item.

Exercise 25 Implement a binary search on your computer.

Now, we consider the running time of the binary search method. The essence of the running time is the number of comparisons. Therefore, we here simplify the analysis and investigate only the number of comparisons. (The entire running time is proportional to the number of comparisons; it is essentially the same as considering it by using the O-notation.) The algorithm halts if it finds $a[i] = x$ for some i; however, we consider the worst case; that is, the algorithm checks the last element and outputs "Not found." In this case, the range of the array is halved in each comparison and the algorithm halts after checking the last element in the array of size 1. Therefore, the number of comparisons $T_B(n)$ in the binary search on an array of size n satisfies the relations

$$\begin{aligned} T_B(n) &= 1 \qquad (\text{if } n = 1) \\ T_B(n) &= T_B(\lceil (n-1)/2 \rceil) + 1 \qquad (\text{if } n > 1), \end{aligned}$$

where $\lceil (n-1)/2 \rceil$ denotes the minimum integer greater than or equal to $(n-1)/2$. In general, the algorithm picks up one data element from n data, compares it, and divides the remaining $(n-1)$ elements. In this case, if $n-1$ is even we have $(n-1)/2$ elements, but we still have $n/2$ in the worst case (or a larger range) if $n-1$ is odd. Since such details are not essential, we simplify the equations as

$$\begin{aligned} T'_B(n) &= 1 \qquad (\text{if } n = 1) \\ T'_B(n) &= T'_B(n/2) + 1 \qquad (\text{if } n > 1). \end{aligned}$$

Moreover, we assume that $n = 2^k$. Then, we have

$$T_B(2^k) = T_B(2^{k-1}) + 1 = T_B(2^{k-2}) + 2 = \cdots = T_B(1) + k = 1 + k.$$

From $n = 2^k$, we obtain $k = \log n$. In general, if n cannot be represented in 2^k for an integer k, we may add imaginary data and consider the minimum n' such that $n < n' = 2^k$ for some integer k. In this case, we can analyze for n' and apply the analysis results to the smaller n. From these arguments, we have the following theorem.

Theorem 6 *The binary search on n data runs in* $O(\log n)$ *time.*

As emphasized in the logarithmic function in Sect. 5, the function in $O(\log n)$ is quite small, and the binary search runs quite fast. In comparison to the original time $\Theta(n)$ for linear search, it may be said to be "almost constant time." For example, the population of Japan is around 120,000,000 and $\log 120000000 = 26.8385$. That is, if you have a telephone book that contains the entire Japanese population, you can reach any person in at most 27 checks. If you use a linear search, the number of checks is still 120,000,000, and even the block search of $O(\sqrt{n})$ also requires checking $\sqrt{12000000} \sim 11000$ times. Therefore, 27 is notably smaller.

Exercise 26 ☕☕ The largest Japanese dictionary contains of around 240,000 words. If you use a binary search, how many words need to be checked to find a word in the worst case?

We have considered the linear, block, and binary search methods. One may think that the binary search method, as an improvement on the block search method, is complicated. However, the binary search algorithm is indeed simple. Therefore, the implementation is easy, and it is in fact quite a fast algorithm. For this reason, the block search method is not used in practice.

Lower bound of searching. The binary search method is a simple and fast algorithm. In fact, it is the best algorithm and cannot be improved further from the theoretical viewpoint. More precisely, the following theorem is known.[2]

Theorem 7 *Let A be any algorithm that searches by comparing a pair of data items. If A can use fewer than* $\log n$ *comparison operations, the search problem for n data cannot be solved by A.*

To give a strict proof, we have to borrow some notions from information theory. However, intuitively, it is not very difficult to understand. As seen in the RAM model in Sect. 1, to store n different data in a memory cell and give them unique addresses (or indices), each address requires $\log n$ bits. This fact implies that, to distinguish n data, $\log n$ bits are necessary. On the other hand, when we compare two data items, we obtain 1-bit information about their ordering. Therefore, to specify the position of x in the array $a[]$ of n elements, we need to make $\log n$ comparisons to obtain that $\log n$ bit address. In other words, if some algorithm uses only fewer than $\log n$ comparisons, we can make counter data to the algorithm. For this counter data, the algorithm will leave two or more data that should be compared with x.

3.2 Hashing

In Theorem 7, we show the lower bound $\log n$ comparisons to solve the searching if the algorithm uses *comparison as a basic operator*. If you sense something strange about the basic operation, you are clever. Indeed, sometimes we can break this limit by using another technique not based on comparison. The **hash** is one of them. We first observe a simple hash to understand the idea.

Simple hash. The search problem asks whether there is an index i with $a[i] = x$ for a given array $a[1], a[2], \ldots, a[n]$ and data x. We here assume that we know beforehand that "x is an integer between 1 and n." For example, most student examination scores are between 1 and 100; hence, this assumption is a simple but realistic one. The array $a[i]$ indicates whether there is a student whose score is i. To represent this, we define that $a[i] = 0$ means "there is no data item of value i" and $a[i] = i$ if "there is a data item of value i", and the array is supposed to be initialized following the definition.

[2] **Results of a comparison**:
Here, to simplify, we assume that $x < y$ or $x > y$ holds in a comparison. We omit the other possible case, $x = y$, in this book. Another issue is that $\log n$ is not an integer in general. To be precise, we have to take $\lceil \log n \rceil$ or $\lfloor \log n \rfloor$ according to the context, which is also omitted in this book.

Then, for a given x, we can solve the search problem in $O(1)$ time by checking whether $a[x] = x$.

Idea of general Hash. Now, we turn to general hash. In general, there is no limitation on the range of x. Thus, x can be much larger than n. How can we deal with such a case? The answer to this question is the notion of the **hash function**. The hash function $h(x)$ should satisfy two conditions:

(1) The function $h(x)$ takes an integer in the area $[1...n]$.
(2) If x is in the array $a[]$, it is stored as in $a[h(x)]$.

If we already know that the data in the array $a[]$ and a function h satisfy these conditions, the idea of hash can be described in algorithmic form as

```
Algorithm 19: Hash($a, x$)
  Input  : An array $a[1], a[2], \ldots, a[n]$ and data $x$
  Output: The index $i$ if there is an $i$ with $a[i] = x$; otherwise, "Not found."
if $a[h(x)] = x$ then
    output $h(x)$;
else
    output "Not found";
end
```

Since it is quite simple, it is easy to see that the algorithm certainly finds x if it exists, and its running time is $O(1)$. In short, the function $h(x)$ computes the index of x in the array $a[]$. Thus, if the function $h(x)$ is correctly implemented, the searching problem can be solved efficiently. In the notion of hash, x is called a **key** and $h(x)$ is called the **hash value** for the key x.

Implementation of hash function. As we have already seen, to perform the hash function correctly its implementation is crucial. Unfortunately, we have no silver bullet for designing a universal hash function. The difficulty is in satisfying the second condition. To achieve this,

(3) For any pair x and x', we have to guarantee that $x \neq x'$ implies $h(x) \neq h(x')$.

Otherwise, for some two different keys x and x', we will refer to the same data $a[h(x)] = a[h(x')]$. In the general case, or when there is no limit to the key, we cannot guarantee this condition. That is, even if we design a "nice" hash function, we can say at most that

(3') For $h(x)$, we have few x' with $x \neq x'$ such that $h(x) = h(x')$.

A case where "$x \neq x'$ and $h(x) = h(x')$" is called a **collision** of the hash value. When you use a hash function, you have to create a design that avoids collisions as far as possible, and you have to handle collisions. Hereafter, we introduce the design technique for hash functions and the manner in which we can handle collisions. This book introduces only a few typical and simple techniques, but there are many others for handling collisions.

Typical hash function. When x takes an integer, we have to map to an integer in $[1...n]$, and the most simple hash function is "(x modulo n) + 1." If x can be biased, choosing suitable integers a and b beforehand, we take "(($ax + b$) modulo n) + 1." Experimentally, it behaves well when you take a, b and n relatively prime to each other. If x is not an integer, we can use its binary representation in the computer system as a binary number and handle in the same way.

Avoiding collision. For a given key x, assume that it occurs that $a[h(x)] = x'$ with $x \neq x'$; that is, we have a collision. In this case, a simple solution is to prepare another function to indicate "the next position" for x. Among these techniques, the simplest is to let "the next position" be the next empty element in $a[]$. That is, algorithmically, it is described as

(1) Let $i = 0$.
(2) For a given key x, check $a[h(x) + i]$, and report $h(x) + i$ if $a[h(x) + i] = x$.
(3) If $a[h(x) + i]$ is empty, report "Not found."
(4) Otherwise, if $a[h(x) + i] = x'$ for some x' with $x' \neq x$, increment i (or perform $i \leftarrow i + 1$), and go to step 2.

The implementation is easy and it runs efficiently if few collisions occur.

Caution for usage. If you can use the hash function correctly, your algorithm runs fast. If no or few collisions occur, the search can be completed in $O(1)$ time. Therefore, the crucial point is to design the hash function $h(x)$ such that a few or fewer collisions occur. To achieve this, the density of the data in $a[]$ is one of the issues. In a general search problem, sometimes new data are added, and useless data are removed. Therefore, not all the elements in $a[]$ are necessarily valid. Rather, if $a[]$ contains few data, few collisions occur, even if you use a simple hash function. In contrast, if most of $a[]$ are used to store data, collisions frequently occur and the time required for the computation to avoid collisions becomes an issue. Since the avoiding method introduced above essentially performs a linear search, if collisions occur frequently, we have to refine this process. There are many tricks for handling this problem according to the size of the memory and the number of data; however, we omit them in this book.

3.3 Sorting

Next, we turn to the sorting problem. As seen in the binary search, if the array is organized beforehand, we can search quite efficiently. If all the items in your dictionary are incoherently ordered, you cannot find any word. Thus, you can understand that huge data are meaningful only if the entries are in order. Therefore, the following sorting problem is important in practice.

Sorting:

Input: Array $a[1], a[2], \ldots, a[n]$.
Output: Array $a[1], a[2], \ldots, a[n]$, such that $a[1] \leq a[2] \leq \cdots \leq a[n]$ holds.

We first mention two points that should be considered when handling the sorting problem.

First, it is required that in the array, the ordering of each pair is defined correctly. (Using a technical term, we can say that we have "total ordering" over the data.) For example, in the game of "paper, rock, scissors," we have three choices and each pair has an ordering. However, they are not in total ordering, and hence, we cannot sort them. More precisely, the data should have the following properties. (1) For each pair of x and y, we have $x < y$ or $x > y$, and (2) for each triple of x, y, and z, $x < y$ and $y < z$ implies $x < z$. In other words, once we know $x < y$ and $y < z$, we can conclude $x < z$ without comparing them. Oppositely, not only numerical data, but also other data can be sorted if total ordering is defined over the data. Typically, we can sort alphabetical words as in a dictionary.

Second, in sorting, if the data contain two or more items with the same value, we have to be careful. Depending on the sorting algorithm, the behavior differs for two items of the same value. For example, when we have two items $a[i] = a[j]$ with $i < j$, the sorting algorithm is called **stable** if the ordering of $a[i]$ and $a[j]$ is always retained in the output. If we are dealing with only numerical data, even if the items are swapped, we are not concerned. However, the target of sorting may not be only numerical data. Sometimes the bunch of data consists of some indices of complicated data. For example, imagine that you have a huge amount of data consisting of responses to a questionnaire, and you wish to sort them by the names of the responders. In this case, it is not good if the attributes of two people of the same name are swapped. Thus, the stability of sorting can be an issue in some applications.

3.3.1 Bubble Sort

One of the simplest sorting techniques is called **bubble sort**. The basic idea is natural and simple: Compare two consecutive items, swap them if they are in reverse order, and repeat. Now we explain in more detail. The algorithm first performs the following step.

- For each $i = 1, 2, \ldots, n-1$, compare $a[i]$ and $a[i+1]$, and swap them if $a[i] > a[i+1]$.

The important point is that after this step, *the biggest item is put into* $a[n]$. It is not difficult to observe this fact since the algorithm swaps from $a[1]$ to $a[n-1]$. Once the algorithm has hit the biggest item, this item is always compared with the next item one by one, and eventually is moved to $a[n]$. Then, we can use the same algorithm for the items in $a[1]$ to $a[n-1]$. After this step, the second biggest item is put into $a[n-1]$. We can repeat this process for each subproblem, and we finally have an array $a[1]$ of size one, which contains the smallest item, and the sorting is complete. Since we can imagine that the big item floats to the end of the array and the small items stay condensed at the front, this method is called **bubble sort**.

Example 6 We confirm the behavior of the bubble sort method. Let assume that the array contains the data

$$15\ 1\ 5\ 8\ 9\ 10\ 2\ 50\ 40.$$

First, we proceed to compare and swap each pair of two items from the top of the array. Specifically, we first compare (15, 1) and swap them. The next pair is (15, 5) and its components are also swapped. We demonstrate this below. The pair compared is underlined.

$$\begin{array}{ccccccccc}
\underline{15} & \underline{1} & 5 & 8 & 9 & 10 & 2 & 50 & 40 \\
1 & \underline{15} & \underline{5} & 8 & 9 & 10 & 2 & 50 & 40 \\
1 & 5 & \underline{15} & \underline{8} & 9 & 10 & 2 & 50 & 40 \\
1 & 5 & 8 & \underline{15} & \underline{9} & 10 & 2 & 50 & 40 \\
1 & 5 & 8 & 9 & \underline{15} & \underline{10} & 2 & 50 & 40 \\
1 & 5 & 8 & 9 & 10 & \underline{15} & \underline{2} & 50 & 40 \\
1 & 5 & 8 & 9 & 10 & 2 & \underline{15} & \underline{50} & 40 \\
1 & 5 & 8 & 9 & 10 & 2 & 15 & \underline{50} & \underline{40} \\
1 & 5 & 8 & 9 & 10 & 2 & 15 & 40 & \boxed{50}
\end{array}$$

At this step, it is guaranteed that the last item, 50, is the maximum value, and therefore, we never touch it hereafter. In the following, we denote the fixed maximum value (locally) by a rectangle. We now focus on the array preceding the last value, 50, and repeat comparisons and swapping in this area. Then, we have

$$1\ 5\ 8\ 9\ 2\ 10\ 15\ \boxed{40}\ \boxed{50}$$

Now, we confirm that 40 is the maximum value, and we never touch it hereafter. We repeat,

$$1\ 5\ 8\ 2\ 9\ 10\ \boxed{15}\ \boxed{40}\ \boxed{50}$$

and similarly 15 is fixed,

$$1\ 5\ 2\ 8\ 9\ \boxed{10}\ \boxed{15}\ \boxed{40}\ \boxed{50}$$

10 is fixed,

$$1\ 2\ 5\ 8\ \boxed{9}\ \boxed{10}\ \boxed{15}\ \boxed{40}\ \boxed{50}$$

and 9 is fixed. In this example, although no more sorting is required, the real algorithm proceeds and determines the ordering of the values one by one.

The bubble sort algorithm can be described as follows.

```
Algorithm 20: BubbleSort($a$)
  Input  : An array $a[]$ of size $n$
  Output: An array $a[]$ that is sorted
1  for $j \leftarrow n-1, n-2, \ldots, 2$ do
2      for $i \leftarrow 1, 2, \ldots, j$ do
3          if $a[i] > a[i+1]$ then
4              $tmp \leftarrow a[i]$;
5              $a[i] \leftarrow a[i+1]$;
6              $a[i+1] \leftarrow tmp$;
7          end
8      end
9  end
10 output $a[]$;
```

In lines 4–6, the algorithm swaps two elements $a[i]$ and $a[i+1]$ using a temporary variable tmp. (Of course, you can use the technique in Exercise 3. The method in Exercise 3 has an advantage that it requires no additional memory, but it seems to be less popular because it requires additional mathematical operations.)

The bubble sort algorithm is simple and easy to implement. Therefore, it runs efficiently when n is small. However, when n increases, the running time becomes longer. The details are discussed in Sect. 3.3.4.

3.3.2 Merge Sort

The **merge sort** method is one of the fast sorting algorithms and is based on a simple idea. It is also a nice example of the notion of **divide and conquer** described in Chap. 2.

In the merge sort method, the algorithm first only splits the array into the former half and the latter half. Then, it sorts each half recursively, based on the idea of divide and conquer. We do not consider how to solve these two subproblems at this point. We just assume that we have succeeded in sorting the former and latter halves. Now, we have obtained two sorted arrays. Then, we need to merge these two sorted arrays into one sorted array. How can we do this? We use an additional array. If we have a new array, it is sufficient to compare the top elements of these two arrays, select the smaller element, and put it into the new array. We already know that these two

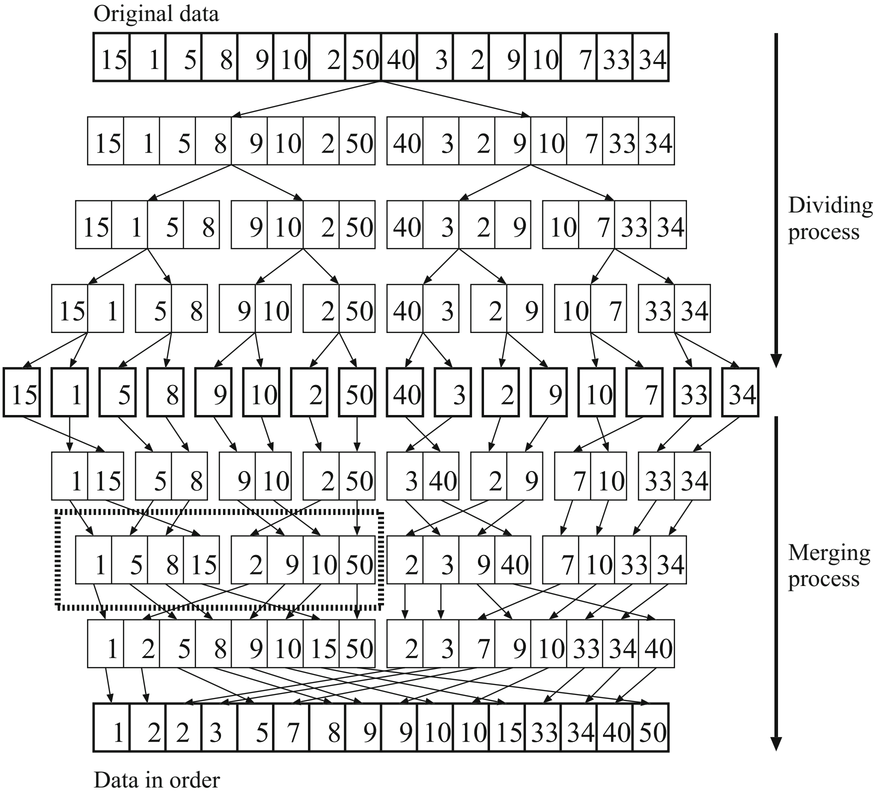


Fig. 3.3 Behavior of merge sort

arrays have already been sorted, and it is sufficient to compare the top elements in these two arrays each time, select the smaller one, and add it as the last element of the new array.

To achieve the divide-and-conquer step, we can use recursive calls. When we use recursive calls correctly, we have to give the last step for the extremely small subproblem. In the merge sort method, it is sufficient to consider the array when it contains only one element, and we cannot split it further. This is simple; just return the unique element.

Example 7 We check the behavior of the merge sort algorithm using a concrete example. Assume that the content of a given array is

15 1 5 8 9 10 2 50 40 3 2 9 10 7 33 34

Merge sort is a sorting method based on dividing and conquering by making recursive calls. The array is sorted as shown in Fig. 3.3. First, the array is divided into two halves repeatedly until each subarray consists of one element. When the array has been completely divided, the merging process starts. Since it is guaranteed that any two arrays are already sorted, when its aim is to merge them, the algorithm can eventually obtain one sorted array by merging two arrays of the previous elements. For example, in the figure, when the algorithm merges two subarrays 1, 5, 8, 15 and 2, 9, 10, 50 (in the dotted box), it first compares the top elements of the two arrays, 1 and 2. Then, since 1 is smaller, it moves 1 to the auxiliary array and compares the next items, 5 and 2. Then, 2 is smaller than 5 in this case, so 2 is placed after 1 in the auxiliary array, and the algorithm next compares 5 and 9. In this way, the algorithm always compares the top elements in the two given subarrays and adds the smaller one at the end of the auxiliary array, and finally, it obtains 1, 2, 5, 8, 9, 10, 15, 50 in the auxiliary array.

In merge sort, subproblems are solved recursively, and hence, information on the first and last positions of the array to be sorted is required. We suppose that the algorithm sorts an array $a[]$ from index i to index j and calls a procedure of the MergeSort(a, 1, n) style. We also suppose that this algorithm itself produces no output, and the input array is sorted after performing the algorithm. In the following algorithm, the variable m is the medium index of the array. More precisely, the array $a[i], a[i+1], \ldots, a[j]$ is divided into two subarrays $a[i], a[i+1], \ldots, a[m]$ and $a[m+1], a[m+2], \ldots, a[j]$ of the same size, which are sorted separately and merged afterward. In the merging process, the algorithm compares $a[p]$ in $a[i], \ldots, a[m]$, and $a[q]$ in $a[m+1], \ldots, a[j]$. It also prepares an auxiliary array $a'[]$ and uses $a'[i], a'[i+1], \ldots, a'[j]$ to store the merged data. In addition, it uses a variable r for indexing in $a'[]$. We also use another popular, but not beautiful, technique called "flag" to check whether one of two arrays becomes empty.[3] This variable flag is initialized by 0 and becomes 1 if one of two subarrays becomes empty.

[3] **Flag**:
As introduced in the text, this is a common technique in programming where a variable that indicates that the status is changed from, say, 0 to 1, is used. This variable is called a **flag**. We sometimes use the expression that a flag "stands" when the value of the flag is updated. Maybe it comes from the image of setting or positioning small flags. The idea of a flag is useful, but it is not very smart in general since it tends to make the structure of the program untidy. If you can use some standard control statements, such as while, repeat, for, and others, it is better to do so. It might be better to reconsider the structure or logic in the program if you cannot avoid it.

```
Algorithm 21: MergeSort($a, i, j$)
  Input  : An array $a[1], a[2], \dots, a[n]$ and two indices $i$ and $j$.
  Output: Array $a[1], a[2], \dots, a[n]$, such that $a[i] \le a[i+1] \le \cdots \le a[j]$ holds (and the
          other range does not change).
 1 if $i \neq j$ then                                  /* do nothing if $i = j$ */
 2     $m \leftarrow \lfloor (i+j)/2 \rfloor$ ;                      /* $m$ indicates the center */
 3     MergeSort($a, i, m$); MergeSort($a, m+1, j$) ;          /* sort recursively */
 4     $p \leftarrow i; q \leftarrow m+1; r \leftarrow i$ ;                 /* initialize variables */
 5     $flag \leftarrow 0$ ; /* one array becomes empty if this flag becomes 1 */
 6     while $flag = 0$ do
 7         if $a[p] > a[q]$ then  /* move the top element in the latter half
            */
 8             $a'[r] \leftarrow a[q]$;
 9             $q \leftarrow q+1$;
10             if $q = j+1$ then $flag \leftarrow 1$ ;
11         else          /* move the top element in the former half */
12             $a'[r] \leftarrow a[p]$;
13             $p \leftarrow p+1$;
14             if $p = m+1$ then $flag \leftarrow 1$ ;
15         end
16         $r \leftarrow r+1$;
17     end
18     while $p < m+1$ do  /* some elements are left in the former half
        */
19         $a'[r] \leftarrow a[p]$;
20         $p \leftarrow p+1$;
21         $r \leftarrow r+1$;
22     end
23     while $q < j+1$ do  /* some elements are left the latter half */
24         $a'[r] \leftarrow a[q]$;
25         $q \leftarrow q+1$;
26         $r \leftarrow r+1$;
27     end
28     copy the elements in $a'[i], \dots, a'[j]$ to $a[i], \dots, a[j]$;
29 end
```

In line 3, the algorithm sorts two divided subarrays by two recursive calls. Lines 6–17 deal with the first merging process, which copies smaller elements into $a'[r]$ by comparing the two top elements in the two subarrays. (The case where the latter subarray has a smaller element is dealt with in lines 8, 9, and 10, and the case where the former subarray has a smaller element (or the two elements are equal) is handled in lines 12, 13, and 14.) This first process continues until one of the two subarrays becomes empty. If one subarray becomes empty, the algorithm performs lines 18–28. If $flag \leftarrow 1$ is performed in line 10, it runs in lines 18–22, and if $flag \leftarrow 1$ is performed in line 14, it runs in lines 23–27. That is, the algorithm runs only one of the routines written in lines 18–22 or lines 23–27. In this process, all the leftover elements are copied to $a'[r]$. The description of the program itself is long, but it is easy to follow if you are sure of the idea of merge sort. As shown later, the merge sort runs quite fast from not only the theoretical but also the practical viewpoint. It runs constantly fast for any input and is also a stable sort.

Tip on implementation. The merge sort is relatively simple, but it requires copying to an additional array of the same size. In the above description of the algorithm, we use an additional array $a'[]$ of the same size as $a[]$. Concretely, given indices i and j, the algorithm divides the range from i to j into halves, sorts them separately, and merges them by writing data to $a'[i]$ to $a'[j]$. In the naive implementation, it writes back into $a[]$ in line 28. In other words, the algorithm always uses $a[]$ as a main array and $a'[]$ as an auxiliary array. This part can be made faster by using a neat idea. That is, we alternate $a[]$ and $a'[]$ as a main array and an auxiliary array. After using $a'[]$ as the auxiliary array and sorting the main array $a[]$, next we use $a'[]$ as the main array and $a[]$ as the auxiliary array. Then, we no longer need the copy operation in line 28, and the number of copy operations is halved. This small tip works effectively when the data are huge.

On the other hand, in the merge sort algorithm, avoiding this auxiliary array in order to save memory is difficult. Namely, if you need to sort huge data, you have to prepare an additional space of the same size, which is a weak point of merge sort. This is one of main reasons why it is not used as frequently as the quick sort algorithm, which is introduced in the next section.

3.3.3 Quick Sort

The **quick sort** is an algorithm that in general is used most frequently. As the name indicates, it runs quite fast. From both the practical and theoretical viewpoint, it has some interesting properties, and we should learn and understand the basic ideas in order to be able to use quick sort correctly. As is the merge sort algorithm, the quick sort algorithm is based on the idea of **divide and conquer**. In merge sort, we divide the array into two halves, solve two subproblems, and merge them later. In quick sort, we divide the array into two parts by collecting “smaller elements” into the former part and “bigger elements” into the latter part. More precisely, in quick sort, we arrange the array so that each element x in the former part is less than or equal to any element y in the latter part. Later, the array is divided into two parts. In other words, after dividing in quick sort, the maximum element x in the former part and the minimum element y in the latter part satisfy $x \leq y$.

The key for the division is an element called the **pivot**. The pivot is an element in the given array, and the algorithm uses it as a point of reference to divide it into “smaller” and “bigger” elements. The selection of the pivot is one of the main issues in quick sort; however, meanwhile, we assume that the algorithm chooses the last element in the array to make the explanation simple. We remark that the number of smaller and bigger elements is not the same in general. That is, in general, the array will be divided into two subarrays of different size. This is the key property of quick sort. If you choose a pivot that divides the array into two subarrays of almost the same size, the algorithm runs efficiently. On the other hand, if you fail to choose a pivot such as this and obtain two subarrays of quite different size, it seems that sorting does not go well. We will consider this issue more precisely later.

Once the algorithm selects the pivot, p, it rearranges the array such that the former part contains elements less than or equal to p and the latter part contains elements greater than or equal to p. We note that elements equal to p can be put into either the former or the latter part, according to the implementation of the algorithm. After dividing the elements into the former and latter parts, the algorithm places p between them in the array, and then it has three parts: the former part, the pivot, and the latter part. Now, after making recursive calls for the former and the latter part, the quick sort ends. As in the merge sort method, if the size of the array is 1, the algorithm has nothing to do. However, we have to consider the case where the size of the subproblem becomes 0 suddenly because of the selected pivot. This case occurs if the selected pivot is accidentally the minimum or maximum in the array. Therefore, in the last basic part of the recursive call, the algorithm does nothing if the size of the array is *less than* or equal to 1. This is the end of the explanation of the basic idea of quick sort. The basic idea itself is not very difficult to comprehend.

Example 8 Now we observe the behavior of the quick sort algorithm for a concrete input. Let us assume that the array consists of the data

$$40\ 1\ 5\ 18\ 29\ 10\ 2\ 50\ 15$$

Then, the first pivot is the last element, 15. Now, the smaller elements are 1, 5, 10, and 2, and the bigger elements are 40, 18, 29, and 50. Therefore, the array is rearranged as

$$1\ 5\ 10\ 2\ \underline{15}\ 40\ 18\ 29\ 50$$

The pivot is indicated by the underline. Hereafter, this pivot, 15, is fixed in the array, and never touched by the algorithm. Now, two recursive calls are applied to the former part and the latter part. Here, we consider only the former part:

$$1\ 5\ 10\ 2$$

Then, the next pivot is 2. The array is divided into two parts by this pivot, and we obtain

$$1\ \underline{2}\ 5\ 10$$

Now, the former part contains only one element 1, and the recursive call halts for this data item. For the latter part,

$$5\ 10,$$

a recursive call is applied. Then, 10 is the next pivot and is compared to 5.

While the basic idea of the quick sort method is simple, it is difficult to implement as an accurate algorithm without bugs. In fact, it is well known that some early papers contain small bugs in their algorithm descriptions. Such bugs can appear when a given array contains many elements of the same value; this stems from the fact that we have some options for the boundary between the former and the latter part. (Remember that elements of the same value as the pivot can be put into either part.) In this paper, we introduce a simple implementation method that handles sensitive cases appropriately.

Implementation of quick sort. The main part of the quick sort algorithm, for two given elements $a[s]$ and $a[t]$, divides the array into two parts using the pivot, which is selected as described in the previous procedure. Then, it makes two recursive calls for the former part and the latter part, respectively. This implementation is simple; a typical implementation is given below.

```
Algorithm 22: QuickSort($a, s, t$)
  Input  : An array $a[]$, two indices $s$ and $t$ with $s \leq t$
  Output: The array $a[]$ such that the part from $a[s]$ to $a[t]$ is sorted, and the second part is not
          touched
if $s < t$ then    /* Do nothing if the array contains 0 or 1 element
  */
    $m$ ←Partition($a, s, t$) ;              /* $m$ is the index of the pivot */
    QuickSort($a, s, m - 1$);
    QuickSort($a, m + 1, t$);
end
```

The key point of the implementation of the quick sort algorithm is the procedure named Partition. This procedure does the following. (1) Decides the pivot from $a[s]$ to $a[t]$, (2) rearranges the other elements using the pivot, and (3) returns the index m, such that $a[m]$ is the correct place for the pivot. That is, after performing the partition procedure, any element in $a[s]$ to $a[m-1]$ is less than or equal to $a[m]$, and any element in $a[m+1]$ to $a[t]$ is greater than or equal to $a[m]$. Therefore, after this procedure, we do not consider $a[m]$ and sort the former part and the latter part independently. Hereafter, we consider the details of the implementation of the partition procedure.

Basically, the partition procedure using the pivot is performed from the top of the array. We consider the details of the partition algorithm that divides the array from $a[s]$ to $a[t]$ into two parts. Hereafter, we assume that the last element $a[t]$ is chosen as the pivot p. Using two additional variables, i and j, we divide the array from $a[s]$ to $a[t-1]$ into three parts as described below. Then, we maintain the assertions for each of three parts[4]:

[4] **Assertion of a program**:
A property that always holds in a computer program is called an **assertion**. By maintaining reasonable assertions, we can avoid introducing bugs into our programs.

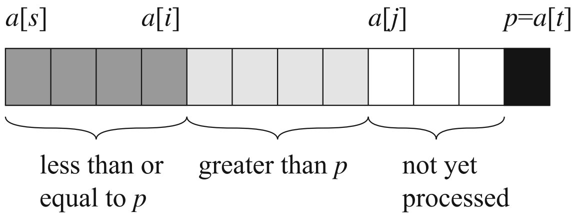


Fig. 3.4 Assertion in partition

- From $a[s]$ to $a[i]$: They contain elements less than or equal to p.
- From $a[i+1]$ to $a[j-1]$: They contain elements greater than p.
- From $a[j]$ to $a[t-1]$: They are not yet touched by the algorithm.

The states of assertions are illustrated in Fig. 3.4. We note that this partition has no ambiguity; each element equal to the pivot should be put into the former part. If we maintain these three assertions and j reaches t from s, we can swap $a[t]$ (pivot) and $a[i+1]$, and return $i+1$ as the index m. A concrete example of the implementation of partition is given below. (Here, swap(x, y) indicates a subroutine that swaps x and y. We have already used this in the bubble sort method; alternatively, we can use the technique in Exercise 3.)

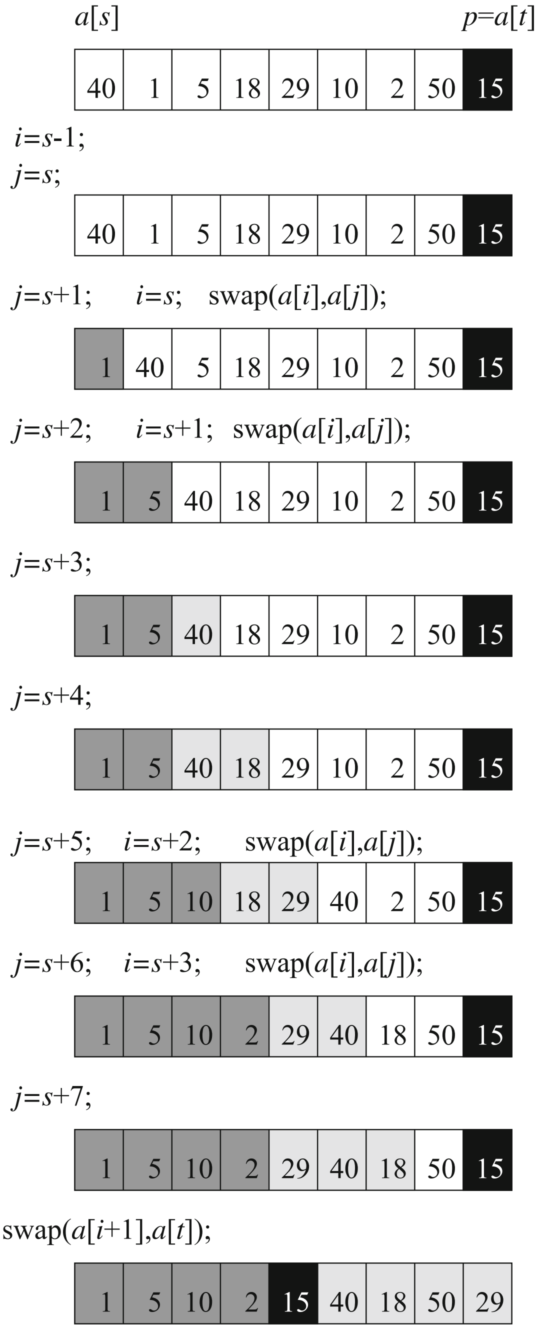
```
Algorithm 23: Partition($a, s, t$)
  Input  : An array $a[]$ and two indices $s$ and $t$ with $s \leq t$
  Output: Divided $a[]$ and the index of the pivot in the array
$p \leftarrow a[t]$;
$i \leftarrow s-1$;
for $j \leftarrow s, s+1, \ldots, t-1$ do
    if $a[j] \leq p$ then
        $i \leftarrow i+1$;
        swap($a[i], a[j]$);
    end
end
swap($a[i+1], a[t]$);
return $i+1$;
```

By the assertions, during the performance of the program, the array is always partitioned into four blocks, as shown in Fig. 3.4 (we consider the last pivot $p = a[t]$ is also one block of size 1). Some of the four blocks may be of size 0 in some cases. Including these extreme cases, we can confirm that the assertions hold during the performance of the program with this implementation. The behavior of this algorithm for the array

40 1 5 18 29 10 2 50 15

is depicted in Fig. 3.5. It is not difficult to follow the processes one by one in the figure.

Fig. 3.5 Behavior of partition



3.3.4 Analyses and Properties of Sorting

In this section, we show the properties of the three sorting algorithms, analyze their complexities, and compare them. In particular, we need to learn well the properties of the quick sort method. In the merge sort and quick sort methods, the description of the algorithms becomes complicated. For example, a recursive call requires many background tasks, which do not appear explicitly. However, such a detail is not essential from theoretical viewpoint, and we ignore these detailed task system maintains. To compare the three sort algorithms, we count the number of comparison operations in them. That is, we regard the time complexity of a sorting algorithm to be proportional to the number of comparison operations.

Analysis of bubble sort. The bubble sort algorithm is simple, and it is not difficult to count the number of comparison operations. At the first step, in an array of size n, to put the biggest element into $a[n]$, it performs

- for each $i = 1, 2, \ldots, n-1$, compare $a[i]$ and $a[i+1]$, and swap them if $a[i] > a[i+1]$.

This step requires, clearly, $n-1$ comparison operations. Similarly, it performs $n-2$ comparison operations to put the biggest element in the array of size $n-1$ to $a[n-1]$, ..., and 1 comparison operation to put the biggest element in the array of size 2 to $a[2]$. In total, it performs $(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$ comparison operations. Using the O notation, this algorithm runs in $\Theta(n^2)$ time.

The bubble sort algorithm needs a few extra variables; the space complexity (excluding the input array) is $O(1)$, which means that it uses a constant number of variables. Moreover, since the algorithm does nothing when $a[i] = a[i+1]$, it is a stable sort.

Analysis of merge sort. For the merge sort algorithm, space complexity is easier to determine than time complexity. It is not difficult to see that the algorithm needs an additional array of the same size as $a[n]$ and some additional variables. Therefore, the algorithm requires $\Theta(n)$ space complexity. Sometimes, this is an issue when n is large and each element occupies many memory cells.

Next, we turn to time complexity. Figure 3.3 gives us an important clue for estimating the time complexity of the merge sort algorithm. A given array of size n is divided into two parts of the same size. Next, each subarray is again divided into two parts of the same size; namely, we obtain four subarrays of size $n/4$. It is not easy to see the time complexity if we consider each subarray, and therefore, we consider the levels of the dividing process. That is, the first division is in level 1, and we obtain two subarrays of size $n/2$. In the next level 2, we obtain four subarrays of size $n/4$ with two divisions. In the level 3, we obtain eight subarrays of size $n/8$ with four more divisions. That is, in each level i, we apply the dividing process 2^{i-1} times and obtain 2^i subarrays of size $n/2^i$. (Precisely speaking, we have some errors if $n/2^i$ is not an integer, but we do not pay attention to this issue here.) Eventually, if the size of the subarray becomes one, we cannot divide it further. In other words, the division process will be performed up to the level k, where k is an integer satisfying $2^{k-1} < n \leq 2^k$. This implies $k = \lceil \log n \rceil$. That is, the division process will be performed to the level $\lceil \log n \rceil$.

Then, how about the time complexity in level i? In each level, an array is divided into two subarrays, and two subarrays are merged into one array after two recursive calls for the subarrays. In these two processes, the first process is easy; only the central index, which takes $O(1)$ time, is computed. The second process is the issue.

First, when we estimate time complexity of an algorithm, we have to count the number of operations along the flow of the algorithm, which is determined by each statement in the algorithm. However, in the merge sort method, it is not so easy to follow the flow of the algorithm.

Therefore, we change our viewpoint from that of the algorithm to that of the manipulated data. That is, we count the number of operations that deal with each data item from the viewpoint of the data themselves. After the division in level i, each subarray of size $n/2^i$ is merged with another subarray of size $n/2^i$ and they together produce a new array of size $n/2^{i-1}$. Here, each element in these arrays is touched exactly once, copied to another array, and never touched again in this level. Namely, each data item is accessed and manipulated exactly once. Therefore, the time complexity of merging two arrays of size $n/2^i$ into one array of size $n/2^{i-1}$ is $\Theta(n/2^{i-1})$ time.

In other words, the contribution of a subarray of size $n/2^i$ to the time complexity is $\Theta(n/2^i)$. Now, let us again observe Fig. 3.3. In level i, we have 2^i subarrays of size $n/2^i$, and each subarray contributes $\Theta(n/2^i)$ to the time complexity. Thus, in total, the time complexity at level i is $\Theta(n)$. We already know that the number of levels is $\lceil \log n \rceil$. Hence, the total time complexity of merge sort is $\Theta(n \log n)$.

As observed in Sect. 5, the function $\log n$ increases much more slowly than the function n. That is, as compared to the time complexity $\Theta(n^2)$ of the bubble sort method, the time complexity $\Theta(n \log n)$ of merge sort is quite a bit faster. Moreover, the running time of merge sort is independent of the ordering of the data, and it is easy to implement as a stable sort.

Exercise 27 ☕☕ Describe the points to which you should pay attention when you implement the merge sort method as a stable sort.

Analysis of quick sort. In the practical sense, quick sort is quite a fast algorithm and, in fact, is used in real systems. However, since quick sort has some special properties when it is applied, we have to understand its behavior.

The space complexity of quick sort is easy to determine. It requires a few variables, but large additional memories are not needed if we swap data in the array itself (we sometimes call this method "in place"). Therefore, excluding the input array, the space complexity is $O(1)$. In general, the quick sort method is not stable. (For example, in the partition procedure described in this book, it finally swaps the pivot and $a[i + 1]$. In this case, when $a[i + 1] = a[i + 2]$, their ordering is swapped and the resulting array is no longer stable.)

Then, how about time complexity? In the quick sort method, the algorithm first chooses the pivot and then divides the array into two parts by comparing the data items to this pivot. The running time depends on the size of these two parts. Therefore, the dexterity of this choice of pivot is the key point for determining the time complexity of the quick sort method. We consider this part step by step.

Ideal case. When does the algorithm run ideally, or as fast as it can? The answer is "when the pivot is always the median in the array." In this case, the array is divided into two parts of the same size. Then, we can use the same analysis as we used for merge sort, and the number of levels of recursive calls is $\Theta(\log n)$. The point in which it differs from the merge sort algorithm is that quick sort does not need to merge two subarrays after recursive calls. When the algorithm proceeds to the last case after repeating the procedure of dividing by pivots, it does not merge the subarrays. Therefore, it is sufficient to consider the cost of the partition of the array. When we consider its time complexity from the viewpoint of the algorithm, it is not so easy to analyze it. However, as in the case of merge sort, it is easy from the viewpoint of the data in the array. Each element in the current subarray is compared to the pivot exactly once, and swapped if necessary. Therefore, the time complexity of the division of an array of size n' is $\Theta(n')$. Thus, the total time complexity is $\Theta(n \log n)$, the same as that of merge sort.

Worst case. Now, we consider the worst case for the quick sort algorithm. This case occurs when the algorithm fails to choose the pivot correctly. Specifically, the selected pivot is the maximum or minimum value in the array. Then, this "division" fails, and the algorithm obtains one array of size 1 that contains only the pivot and another array that contains all the remaining elements. We consider the time complexity of this worst case. Using the first pivot, the array of size n is divided into two subarrays of size 1 and $n - 1$. Then, the number of comparison operations is $n - 1$ since the pivot is compared to the other $n - 1$ elements. Next, the algorithm repeats the same operation for the array of size $n - 1$; it performs $n - 2$ comparison operations. Repeating this process, in total, the number of comparison operations is $(n - 1) + (n - 2) + \cdots + 2 + 1 = n(n - 1)/2$. Therefore, its running time is $\Theta(n^2)$, which is as same as that of the bubble sort algorithm.

One might think that this is an unrealistic case. In the algorithm above, we chose the pivot from the last element in the array, which is not a good idea. For this algorithm, suppose that the input array is already sorted. Then, certainly, we have this sequence of the worst case. Thus, quick sort, although its implementation is easy, is not at all quick.

Analysis of randomized quick sort. The key point in the quick sort method is how to choose the pivot in a given array. How can we do this? In this book, we recommend using "random selection." Specifically, for a given array, choose the pivot *randomly*. The implementation of this randomized quick sort is quite simple:

- From the given array, select an element at random.
- Swap this selected element and the last element in this array.
- Perform the previous algorithm that uses the last element as the pivot.

Although this is a simple trick, it produces a great result.

If we select the pivot randomly, the worst case may occur with some probability. That is, we may randomly select the maximum or minimum element in the array. This probability is not zero. Some readers may worry about this worst case. However, we can *theoretically* prove that this probability is negligible. The following theorem is interesting in the sense that it proves that a simple randomized algorithm is powerful and attractive.

Theorem 8 *The expected running time of the randomized quick sort algorithm is* $O(n \log n)$.

Proof In general sorting algorithms, we count the number of comparison operations between two elements in the array to determine its running time. (More precisely, we regard the number of operations to be proportional to the running time.) We consider the probability that the parts of any pair of elements in the array are compared with each other in the randomized quick sort. The expected number of comparison operations is the summation of the probabilities for all possible pairs.

We first consider the array $a[1], \ldots, a[n]$ after sorting this array. Select any two elements $a[i]$ and $a[j]$ with $i < j$. The crucial point is that their location before sorting or in the input array is not important. Now, we consider the probability that $a[i]$ and $a[j]$ are compared with each other in the algorithm. We turn to the behavior of the quick sort method: The algorithm picks up one element as the pivot and divides the array by comparing each element in the array to the pivot. That is, $a[i]$ and $a[j]$ are compared with each other if $a[i]$ and $[j]$ are in the same array, and one of $a[i]$ and $a[j]$ is selected as the pivot. On the other hand, when are $a[i]$ and $a[j]$ not compared with each other? In the case where another element $a[k]$ between $a[i]$ and $a[j]$ is selected as the pivot before $a[i]$ and $a[j]$, and $a[i]$ and $a[j]$ are divided into the other arrays by comparing them with the pivot $a[k]$.

From this observation, regardless of the first positions of this pair in the input array, among the ordered data $a[i]$, $a[i+1]$, …, $a[j-1]$, $a[j]$ of size $j-i+1$, the probability that $a[i]$ and $a[j]$ are compared with each other in the algorithm is exactly equal to the probability that either $a[i]$ or $a[j]$ is selected as the pivot before any element $a[k]$ with $i < k < j$. The algorithm selects the pivot uniformly at random, and thus, this probability is equal to $2/(j-i+1)$. That is, the pair of $a[i]$ and $a[j]$ is compared with probability $2/(j-i+1)$, and this probability is independent of how i and j are selected.

Taking the sum of these probabilities, we can estimate the expected value of the number of comparison operations. That is, the expected number of times two elements are compared is given as

$$\sum_{1 \le i < j \le n} \frac{2}{j-i+1} = \sum_{1 \le i < n} \sum_{i < j \le n} \frac{2}{j-i+1}.$$

Since $(j-i+1)$ takes from 2 to $n-i+1$ for each i, we have

$$\sum_{1\le i<n}\sum_{i<j\le n}\frac{2}{j-i+1}=\sum_{1\le i<n}\sum_{1\le k\le n-i}\frac{2}{k+1}<\sum_{1\le i<n}2H(n-i)$$
$$\le\sum_{1\le i\le n}2H(n)=2nH(n).$$

Here, $H(n)$ denotes the nth harmonic number, and we already know that $H(n) = O(\log n)$.[5] Thus, the expected value of the running time of the randomized quick sort is $O(n \log n)$. □

In the previous proof, we use the **linearity of expectation**. It is quite strong property of expected number. To readers who are not familiar with probability, we here give the definition of expected number and give a note about the property:

Definition of expected number

Suppose that you have n distinct events and, for each $i = 1, 2, \dots, n$, the value x_i is obtained with probability p_i. Then, the **expected value** is defined by $\sum_{i=1}^{n} p_i x_i$. For example, when you roll a die, you obtain one of the values from 1 to 6 with probability 1/6. The expected value is $\sum_{i=1}^{6} i/6 = (1 + 2 + \cdots + 6)/6 = 21/6 = 3.5$.

Linearity of expectation

In the argument in the previous proof, we have a small gap from the viewpoint of probability theory. In fact, a more detailed discussion is needed, which is omitted here. More precisely, we use the following property implicitly: **The expected value of a sum of random variables is equal to the sum of the individual expectations**. This property is not trivial, and we need a proof. However, we will use this property as a fact without a proof. The reader may check the technical term "linearity of expectation" for further details of this notion.

In Theorem 8, we claim that the running time is $O(n \log n)$; however, the expected value is in fact bounded above by $2nH(n)$. That is, the constant factor in this $O(n \log n)$ is very small. This is twice as much as the ideal case. This fact implies that the randomized quick sort runs quite quickly in a practical sense. Of course, this value is an expected value, or average value, and it can take longer, but we rarely have such an extreme case. This "rareness" can be discussed in probabilistic theory; however, we stop here since it is beyond the scope of this book.

[5] **Changing the base of** log:
It is known that $\log_a b = \log_c b / \log_c a$ for any positive real numbers a, b, and c. This is called the change-of-base formula. Under the O notation, we can say that $\ln n = \Theta(\log n)$ and $\log n = \Theta(\ln n)$. That is, every logarithmic function $\log_c$ is the same as the other logarithmic function $\log_{c'}$ up to a constant factor.

Efficient implementation of quick sort

At a glance, since the idea itself is simple, it seems to be easy to implement the quick sort method. However, there are many tricky inputs, such as "the array of many elements of the same value," and it is not an easy task to build a real program that behaves properly for any input. I checked and found that there are two different styles for implementing the partition procedure in the quick sort method. The first, which is simple and easy to understand, and works well, is described in this text. I borrowed this implementation style from the well-known textbook "Algorithm Introduction" used at the Massachusetts Institute of Technology. It is easy to follow that when this style is used, the algorithm works well for any input. The other implementation style was proposed in many early studies in the literature on the quick sort method. As mentioned, it is known that some program codes in the early literature contain errors, which implies that it is not very easy to implement this style correctly. Of course, in addition to the early studies, the implementation based on this style has been well investigated, and the correct implementation has now been stated and established. I do not explain the details in this text since it is a bit more complicated than the style proposed in this book.

However, essentially, both implementation styles have the same framework, and the analysis and the running time are the same from the theoretical viewpoint. In both, the real program code is short and the algorithm runs quickly. Experimental results imply that the latter (or classic) style is a bit faster. The background and real compact implementations are discussed in "Algorithms in C." Readers may check the book list in Chap. 7 when they have to implement practical algorithms.

Exercise 28 ☕☕☕ As explained, quick sort is a truly quick algorithm. However, it requires some overhead processes as its background task when it makes a recursive call. Therefore, when there are quite a few elements, simpler algorithms, such as bubble sort, run faster. Consider how one can use the advantages of both these sorting algorithms.

3.3.5 Essential Complexity of Sorting

In this section, three sorting algorithms are introduced. The time complexities of the bubble sort, merge sort, and quick sort methods are measured by the number of comparison operations, and we obtain $\Theta(n^2)$, $\Theta(n \log n)$, and $O(n \log n)$ for each of them, respectively. First, any algorithm has to read the entire input data as an array, and each element in the array should be compared to some other data at least once. Therefore, no algorithm can sort n elements in less than $\Theta(n)$ time.

As learnt in Sect. 1.5, the function $\log n$ increases much more slowly than the function n. Therefore, the improvement from $\Theta(n^2)$ to $\Theta(n \log n)$ is drastic, and

$\Theta(n \log n)$ is not very much slower than $\Theta(n)$. Nevertheless, there is some doubt whether we can improve on this gap or not.

Interestingly, in a sense, we cannot further improve sorting algorithms. The following theorem is known.

Theorem 9 *When a comparison operation is used as a basic operation, no algorithm can solve the sorting problem with fewer than $\Omega(n \log n)$ operations.*

As for the lower bound of the searching problem in Theorem 7, we need some knowledge of information theory to discuss it in detail. In this book, we do not go into the details, and give a brief intuitive proof.

Proof To simplify, we assume that all elements in the array $a[]$ are distinct; that is, we assume $a[i] \neq a[j]$ if $i \neq j$. We first observe that one comparison operation distinguishes two different cases. That is, when we compare $a[i]$ with $a[j]$, we can distinguish two cases $a[i] < a[j]$ and $a[i] > a[j]$.

Now, we suppose that the $a[1], a[2], \ldots,$ and $a[n]$ are all distinct. We consider the number of possible inputs by considering only their ordering. For the first element, we have n different cases where it is the smallest element, the second smallest element, ..., and the largest element. For each of these cases, the second element has $n - 1$ different cases. We can continue until the last element. Therefore, the number of possible orderings of the input array is

$$n \times (n-1) \times \cdots 3 \times 2 \times 1 = n!.$$

Therefore, any sorting algorithm has to distinguish $n!$ different cases using the comparison operations.

Here, one comparison operation can distinguish two different cases. Thus, if the algorithm uses a comparison operation k times, it can distinguish at most 2^k different cases. Hence, if it distinguishes $n!$ different cases by using k comparison operations, k has to satisfy $n! \leq 2^k$. Otherwise, if k is not sufficiently large, the algorithm cannot distinguish two (or more) different inputs, and hence, it should fail to sort for one of these indistinguishable inputs.

Now, we use a secret formula called "Stirling's formula."[6] For some constant c, we use

$$n! = c\sqrt{n}\left(\frac{n}{e}\right)^n.$$

Taking the logarithm on the right-hand side, we obtain

$$\log\left(c\sqrt{n}\left(\frac{n}{e}\right)^n\right) = \log c + \frac{1}{2}\log n + n\log n - n\log e = \Theta(n\log n).$$

[6] **Stirling's formula**:
The precise Stirling's formula is $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. The error of this approximation is small, and this formula is useful for the analyses of algorithms. For the present, in a rough-and-ready manner, the approximation $n! \sim n^n$ is worth remembering for estimating computational complexity roughly in order to analyze an algorithm.

Looking at this equation carefully, we can see that, for sufficiently large n, it is almost $n \log n$. (More precisely, the other terms except $n \log n$ are relatively small as compared to $n \log n$, and hence, they are negligible.) On the other hand, $\log 2^k = k$. Therefore, taking the logarithm of the inequality $n! \leq 2^k$, we obtain $\Theta(n \log n) \leq k$. That is, no algorithm can sort n elements correctly using comparison operations less than $\Omega(n \log n)$ times. □

3.3.6 Extremely Fast Bucket Sort and Machine Model

In Theorem 9, we show the lower bound $\Omega(n \log n)$ of the running time for any sorting algorithm based on the *comparison operation*. As in the hash function in Sect. 3.2, we have to take care of the condition. In fact, we can violate this lower bound if we use a sorting algorithm not based on the comparison operation. For example, suppose that we already know that each data item is an integer between 1 and m. (This is the same assumption as that for the simple hash function in Sect. 3.2.) We note that this assumption is natural and reasonable for daily data, such as examination scores. In this case, the algorithm named **bucket sort** is quite efficient. In the bucket sort method, the algorithm prepares another array $b[1]$, $b[2]$, ..., and $b[m]$. Each element $b[j]$ stores the number of indices i such that $a[i] = j$. Then, after the algorithm reads the data once, $b[1]$ indicates the number of indices with $a[i] = 1$, $b[2]$ indicates the number of indices with $a[i] = 2$, and so on. Therefore, now the algorithm outputs $b[1]$ 1s, $b[2]$ 2s, ..., $b[i]$ is, ..., and $b[m]$ ms. (Or the algorithm can write back to $a[]$ from $b[]$.) That is the end of the sorting. The code of the algorithm is described below.

```
Algorithm 24: BucketSort($a$)
  Input : An array $a[1], a[2], \ldots, a[n]$. Each $a[i]$ is an integer with $1 \leq a[i] \leq m$ for some
          $m$.
  Output: Sorted data.
 1 prepare an array $b[]$ of size $m$;
 2 for $j \leftarrow 1, 2, \ldots, m$ do
 3     $b[j] \leftarrow 0$ ;                                  /* initialize by 0. */
 4 end
 5 for $i \leftarrow 1, 2, \ldots, n$ do
 6     $b[a[i]] \leftarrow b[a[i]] + 1$;
 7 end
 8 for $j \leftarrow 1, 2, \ldots, m$ do
 9     if $b[j] > 0$ then output $j$ $b[j]$ times;
10 end
```

Exercise 29 ☕☕ In line 9, the algorithm above outputs only the contents of $a[]$ in increasing order. Modify the algorithm to update $a[]$ itself.

Now, we consider the computational complexity of bucket sort. First, we consider space complexity. If we prepare the array $b[]$ straightforwardly, it takes $\Theta(m)$. We may reduce some useless space if the data are sparse, but this case does not concern us. In any case, $O(m)$ space is sufficient. Next, we consider time complexity. It is clearly achievable by $\Theta(n + m)$ with naive implementation.

Quick sort is the sort algorithm that can be applied to any data if the comparison operator is well-defined on any pair of data elements. That is, if two data are comparable and the ordering among the whole data is consistent, we can sort the data by using the quick sort method. For example, we can sort even strings, images, and movies if their orderings are well-defined (and it is not difficult to define a reasonable comparison operator according to the purpose of sorting.) That is, the quick sort method is widely applicable to many kinds of data. The bucket sort method has limitations as compared to quick sort; however, in those cases where it can be applied, bucket sort is quite useful since it runs reasonably fast. For example, imagine that you have to sort several millions of scores. If you know that all the scores are integers between 1 and 100, since $m \ll n$, bucket sort is the best choice. As you can imagine from the behavior of the algorithm, it runs almost in the same time as that taken for just reading data.

We note that bucket sort explicitly uses the property "computer can read/write $b[i]$ in memory in a unit time," which is the property of the RAM model.

Exercise 30 ☕☕☕ We consider the following algorithm named **spaghetti sort**:

```
Algorithm 25: SpaghettiSort($a$)
  Input  : An Array $a[1], a[2], \ldots, a[n]$.
  Output: Sorted array $a[]$.
for $i \leftarrow 1, 2, \ldots, n$ do
    cut dry spaghetti strands to length $a[i]$;
end
bunch all the spaghetti strands in your left hand and stand them on your table;
for $i \leftarrow 1, 2, \ldots, n$ do
    select the longest spaghetti strand in your right hand and put it on your table;
end
output the lengths of these spaghetti strands from one end to the other end;
```

Estimate the time complexity of this algorithm. Theoretically, it seems that this algorithm runs fast; however, you would not like to use it. Why is this algorithm not in practical use?

Chapter 4
Searching on Graphs

Abstract The notion of a *graph* is a mathematical concept that was introduced to represent a network of objects. The structure of a graph is very basic, and many problems can be modeled using graphs. In this chapter, we learn basic algorithms for searching a graph.

What you will learn:

- Searching algorithms for graphs
- Reachability
- Shortest path problem
- Search tree
- Depth-first search
- Breadth-first search
- Dijkstra's algorithm

4.1 Problems on Graphs

Have you ever seen a Web page that is public on the Internet on your personal computer or mobile phone? Of course, you have. The Web page has a useful function that is called a **hyperlink**, and you can jump to another Web page to see it just by a click. Suppose that when you open your usual Web page, you happen to think "I'd like to see that Web page I saw last week..." You remember that you reached that Web page by traversing a series of hyperlinks; however, you have no memory of the route you traversed. This is nothing out of the ordinary.

Hyperlinks make up a typical graph structure. Each Web page corresponds to a vertex in the graph, and each hyperlink joins two vertices. We do not know whether the entire graph is connected or disconnected. **Searching problems** for a graph are the problems that ask whether two vertices on a graph are reachable, that is, whether

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_4

there is a route from one vertex to another on the graph. In this section, we consider the following three variants of the searching problem for a graph.

[Problem 1] Reachability problem:

Input: A graph G containing n vertices $1, 2, \ldots, n$. We assume that the start vertex is 1 and the goal vertex is n.
Problem: Check whether there is any way that we can reach n from 1 along the edges in G. The route is not important. The graph may be disconnected. If we can reach n from 1 on G, output "Yes"; otherwise, output "No."

[Problem 2] Shortest path problem:

Input: A graph G containing n vertices $1, 2, \ldots, n$. We assume that the start vertex is 1 and the goal vertex is n.
Problem: We also assume that there is no cost for passing through each edge; or we can also consider that each edge has a unit cost. Then, the problem is to find the length of the shortest path from 1 to n. Precisely, the length of a path is the number of edges along it. If n is not reachable from 1, we define the length as ∞.

[Problem 3] Shortest path problem with minimum cost:

Input: A graph G containing n vertices $1, 2, \ldots, n$. We assume that the start vertex is 1 and the goal vertex is n.
Problem: We also assume that each edge has its own positive cost. Then, the problem is to find the shortest path from 1 to n with minimum cost. The cost of a path is defined by the summation of the cost of each of its edges. If n is not reachable from 1, we define the cost as ∞.

It is easy to see that the above problems are all familiar in our daily life; they appear in car navigation and route finder systems and are solved every day. We learn how we can handle these problems, and what algorithms and data structures we can use to solve them.

4.2 Reachability: Depth-First Search on Graphs

First, we consider Problem 1, which concerns the reachability in a given graph. For a given graph with n vertices $1, 2, \ldots, n$, we assume that the edge set is given by the adjacency set. That is, the set of neighbors of the vertex i is represented

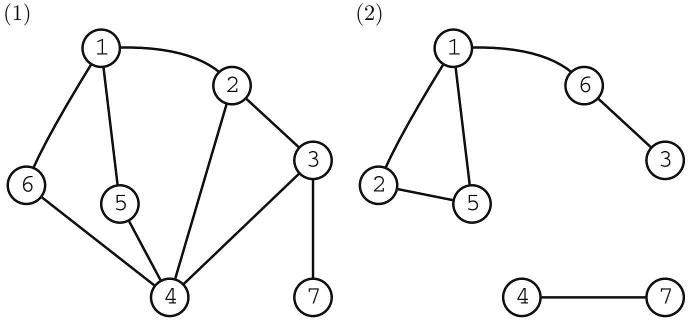


Fig. 4.1 Two examples of a graph

by $A[i, 1]$, $A[i, 2]$, ... Hereafter, to simplify, we denote by d_i the number of elements in the neighbor set of vertex i. (This number d_i is called the **degree** of vertex i.) That is, for example, the set of neighbors of vertex 2 is $A[2, 1], A[2, 2], \ldots, A[2, d_2]$. The number d_i can be computed by checking whether $A[i, j] = 0$ for each $j = 1, 2, \ldots, d_i$ in $O(d_i)$ time. Thus, we can compute it every time we need to; alternatively, we can compute it a priori and remember it in the array $d[1], d[2], \ldots, d[n]$. The notation d_i is used for convenience, and we do not consider the details of the implementation of this part.

Example 9 Consider the two graphs given in Fig. 4.1. The adjacency set A_1 of graph (1) on the left and the adjacency set A_2 of graph (2) on the right are given as follows (the numbers are sorted in increasing order so that we can follow the algorithm easily).

$$A_1 = \begin{pmatrix} 2\,5\,6\,0\,0\,0 \\ 1\,3\,4\,0\,0\,0 \\ 2\,4\,7\,0\,0\,0 \\ 2\,3\,5\,6\,0\,0 \\ 1\,4\,0\,0\,0\,0 \\ 1\,4\,0\,0\,0\,0 \\ 3\,0\,0\,0\,0\,0 \end{pmatrix}, \; A_2 = \begin{pmatrix} 2\,5\,6\,0\,0\,0 \\ 1\,5\,0\,0\,0\,0 \\ 6\,0\,0\,0\,0\,0 \\ 7\,0\,0\,0\,0\,0 \\ 1\,2\,0\,0\,0\,0 \\ 1\,3\,0\,0\,0\,0 \\ 4\,0\,0\,0\,0\,0 \end{pmatrix}.$$

We use the following degree sequences for each graph: For graph (1), we have $d[1] = 3$, $d[2] = 3$, $d[3] = 3$, $d[4] = 4$, $d[5] = 2$, $d[6] = 2$, and $d[7] = 1$. For graph (2), we have $d[1] = 3$, $d[2] = 2$, $d[3] = 1$, $d[4] = 1$, $d[5] = 2$, $d[6] = 2$, and $d[7] = 1$. We assume that these arrays are already initialized before performing our algorithms.

How can you tackle Problem 1 if you do not have a computer and you have to solve it by yourself? Imagine that you are at vertex 1, and you have a table $A[1, j]$ that indicates where you can go next. If I were you, I would use the following strategy.

(1) Pick an unvisited vertex in some order and visit it.
(2) If you reach vertex n, output "Yes" and stop.
(3) If you check all reachable vertices from vertex 1 and never reach vertex n, output "No" and stop.

How can we describe this strategy in the form of an algorithm? First, we prepare a way of checking whether the vertex has already been visited or not. To do this, we use an array $V[]$ that indicates the following. Array $V[i]$ is initialized by 0, and vertex i has not yet been visited if $V[i] = 0$. On the other hand, if $V[i] = 1$, this means that the vertex i has already been visited by the algorithm. That is, we have to initialize it as $V[1] = 1$, $V[2] = V[3] = \cdots = V[n] = 0$. Next, we use the variable i to indicate where we are. That is, we initialize $i = 1$, and we have to output "Yes" and halt if $i = n$ holds. Based on the idea above, the procedure at vertex i can be described as follows (we will explain what DFS means later).

```
Algorithm 26: DFS($A, i$)
  Input  : Adjacency set $A[]$ and index $i$
  Output: Output "Yes" if it reaches vertex $n$
  $V[i] \leftarrow 1$;
  if $i = n$ then
      output "Yes" and halt;
  end
  for $j \leftarrow 1, 2, \ldots, d_i$ do
      if $V[A[i, j]] = 0$ then
          DFS($A, A[i, j]$);
      end
  end
```

We note that this DFS algorithm is a recursive algorithm. The main part of this algorithm, which is invoked at the first step, is described below.

```
Algorithm 27: First step of the algorithm for the reachability problem DFS-main($A$)
  Input  : Adjacency set $A[]$
  Output: Output "Yes" if it reaches vertex $n$; otherwise, "No"
  for $i = 2, \ldots, n$ do
      $V[i] \leftarrow 0$;
  end
  DFS($A$, 1);
  output "No" and halt;
```

Structure of an algorithm

Most algorithms can be divided into two parts containing respectively the "general process" and the "first step," which contains initialization and invoke the first process. This is conspicuous, particularly in recursive algorithms. In the algorithms up to this point in this book, the first step has been written in one line; we describe this in the main text. However, in general, we need some preprocessing including the initialization of variables. Hereafter, the algorithms named "$*$-main" represent this first step, and the general process is located in the algorithm that has a name that does not include "main."

This is the end of the algorithm's description. Are you convinced that this algorithm works correctly? Somehow, you do not feel sure about the correctness, do you? Since this algorithm is written in recursive style, you may not be convinced without considering the behavior of the algorithm more carefully. First, to clarify the reason why you do not feel sure, we consider the conditions when the algorithm halts. The algorithm halts when one of two cases occurs:

- When $i = n$, that is, it reaches to the goal vertex. Then, it outputs "Yes" and halts.
- In the DFS algorithm, the process completes the recursive calls in line 4 without outputting "Yes." Then, the DFS-main outputs "No" in line 5 and halts.

This would be all right in the first case. The algorithm certainly reaches the goal vertex n through the edges from the start vertex 1. The case that causes concern is the second case. Is the answer "No" always correct? In other words, does it never happen that the algorithm outputs "No" even though a path exists from vertex 1 to vertex n? This is the point. To understand this in depth, we first examine two typical cases. The first case is where the algorithm outputs "Yes."

Example 10 We perform the algorithm at vertex 1 in graph (1) in Fig. 4.1. (We denote the array V by $V_1[]$ to distinguish the second example.) First, Algorithm 27 performs the initialization in lines 1 to 3 and calls the procedure DFS$(A_1, 1)$. In DFS$(A_1, 1)$, DFS$(A_1, A_1[1, 1] = 2)$ is again called in line 7. That is, it moves to the neighbor vertex 2 from vertex 1. Then, Algorithm 26 is applied to vertex 2, and we have the visited mark by setting $V_1[2] = 1$. Since $2 \neq n(= 7)$, the process moves to line 5. First, since $V_1[A_1[2, 1]] = V_1[1] = 1$, it confirms that vertex 1 had already been visited; thus, it does not call DFS (in line 7) for this vertex 1. Next, since $V_1[A_1[2, 2]] = V_1[3] = 0$, it finds that vertex 3 has not already been visited, calls DFS$(A_1, A_1[2, 2])$ in line 7, moves to vertex 3, and so on. Namely, it continues to check $V_1[]$ and does not visit it if it has already been visited; otherwise, it visits this vertex. In this graph, the algorithm visits in the order

$$\text{vertex } 1 \rightarrow \text{vertex } 2 \rightarrow \text{vertex } 3 \rightarrow \text{vertex } 4 \rightarrow \text{vertex } 5 \rightarrow \text{vertex } 6 \rightarrow \text{vertex } 7$$

and outputs "Yes" when it visits the last vertex 7.

Next, we turn to the second example, that is, the "No" instance.

Example 11 We perform the algorithm at vertex 1 in graph (2) in Fig. 4.1. (We denote the array V by $V_2[]$ this time.) As before, the algorithm visits vertices 1, 2, and 5. When it visits vertex 5, it finds it has no more unvisited vertices from vertex 5. Then, the searching process in DFS$(A_2, A_2[2, 2])$ ends and returns to vertex 1. This time, the DFS$(A_2, A_2[1, 1])$ is called in line 7 of Algorithm 26. In the next step, DFS$(A_2, A_2[1, 2])$ is not called because $V_2[A_2[1, 2]] = V_2[5] = 1$. Then, DFS$(A_2, A_2[1, 3])$ is called since $V_2[A_2[1, 3]] = V_2[6] = 0$, and the algorithm moves to vertex 6. Then, it moves to vertex 3 from vertex 6 and finds that there are no longer any unvisited vertices from vertex 3. Thus, this procedure call returns to vertex 1; no more unvisited vertices exist, and hence, the output is "No."

You can understand this case in depth if you follow these "Yes" and "No" examples. Now we turn to the following case. The algorithm outputs "No" for some reason, although it is possible to reach from vertex 1 to vertex n. Does this case never happen? The answer is *never*. It is not trivial, and so we here show the proof of this fact.

Theorem 10 *The DFS algorithm for the reachability problem is certain to output "Yes" when it is possible to reach from vertex 1 to vertex n; otherwise, the output is "No." In other words, this algorithm works correctly.*

Proof First, we check that this algorithm always halts in finite time regardless of its output. It is clear that the algorithm always halts after two **for** statements. Therefore, we focus on the recursive call for DFS. There are two recursive calls that the algorithm makes (line 7 in DFS and line 4 in DFS-main). In both cases, DFS$[A, i]$ is called for some i satisfying $V[i] = 0$. Then, immediately afterward, $V[i] \leftarrow 1$ is performed at line 1 in DFS. Thus, once this recursive call has been made for index i, the recursive call for DFS will be never performed for this i again. On the other hand, the integer variable i takes its value with $1 \le i \le n$. Therefore, the number of recursive calls in this algorithm is bounded above by n. Thus, this algorithm always halts with at most n recursive calls. The algorithm can halt at line 3 in DFS and at line 5 in DFS-main, and hence, the algorithm outputs either "Yes" or "No" when it halts.

We first consider the case where the algorithm outputs "Yes." In this case, the correctness of the algorithm is easy to see. From the construction of the algorithm, we can join the variables in the recursive calls of DFS and obtain the route from vertex 1 to vertex n.

Next, we assume that the algorithm outputs "No" as the answer. Here, we have the problem "Is this answer 'No' certainly correct?" We prove the correctness by contradiction. We assume that there exists a graph G that includes a path joining vertices 1 and n, and the DFS-main algorithm fails to find it and outputs "No." Let P be the set of visited vertices during the journey that the algorithm traverses on G. Then, P contains vertex 1 by definition and does not contain vertex n by assumption. On the other hand, since we assume that we can reach vertex n from vertex 1, there is a path Q joining vertices 1 and n (if there are two or more, we can select any one of them). Path Q starts from vertex 1 and ends at vertex n. We have common elements that belong to both P and Q; at least, vertex 1 is one of the common elements. Among the common elements of P and Q, let i be the closest vertex to n in Q. Since i belongs to P and P does not contain n, i is not n. Let k be the next vertex of i on Q (we note that k itself can be n). Then, by the assumption, P contains i, but does not contain k (Fig. 4.2).

Now, P is the set of vertices visited by DFS. Thus, during the algorithm, DFS(A, i) should be called. In this procedure, for each neighbor $A[i, j](j = 1, 2, \ldots, d_i)$ of vertex i, the algorithm calls DFS$(A, A[i, j])$ if $V[A[i, j]] = 0$, that is, if vertex $A[i, j]$ has not already been visited. Vertex k is one of the neighbors of i, and hence, $A[i, j'] = k$ should hold for some j', which means that DFS(A, k) should be called for this j'. This is a contradiction. Thus, such a vertex k does not exist.

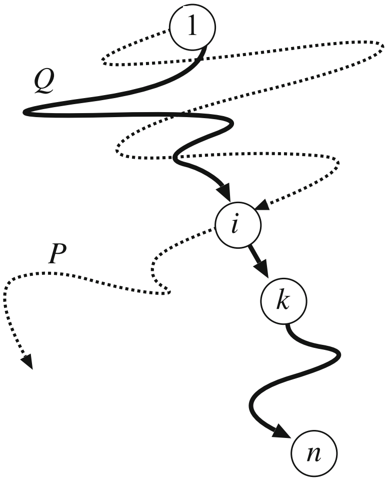


Fig. 4.2 Correctness of the algorithm for the reachability problem

Therefore, when vertex n is reachable from vertex 1, the DFS algorithm should find a path from 1 to n and output "Yes." □

We here also state a theorem about the time complexity of the DFS algorithm.

Theorem 11 *For a graph of n vertices with m edges, the DFS algorithm solves the reachability problem in $O(n+m)$ time.*

Proof By careful observation of the DFS algorithm, it can be seen that the crucial point is the estimation of the time complexity of the for loop. We can consider this part as follows. Let us consider an edge $\{i, j\}$. For this edge, the algorithm performs the for loop at vertex i and another for loop at vertex j. This edge had never been touched by another for loop. Therefore, the total time complexity consumed by the for loops for each vertex can be bounded above the number of edges up to constant. (A similar idea can be found in the proof of Theorem 3.) Therefore, summing the total running time $O(n)$ for each vertex, the entire running time is $O(n+m)$. □

4.2.1 Search Trees and Depth-First Search

Finally, we mention that why the search method introduced in this section is called DFS. DFS is an abbreviation of "depth-first search." Then, what is searching in "depth first" style? We have to learn about "search trees" to know what "depth first" means. We here focus on the ordering of visits. What property does the ordering of visited vertices by the DFS algorithm have? Hereafter, we assume that the input graph is connected since the DFS algorithm visits only the vertices in a connected component of the input graph.

At the beginning of the search, we start from vertex 1. Therefore, each of the other vertices i with $i \neq 1$ is visited from another vertex if the graph is connected. Let j be the vertex from which the algorithm first visits vertex i. That is, the DFS algorithm was staying at vertex j and visits vertex i the first time through edge $\{j, i\}$. For each vertex from 2 to n, we can consider such an edge for visiting each of them the first time. Vertex 1 is the only exception, having no such edge, because it is the vertex from which the DFS algorithm starts. Thus, the number of such edges for visiting the first time is $n - 1$, and the graph is connected. Therefore, using Theorem 4, these edges form a tree in the input graph.

In this tree, vertex 1 is a special vertex. That is, the DFS algorithm starts from vertex 1 and visits all the vertices along the edges in this tree. This tree is called the **search tree**, and this special vertex 1 is called the **root** of the search tree. It is useful that we assign the number to each vertex in the tree according to the distance from the root. This number is called the **depth** of the vertex. The depth is defined by the following rules. First, the depth of the root is defined by 0. Then, the depth of the neighbors of the root is defined by 1. Similarly, the depth of a vertex is i if and only if there is a path of length i from the root to the vertex on the tree.

We now consider the ordering of the visits of the DFS algorithm on the search tree from the viewpoint of the depth of the tree. The algorithm starts searching from vertex 1 of depth 0 and selects one neighbor (vertex 2 in Fig. 4.1(1)) of root 1. The next search target is the neighbor of this vertex 2 of depth 1. In Fig. 4.1(1), vertex 3 is selected as the next vertex of vertex 2, and this vertex 3 has depth 2, and so on. The search tree obtained by the DFS algorithm on the graph in Fig. 4.1(1) is described in Fig. 4.3. The search tree of the DFS algorithm is drawn in bold lines. Vertex 3 is visited as of depth 2, and vertices 4 and 7 are both visited as of depth 3. On the other hand, we can consider another method of searching; for example, in Fig. 4.3, after visiting vertex 2, the algorithm may visit vertices 5 and 6. Namely, the algorithm first checks all the neighbors of the root before more distant vertices. However, the DFS algorithm, although there exist unvisited vertices of smaller depth, gives precedence in its search to more distant or deeper vertices. This is the reason why this algorithm is called "depth-first search."

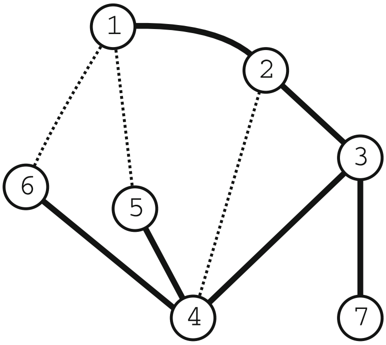


Fig. 4.3 Search tree in the graph in Fig. 4.1(1) obtained by applying the DFS algorithm

Of course, the other method of searching, that is, searching all the vertices of depth 1 after checking the vertex of depth 0, before checking more distant vertices, gives us another searching option. While DFS prefers the depth direction for visiting the vertices in the tree, the second search method prefers the width or breadth direction for searching. This method is called **breadth-first search**; however, we will discuss the details of this algorithm in the next section.

Exercise 31 ☕☕☕ In the example in Fig. 4.3, the DFS algorithm outputs "Yes" and halts when it reaches vertex 7. We now remove the line for output "Yes" from the algorithm and run it. Then, the algorithm, in any case, halts at output "No" after visiting all the vertices in the graph. Now, when we run this modified algorithm on the graph given in Fig. 4.1(1), what search tree do we obtain in the graph?

4.3 Shortest Paths: Breadth-First Search on Graphs

Next, we turn to Problem 2, which is to find the shortest path. Since it is trivial when $n = 1$, hereafter we assume that $n > 1$. This time, we have to find not only reachable vertices, but also the length of the shortest path from the start vertex 1 to the goal vertex n.

To consider how to solve this problem, imagine that we are inside vertex 1. A natural idea is to explore the neighbor vertices of distance 1 from vertex 1 since they are reachable from the vertex 1 directly. If we find the goal vertex n in the set of vertices of distance 1 from vertex 1, we are done. Otherwise, we have to check the set of vertices of distance 2 from vertex 1 from each neighbor of vertex 1, and so on. In this manner, we can find the shortest path from vertex 1 to the goal vertex n. Therefore, briefly, we can solve the problem as follows.

(1) Visit each of the unvisited vertices in the order of their proximity to the start vertex 1.
(2) If we arrive at the goal vertex n, output the distance to it and halt.
(3) If we visit all reachable vertices and still do not reach the goal vertex n, output ∞ and halt.

In comparison to the reachability check, the key point is that we seek from the vertices closer to the start vertex 1. We now consider the search tree for this algorithm. The root is still vertex 1, which is not changed. The algorithm first sweeps all the vertices that are neighbors of 1, and hence, are connected to the root on the search tree; hence, they have depth 1 on the search tree. Similarly, the algorithm next checks all the vertices that are neighbors of the vertices of distance 1 from the root or of depth 1 in the search tree, and hence, each is connected to some vertex of distance 1. That is, the next vertices are of distance 2 from the root and also of depth 2 on the search tree. Therefore, each vertex of depth i on the search tree is of distance i from the root on the original graph.

At a glance, one may think that we need a complicated data structure to implement this algorithm; however, this is not the case. At this point, we have to think carefully. Then, we can see that we can achieve this ordering of searching easily using the queue data structure. That is, the algorithm first initializes the queue with the start vertex 1. After that, the algorithm picks up the top element in the queue, checks whether the vertex i is n or not, and pushes all unvisited neighbors of i into the queue if i is not n.

We consider the general situation of this queue. From the property of the queue, we can observe the following lemma.

Lemma 12 *For each $d = 0, 1, 2, \ldots$, the algorithm will not process the vertices of distance greater than d before processing all vertices of distance d.*

We can prove the lemma precisely by an induction on d; however, the following intuitive explanation is sufficient in this book. We assume that there are some vertices of distance d in the front part of the queue. (Initially, we have only one vertex 1 of distance 0 in the queue.) If some vertices of distance d have been searched, their unvisited neighbors of distance $d + 1$ are put in the latter part of the queue. In this situation, there are no vertices of distance less than d, and there are no vertices of distance greater than $d + 1$. It is not very easy to prove this fact formally, but it is clear when you consider the statement in Lemma 12. The correctness of this search algorithm relies on the property stated in Lemma 12. The queue data structure is effective for achieving this property. Once we have initialized the queue by vertex 1 of distance 0, the above property immediately appears when the queue is used.

Example 12 We consider what happens on the graph on the left in Fig. 4.1 when we apply the algorithm. First, the queue is initialized by vertex 1. The algorithm picks it up, checks it, and confirms that it is not the goal vertex 7. Therefore, the algorithm pushes the unvisited vertices 2, 5, and 6 into the queue. Next, it picks up vertex 2 and confirms that it is not vertex 7. Thus, it pushes the unvisited neighbors 3 and 4 of 2 into the queue (after the vertices 5 and 6). Then, it picks up the next vertex 5, checks whether it is vertex 7 or not, and pushes the unvisited neighbors to the end of the queue. The transitions of the queue and the depths of the vertices are depicted in Fig. 4.4(1). The vertices are described by the numbers in the boxes, and the top of the queue is the left-hand side. The numbers over the boxes are the depths of the vertex, that is, the distance from the root. The bold lines in Fig. 4.4(2) show the resulting search tree.

We here mention one point to which we should pay attention. Let us consider the viewpoint at vertex i. Once this vertex i is put into the queue, it will proceed in the queue, and in its turn, it will be checked as to whether it is the vertex n or not. The point to which we have pay attention is that it is inefficient to put this vertex i into the queue two or more times. (In fact, this can be quite inefficient in some cases. If you are a skillful programmer, this is a nice exercise to find such a case.) Let d be the distance from root 1 to vertex i. By Lemma 12, this inefficiency may occur if vertex i has two neighbors, say j and k, of distance $d - 1$.

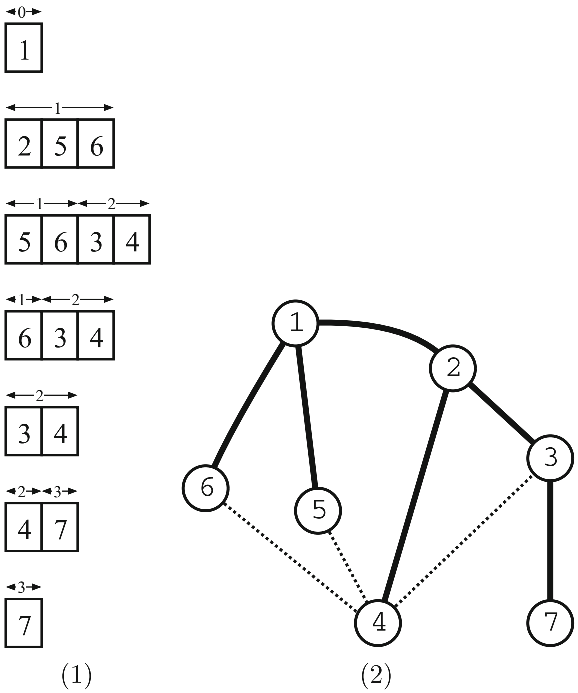


Fig. 4.4 Applying the BFS algorithm on the graph in Fig. 4.1(1): transitions of the queue and the resulting search tree

More precisely, an inefficient case may occur as follows. When two distinct vertices j and k of distance $d-1$ are both adjacent to vertex i of distance d, both vertices j and k may push vertex i into the queue. In Fig. 4.4, once vertex 4 is put into queue, we have to avoid being put it again by vertices 3, 5 and 6 as their neighbors. To avoid pushing vertex into the queue two or more times, let us prepare another array $C[]$. For each vertex i, $C[i]$ is initialized by 0. That is, if $C[i]=0$, vertex i has not yet been pushed into the queue, and we define that $C[i]=1$ means that vertex i has already been pushed into the queue. Once $C[i]$ has been set at 1, this value is retained throughout the algorithm. Each vertex ℓ should push its neighbor i into the queue if and only if i is unvisited and $C[i]=0$. When vertex i is pushed into the queue, do not forget to update $C[i]=1$. When we check the DFS algorithm carefully, we can see that we no longer need the array $V[]$ to keep the information if "the vertex has already been visited." Each vertex i in the queue, unless vertex n exists before it, will be visited at some time, and this information is sufficient, even without $V[]$.

At this time, the algorithm should output not only "Yes" or "No," but also the length of the shortest path to vertex n. Therefore, we use another array, say $D[]$, to store the length of the shortest path to each vertex i. That is, $D[i]=d$ means that we have the shortest path from vertex 1 to vertex i of length d. This value is set when the algorithm visits vertex i the first time, or when $C[i]$ is set at 1, we do not need to initialize it.

We now consider what happens if the algorithm cannot reach vertex n. With the same argument as that of the DFS algorithm, all reachable vertices from vertex 1 will be put into the queue. Therefore, if the queue becomes empty without vertex n being found, this implies that the desired answer is "∞."

Based on the above discussion, the general case can be processed as follows. Note that the queue Q is one of the arguments. (We will explain why this algorithm is named "BFS.")

```
Algorithm 28: BFS($A$, $Q$)
  Input  : Adjacent set $A[]$, queue $Q$
  Output: The (shortest) distance from vertex 1 to vertex $n$
  $i \leftarrow$ pop($Q$);
  if $i = n$ then
      output $D[i]$ and halt;
  end
  for $j \leftarrow 1, 2, \ldots, d_i$ do
      if $C[A[i, j]] = 0$ then
          push($Q$, $A[i, j]$);
          $C[A[i, j]] \leftarrow 1$ ;        /* Vertex $A[i, j]$ is put into the queue */
          $D[A[i, j]] \leftarrow D[i] + 1$ ;    /* We can reach $j$ from $i$ in 1 step */
      end
  end
```

The main part of this algorithm is as follows.

```
Algorithm 29: BFS-main($A$)
  Input  : Adjacency set $A[]$
  Output: Distance to vertex $n$ if it is reachable; otherwise, "$\infty$"
  for $i = 2, \ldots, n$ do
      $C[i] \leftarrow 0$;
  end
  $C[1] \leftarrow 1$;
  $D[1] \leftarrow 0$;
  push($Q$, 1);
  while sizeof($Q$) $\neq 0$ do
      BFS($A$, $Q$);
  end
  output $\infty$ and halt;
```

Exercise 32 ☕☕☕ In the BFS algorithm, two arrays $C[]$ and $D[]$ are used. When $C[i] = 0$, vertex i has not yet been placed into the queue, and $C[i] = 1$ indicates that vertex i had been placed into the queue. When $C[i] = 1$, the value of $D[i]$ gives us the length of the shortest path from vertex 1 to vertex n. Somehow, this seems to be redundant. Consider how can you implement these two pieces of information using one array. Is it an "improvement?"

Exercise 33 ☕☕☕ The above BFS algorithm outputs the distance of the shortest path to vertex n. However, it is natural to desire to output the shortest path itself. Modify the above BFS algorithm to output one shortest path itself. It is natural to output the path from vertex 1 to vertex n; however, it may be reasonable to output the path from vertex n to vertex 1, namely to output the path in the reverse order.

Exercise 34 ☕☕☕☕ Prove the following two claims.

(1) The BFS algorithm certainly outputs the length of the shortest path if vertex n is reachable from vertex 1 on the graph.
(2) The running time of the BFS algorithm is $O(n + m)$, where n is the number of vertices and m is the number of edges.

For the first claim, we can use an induction for the length of the shortest path. In this case, use Lemma 12 for the property of the queue. The proofs of Theorems 10 and 11 are helpful.

We here mention the name of this algorithm, “BFS.” BFS stands for **breadth-first search**. This algorithm first checks the root vertex 1 and then seeks all the vertices of depth 1, that is, all the neighbors of the root. Then, it proceeds to seek all the vertices of depth 2. In this manner, in the search tree, this algorithm proceeds to the next depth after searching all the vertices of the same depth. This can be regarded as a manner of searching that prefers to search in the “breadth” direction. Here, check again the search tree in Fig. 4.4.

4.4 Lowest Cost Paths: Searching on Graphs Using Dijkstra’s Algorithm

We finally turn to the last problem, that is, finding the path of the lowest cost. This problem is more difficult than the previous two. Since each edge has its cost, we have to consider a detour that may give a lower total cost. The representative algorithm for this problem is called **Dijkstra’s algorithm**.[1] In this section, we learn this algorithm.

Dijkstra’s algorithm is an algorithm for computing the minimum cost from the start vertex 1 to the goal vertex n. We first prepare an array $D[]$ for storing this cost, which is initialized as $D[i] = \infty$ for each i. In the final step, $D[i]$ stores the minimum cost to this vertex i from vertex 1. The cost for a path is defined by the summation of the cost for each edge in this path. We also note that each cost of an edge is a positive value. We assume that each cost of an edge $\{i, j\}$ is given by an element of an array $c(i, j)$. For convenience, we define it as $c(i, j) = \infty$ if there is no edge between these two vertices. Since every cost is positive, we have $c(i, j) > 0$ for each $i \neq j$. We also define the cost as $c(i, i) = 0$ for all i. That is, we can move from one vertex to itself with no cost. Moreover, since we are considering an undirected graph, we have $c(i, j) = c(j, i)$ for any pair.

[1] **Edsger Wybe Dijkstra; 1930–2002**:
Dijkstra’s algorithm was conceived by Edsger W. Dijkstra, who was a computer scientist well known for this algorithm.

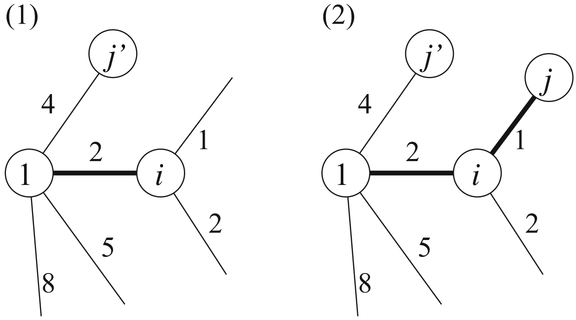


Fig. 4.5 Behavior of Dijkstra's algorithm

We first consider some trivial facts in order to consider the difficulty of the problem. What are the trivial facts that we can observe? For moving from the start vertex 1 to itself, the cost is trivial, namely the cost of moving to vertex 1 should be the minimum cost 0. Since all costs on edges are positive, we can make no more improvements. That is, $D[1]$ is definitely determined as 0. What next? We consider the neighbors of vertex 1 to which we can move in one step. We focus on the costs on these edges. Let i be the neighbor of vertex 1 that has the minimum cost among all the edges incident to vertex 1. Then, this cost $c(1, i)$ is the minimum cost of moving to vertex i from vertex 1. The reason is simple. If we move to another vertex to save the cost, then the cost is greater than or equal to the cost $c(1, i)$. (We remember the cost of each edge is positive, which means that the cost of this detour is not improved by passing more edges.) Therefore, among the neighbors of the vertex 1, for the vertex i that has the minimum cost $c(1, *)$, we have $D[i] = c(1, i)$.

Example 13 The above observation can be seen in Fig. 4.5(1). We can confirm $D[i] = c(1, i) = 2$ for vertex i, since it has the minimum cost among all the edges incident to vertex 1. As yet, we have no idea about the other vertices since some cheaper detours may exist.

Now we have to consider the issue more carefully. The next vertices that can be considered are the neighbors of the vertices 1 and i. Then, we have three sets of vertices, that is, the vertices only adjacent to vertex 1, the vertices only adjacent to vertex i, and the vertices adjacent to both. Of course, we have to consider all the vertices in these three sets. Then, what do we have to compare this with? For the vertices j' adjacent to only vertex 1, is it sufficient to consider the costs $c(1, j')$? No, it is not. Possibly, there may be a cheaper detour starting from vertex i to vertex j'. In this case, of course, it takes additional cost $c(1, i)$; however, it may be worthwhile. In Fig. 4.5(1), we have an edge of cost 1 from vertex i to the upper right direction. This cheap edge *may* lead us to vertex j' with a cost lower than $c(1, j')$. We have a problem.

We consider the potential detour calmly. If there exists a cheaper detour that passes through vertex i, the cost of the neighbor of vertex i on the detour should be low, and this cost is less than $c(1, j')$.

Investigating this point, we see that to "the nearest" neighbor vertex of both vertices 1 and i, we have no cheaper detour since we do know the shortest route to these vertices 1 and i. Therefore, we can compute the minimum cost to this nearest neighbor. That is, we can conclude that the next vertex j to the vertices 1 and i for which we can compute the shortest path is the one that has the minimum cost to it from 1 and i. Precisely, this vertex j has the minimum value of $\min\{c(1, j), c(1, i) + c(i, j)\}$ for all the neighbors of vertices 1 and i. (Here, min returns the minimum value of the parameters, that is, the minimum value of $c(1, j)$ and $c(1, i) + c(i, j)$, in this case. Precisely, this vertex j satisfies $\min\{c(1, j), c(1, i) + c(i, j)\} \le \min\{c(1, j'), c(1, i) + c(i, j')\}$ for any other vertex j' in the neighbors of vertices 1 and i.) Thus, we have to find the vertex j with this condition, and let $D[j] = \min\{c(1, j), c(1, i) + c(i, j)\}$.

Example 14 When the above discussion is applied to the case in Fig. 4.5(1), we can determine vertex j as in Fig. 4.5(2) and $D[j] = c(1, i) + c(i, j) = 2 + 1 = 3$. We cannot yet fix $D[]$ for the other vertices.

Thus far, we have a set $S = \{1, i, j\}$ of vertices for each of which we have computed the shortest route with the minimum cost from vertex 1. We can extend the above idea to the other vertices. First, we summarize the procedure we followed to find the next vertex to be computed using extendable notations.

- First, for the start vertex 1, $D[1] = c(1, 1) = 0$ is determined. Let $S = \{1\}$.
- Next, we find the minimum vertex i with respect to $c(1, i)$, and $D[i] = c(1, i)$ is determined. The set S is updated to $S = \{1, i\}$.
- Then, we find the minimum vertex j with respect to $\min\{D[1] + c(1, j), D[i] + c(i, j)\}$, and $D[j] = \min\{D[1] + c(1, j), D[i] + c(i, j)\}$ is determined. The set S is updated to $S = \{1, i, j\}$.

In the last step, $D[1] = 0$ is added to the summation to clarify the meaning of the equation. We also replace $c(1, i)$ with $D[i]$ in this context. That is, in the general step, we have to find the "nearest" neighbor to the set of the vertices for which the minimum costs have already been computed. Then, we compute the minimum cost to the new nearest neighbor and add the vertex to the set S as the new one for which the minimum cost is fixed.

Therefore, the procedure for the general case can be described as follows.

- Find the closest neighbor vertex j to the set S of the vertices for which the minimum cost of the path to them from the start vertex 1 has already been computed. More precisely, find the vertex j that has the minimum value of $\min_{i \in S}\{D[i] + c(i, j)\}$ among all vertices $j \notin S$.[2]
- For the vertex j with the minimum cost $\min_{i \in S}\{D[i] + c(i, j)\}$, store it to $D[j]$ and add j to the set S.

[2] **The notation of** min:
The notation $\min_{i \in S}\{D[i] + c(i, j)\}$ means the minimum value of $D[i] + c(i, j)$ for each vertex i in the set S. It may be difficult to see at the first time; however, it is concise and useful if you become familiar with it.

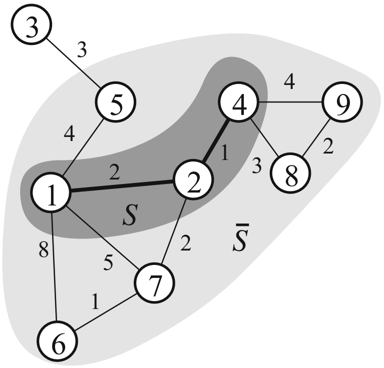


Fig. 4.6 Progress of Dijkstra's algorithm

Note that we take the minimum value twice. That is, we essentially compute a shortest route to each vertex not in S and find the nearest vertex among them.

We are now sure about the basic idea of the algorithm. However, when we try to describe this idea in detail in the form of an algorithm that can be realized as a computer program, we have to fill some gaps between the idea and the algorithm's description. In particular, when we aim to implement this idea as an efficient algorithm, there is still room for using devices.

We here introduce some useful tips for the implementation of Dijkstra's algorithm. First, in addition to set S, we will use another set $\bar{S}$ of vertices that are adjacent to some vertices in S and are themselves not in S. In the above discussion, we considered the nearest neighbor vertex j to some vertex in S that consists of vertices, the costs of which have been already computed. Vertex j should be in this $\bar{S}$. That is, when we seek vertex j with the condition, it is sufficient to search it only in $\bar{S}$, and we can ignore the other vertices in $V\setminus(S \cup \bar{S})$.

In the above discussion, we used an array $D[]$ to store the cheapest cost $D[i]$ from vertex 1 to vertex i, but we can make full use of this array not only for finding the final cost. More specifically, for the vertices i, the array $D[i]$ can store tentative data as follows.

- When i is a vertex in S, as already defined, $D[i]$ stores the cheapest cost from vertex 1 to vertex i.
- When i is a vertex not in $(S \cup \bar{S})$, we set $D[i] = \infty$.
- When i is a vertex in $\bar{S}$, the array $D[i]$ stores the cheapest cost from vertex 1 to vertex i through the vertices only in $S \cup \{i\}$.

Example 15 We confirm the data structure in the graph in Fig. 4.6. In the figure, the set $S = \{1, 2, 4\}$ is the set of vertices for which the minimum costs have been computed. Precisely, we have already $D[1] = 0$, $D[2] = 2$, and $D[4] = 3$, which are the solutions for these vertices. Then, the set $\bar{S}$ is the set of neighbors of some elements in S, and hence, $\bar{S} = \{5, 6, 7, 8, 9\}$. For vertex 5, the tentative cheapest cost is $D[5] = 4$, which is easy to compute. For vertex 6, we go through only the vertices in $S \cup \{6\}$ and we have $D[6] = 8$ so far. We can observe that this cost can be reduced by passing vertex 7. That is, this $D[6]$ will be improved in some future step. For vertex 7, the minimum cost is given by passing vertex 2 and hence we have $D[7] = 4$. Similarly, we have $D[8] = 6$ and $D[9] = 7$ for vertices 8 and 9, respectively. For vertex 3, we have $D[3] = \infty$ for this S (and $\bar{S}$).

Using these data sets S, $\bar{S}$ and D, the required operations of the algorithm can be summarized as follows.

Initialization: For a given graph, it lets $S = \{1\}$, $\bar{S} = \{i \mid$ vertex i adjacent to the vertex $1\}$ and sets the array D as

$$\begin{aligned} D[1] &= 0 \\ D[i] &= c(1, i) && \text{for each } i \in \bar{S} \\ D[j] &= \infty && \text{for each } j \notin S \cup \bar{S}. \end{aligned}$$

General step: First, find the vertex j in $\bar{S}$ that takes the minimum value $\min_{i \in S}\{D[i] + c(i, j)\}$ among $\min_{i \in S}\{D[i] + c(i, j')\}$ for all the vertices j' in $\bar{S}$. Set $D[j]$ as this minimum cost $\min_{i \in S}\{D[i] + c(i, j)\}$ and add j into S. In this case, $D[j]$ is fixed and will no longer be updated.

When the algorithm adds j into S, we have two things to do:

- Add the neighbors of the vertex j not in $\bar{S}$ into $\bar{S}$.
- Update $D[i']$ for each vertex i' satisfying $D[i'] > D[j] + c(j, i')$.

We give a supplementary explanation for the second process. Intuitively, by reason of adding j into S, we obtain a cheaper route to vertices i'. We have two types of such vertices; one has already been in $\bar{S}$, and the other is added to $\bar{S}$ only in the first process. (Note that no vertex in S is updated since its route has been already fixed.) For each vertex i' just added into $\bar{S}$ in the first process, since $D[i'] = \infty$, we can update $D[i'] = D[j] + c(j, i')$ immediately. If vertex i' has been in $\bar{S}$ already, we have to compare $D[i']$ with $D[j] + c(j, i')$, and update if $D[i'] > D[j] + c(j, i')$.

4.4.1 Implementations of Dijkstra's Algorithm

Now we understand Dijkstra's algorithm and the data structure that we need to perform the algorithm. In this section, we describe more details that allow us to implement it in a real program. However, unfortunately, using only the data structures included in this book, we cannot implement a Dijkstra algorithm that runs "efficiently." Therefore, we describe an implementation example using the data structures included in this book, and where we have a gap that can be improved. For advanced readers, it is a nice opportunity to investigate what data structure allows us to improve this implementation.

First, we consider the implementation of the sets S and $\bar{S}$. When we consider that there are some attributes belonging to each vertex, we can realize these sets by an array $s[]$. More specifically, we can define that vertex i is not in any set when $s[i] = 0$, $s[i] = 1$ means vertex i is in $\bar{S}$, and $s[i] = 2$ means i is in S. That is, each vertex i is initialized as $s[i] = 0$ and it changes to $s[i] = 1$ at some step, and finally when $s[i] = 2$, the value $D[i]$ is fixed.

The crucial problem is how we can implement the following two processes correctly.

- The process for finding vertex j in $\bar{S}$ that achieves the minimum value of $\min_{i \in S}\{D[i] + c(i, j)\}$.
- After finding j above, the process involved when adding j into S.

We tentatively name the processes the function **FindMinimum** and the subroutine **Update**, respectively. Then, the main part of the Dijkstra algorithm can be described as follows.

```
Algorithm 30: Dijkstra($A$, $c$)
  Input  : Adjacent set $A[]$ and cost function $c[]$
  Output: Minimum cost $D[i]$ to each vertex $i$ from vertex 1
for $i = 1, \ldots, n$ do
    $s[i] \leftarrow 0$;
end
$D[1] \leftarrow 0$ ;          /* Minimum cost to vertex 1 is fixed at 0. */
$s[1] \leftarrow 2$ ;                    /* Vertex 1 belongs to set $S$ */
for $j = 1, \ldots, d_1$ do
    $s[A[1, j]] \leftarrow 1$ ;       /* Put all neighbors of vertex 1 into $\bar{S}$ */
    $D[A[1, j]] \leftarrow c(1, A[1, j])$ ;      /* Initialize by tentative cost */
end
while $|S| < n$ do
    $j \leftarrow$FindMinimum($s, D, A$);
    Update($j, s, D, A$);
end
output $D[n]$;
```

This algorithm halts when all the vertices have been put into set S. Of course, it is sufficient that it halts when the goal vertex n is put into S. Here, we compute the minimum costs to all the vertices for the sake of simplicity. Hereafter, we consider how we can implement the function **FindMinimum** and the subroutine **Update**.

Function FindMinimum: In this function, we have to find the vertex j from $\bar{S}$ that gives the minimum value of $\min_{i \in S}\{D[i] + c(i, j)\}$ for all vertices in $\bar{S}$. It is easier to see by changing the viewpoint to that of the edges to compute this value. That is, if we can find an edge (i, j) such that i is in S, j is not in S, and $D[i] + c(i, j)$ takes the minimum value among all edges (i', j') with $i' \in S$ and $j' \notin S$, this j not in S is the desired vertex. Therefore, the following algorithm computes this function simply. The variable min keeps the current minimum value of $D[i] + c(i, j)$, and the variable j_m keeps the current vertex that gives the value of min. We have to initialize min by ∞. The variable j_m does not need to be initialized since it will be updated when min is updated.

```
Algorithm 31: Function FindMinimum($s$, $D$, $A$)
  Input  : $s[]$, $D[]$, $A[]$
  Output: Vertex $j$ that gives $\min_{i \in S}\{D[i] + c(i, j)\}$
min ← ∞;
for $i = 1, \ldots, n$ do
    if $s[i] = 2$ then                                   /* vertex $i$ is in $S$ */
        for $j = A[i, 1], \ldots, A[i, d_i]$ do
            if $s[j] = 1$ then                           /* vertex $j$ is in $\bar{S}$ */
                if $min > D[i] + c(i, j)$ then
                    $min \leftarrow D[i] + c(i, j)$;
                    $j_m \leftarrow j$;
                end
            end
        end
    end
end
return $j_m$;
```

The running time of the algorithm does not concern us. This algorithm may touch every vertex and every edge in constant time, and therefore, the running time of one call of **FindMinimum** is $O(n + m)$. In this implementation, there are two points involving redundancy that can be improved:

- In the algorithm, it checks every edge; however, it needs to check only each edge of which one endpoint is in S and the other is in $\bar{S}$.
- The function **FindMinimum** is called repeatedly, and at each call, it computes $D[i] + c(i, j)$ for all edges, finds the edge $\{i, j\}$ having the minimum value, and discards all the values of $D[i] + c(i, j)$. When the algorithm finds the edge $\{i, j\}$, it updates the information around the vertex j and its neighbors by calling **Update**. This update propagates only locally around vertex j. That is, when the function **FindMinimum** is called the next time, for almost all edges, $D[i] + c(i, j)$ is not changed from the last time.

Therefore, one reasonable idea is to maintain only the edges joining the sets S and $\bar{S}$ in a linked list structure. Then, in each step, if we maintain the neighbors of the modified edges with the least cost, we can reduce the running time for the function **FindMinimum**. More precise details of this implementation and the improvement of time complexity are beyond the scope of this book, and so we stop here.

Subroutine Update: In this subroutine, we have to move vertex j from set $\bar{S}$ to set S. This process itself is easy; we just update $s[j] = 1$ to $s[j] = 2$. However, we have to handle the associated vertices k adjacent to vertex j. For them, we have to consider three cases:

- The vertex k is already in S: In this case, we already have the least cost in $D[k]$, and we have nothing to do.
- The vertex k is in $\bar{S}$: In this case, since the edge $\{j, k\}$ may give a short cut to the vertex k, we have to update $D[k]$ to $D[j] + c(j, k)$ if $D[j] + c(j, k)$ is less than the current $D[k]$.

- The vertex k is not in both S and $\bar{S}$: In this case, the edge $\{j, k\}$ gives us a tentative shortest route to the vertex k, and hence, we can set the new cost $D[j] + c(j, k)$ to $D[k]$.

At a glance, these three cases are complicated. However, when we consider the two points

- All $D[k]$s are initialized by ∞ at first, and
- $D[k]$ gives the least cost if vertex k is already in S,

the following one process correctly deals with the above three cases simultaneously.

- If $D[k] > D[j] + c(j, k)$, let $D[k] = D[j] + c(j, k)$.

Therefore, the implementation of the subroutine **Update** can be given in the following simple form.

```
Algorithm 32: Subroutine Update($j$, $s$, $D$, $A$)
  Input  : $j$, $s[]$, $D[]$, $A[]$
  Output: Move vertex $j$ into set $S$
s[j] ← 2;
for $i = 1, \ldots, d_j$ do
    if $s[A[j, i]] = 0$ then
        $s[A[j, i]] = 1$ ;                          /* add $k = A[j, i]$ to $S$ */
    end
    if $D[A[j, i]] > D[j] + c(j, A[j, i])$ then
        $D[A[j, i]] \leftarrow D[j] + c(j, A[j, i])$;
    end
end
```

In this implementation, the subroutine **Update** runs in $O(d(j))$ time, where $d(j)$ is the degree, or the number, of neighbors, of the vertex j. It is difficult to essentially improve this part. However, we do not need to process the neighbors already in S; it can run a bit faster if we can skip them.

4.4.2 Analysis of Dijkstra's Algorithm

As described previously, the implementation in this book is not so efficient that there is room for improvement. However, it is a nice exercise for readers to evaluate the time complexity of the simple implementation above.

In the main part of Dijkstra's algorithm, the initialization takes $O(n)$ time, and the **while** statement is also repeated in $O(n)$ time. In this **while**, the function **FindMinimum** performs in $O(n + m)$ time, and the subroutine **Update** runs in $O(d(j))$ time. Therefore, n repeats of the function **FindMinimum** dominate the running time, and hence, the total running time is estimated as $O(n(n + m))$ time. In a general graph, the number of edges can be bounded above by $n(n - 1)/2$, or $O(n^2)$. Thus, Dijkstra's algorithm in this implementation runs in $O(n^3)$ time. If we adopt a more sophisticated data structure with fine analysis, it is known that Dijkstra's algorithm can be improved to run in $O(n^2)$ time.

Current route finding algorithm is...

On an edge-weighted graph, the shortest path problem with minimum cost is simple but deep. Although the basic idea of Dijkstra's algorithm is not very difficult, we now feel keenly that we need some nontrivial tricks that can be implemented so that it runs efficiently. Some readers may feel that such a graph search problem is certainly interesting, but is a so-called toy problem, that is, it is artificial and has few applications. However, this is not the case. You are daily served by such algorithms for solving graph search problems. For example, take your car navigation system, or your cell phone. In such a device, there exists a program for solving graph search problems. When you go to an unfamiliar place by car or public transportation, by any route, you have to solve the graph search problem. This is, essentially, the shortest path problem with minimum cost. Some decades ago, in such a case, you would have tackled this problem by yourself using maps and timetables. Moreover, this would not necessarily have given you the best solution. However, nowadays, anybody can solve this shortest path problem easily by using an electric device. We owe the development of efficient algorithms for this problem a great debt of gratitude.

Since it is in demand from the viewpoint of applications, research on graph search algorithms is still active. There are international competitions for solving these problems in the shortest time. The up-to-date algorithms can solve the graph search problems even if a graph has tens of thousands of vertices. For example, for the map of the USA, they can find a shortest path between any two towns in some milliseconds, which is amazing. As compared to the simple(!) Dijkstra's algorithm, these algorithms use very many detailed techniques to make them faster. For example, on some specific map, they use knowledge such as "It is not likely that the shortest path passes a point that is farthest from the start point," "Main roads (such as highways) are likely to be used," "Beforehand, compute the shortest paths among some major landmarks," and so on.

Chapter 5
Backtracking

Abstract In this chapter, we learn the technique called **backtracking**. Backtracking is a systematic method of trial and error. In short, it is an approach that tests all possible choices, looking for the desired solution. Up to this point, it is similar to the DFS and BFS algorithms discussed in Chap. 4 and, like these algorithms, checks in the search tree systematically in order. The main feature of backtracking is that, if it finds at some point that there is no possibility of finding a solution, it halts the search, returns to the last branch, and turns to the next choice that has not yet been tested. This trick—return and turn to the next choice—is the reason why this method is called "backtracking." If we can halt the redundant searching, even if the search tree is huge, we may find a solution efficiently.

What you will learn:

- Backtracking
- Eight queens
- Knight's tour

5.1 The Eight Queen Puzzle

We first consider a famous puzzle called **the eight queen puzzle**. The queen is the most powerful piece in the game of chess; it can move any number of squares vertically, horizontally, or diagonally (Fig. 5.1). The eight queen puzzle consists of placing eight queens on an 8×8 chessboard so that no two queens threaten each other. We here consider the following more general problem.

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_5

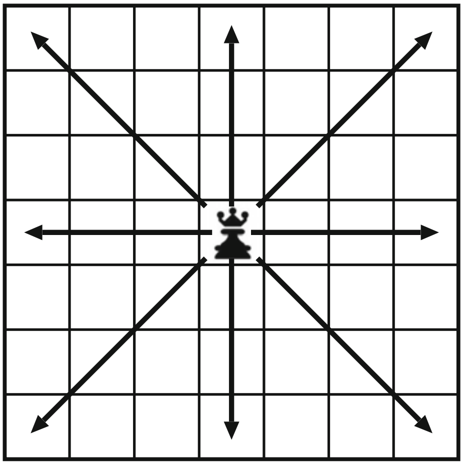


Fig. 5.1 Rules for the queen

The n queen puzzle:

Find an arrangement of n queens on an $n \times n$ chessboard such that no two queens threaten each other. For each n, how many arrangements are there?

At the moment, the goal is to solve the case $n = 8$.

5.1.1 Case Study 1

As a digression, we first consider whether we may place nine queens on an 8×8 chessboard. You may feel that, in some way, the ninth queen is one too many, and it cannot be placed. This intuition is correct; however, it is not very easy to prove it mathematically, or explain it clearly. This intuition can be proved using the following **pigeon hole principle**.

Pigeon hole principle: Assume that $n + 1$ or more pigeons are located in n pigeon holes. Then, there exists at least one hole in which there are two or more pigeons.

It has been claimed that the pigeon hole principle is "trivial." In this book, we accept the correctness of the pigeon hole principle without proof. I believe that the reader will have no objection to this claim. However, while the pigeon hole principle is simple and trivial, it is a very strong tool for some proofs. Representatively, using the pigeon hole principle, we can smartly prove the following theorem.

Theorem 13 *We cannot put $n + 1$ queens on an $n \times n$ chessboard so that no two queens threaten each other.*

Proof On this chessboard, we have only n columns. We assume that we have successfully placed $n + 1$ queens on the chessboard. Then, by the pigeon hole principle, there exists a column in which there are two or more queens. This contradicts the assumption since these two queens threaten each other. Therefore, in any arrangement of $n + 1$ queens, we have such a pair of two queens. Thus, we have the theorem. □

Letting $n = 8$ in Theorem 13, we can confirm that we cannot place the ninth queen on the 8×8 chessboard. Then, oppositely, can we always put n queens on the $n \times n$ board? We will put this problem at the back of our mind at the present time.

5.1.2 Case Study 2

According to convention, we first consider much smaller boards than $n = 8$. First, when $n = 1$, we can put one queen on a board of size 1×1. On the other hand, it is easy to confirm that there is no solution for $n = 2$: We cannot put two queens in the same row or in the same column, and we cannot arrange them along a diagonal line. Thus far, we observe that the n queen problem of finding an arrangement for n queens on a board of size $n \times n$ does not always have a solution.

Now, we turn to $n = 3$, that is, three queens on a board of size 3×3. To simplify the description, we name each cell from $B(1, 1)$ to $B(3, 3)$, as shown at the top of Fig. 5.2. According to the rule for the queen, we have to put a queen in each column. Therefore, we have to put a queen at either $B(1, 1)$, $B(1, 2)$, or $B(1, 3)$ in the first column. We consider these three cases separately.

On $B(1, 1)$: We would like to put the next queen at either $B(2, 1)$, $B(2, 2)$, or $B(2, 3)$. However, the first position at $B(1, 1)$ threatens the second queen if we put it on $B(2, 1)$ or $B(2, 2)$. Therefore, we put it only on $B(2, 3)$. Then, the last queen should be put on $B(3, 1)$, $B(3, 2)$, or $B(3, 3)$; however, $B(3, 1)$ is threatened by the first queen, $B(3, 2)$ by the second queen, and $B(3, 3)$ by both. Thus, in this case, we cannot put the third queen at any position.

On $B(1, 2)$: Although we need to put the second queen at $B(2, 1)$, $B(2, 2)$, or $B(2, 3)$, every cell is threatened by the first queen. Therefore, we have no solution in this case in the second column.

On $B(1, 3)$: This case is a mirror symmetric case of the first case of $B(1, 1)$. Therefore, we have no solution in this case either.

The above argument can be represented by the analysis tree depicted in Fig. 5.2. As a result, we can conclude that the n queen problem has no solution for $n = 3$. For readers who may wonder whether this problem has a solution for $n = 8$, we show a solution for $n = 4$ in Fig. 5.3. In fact, if we analyze this case in the same manner as $n = 3$, we find that it has two solutions; one is shown in Fig. 5.3, and the second is its mirror image. Intuitively, when n becomes larger, the number of choices increases, and hence, the n queen problem has a solution for larger n.

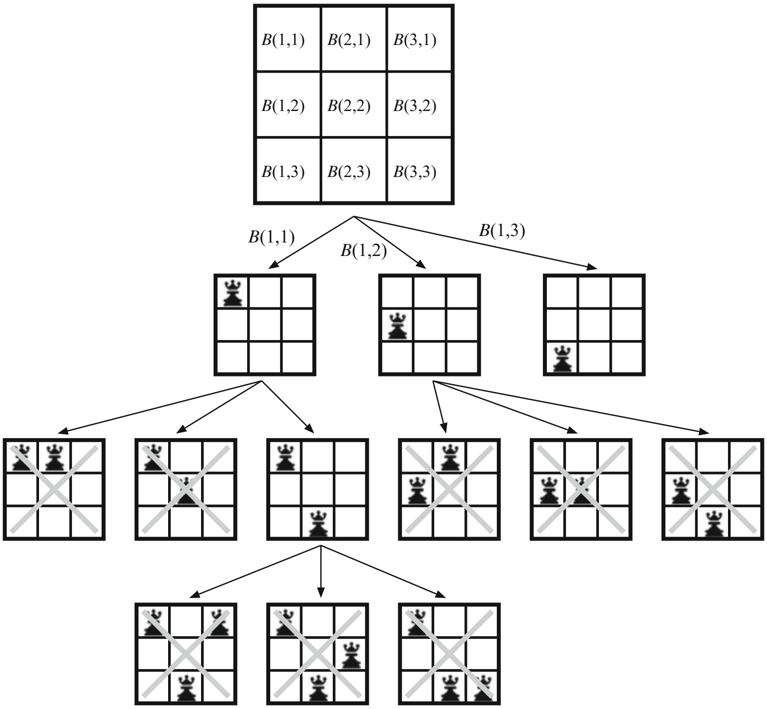
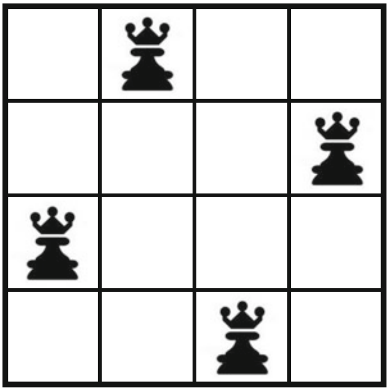


Fig. 5.2 Analysis tree for a board of size 3 × 3

Fig. 5.3 Solution of the n queen problem for $n = 4$



5.1.3 Backtracking Algorithm and Its Analysis

Now, we finish the exercise and turn to the general case. The basic idea is an extension of the method used in the case of a board of size 3 × 3. First, we use the array from $B[1, 1]$ to $B[8, 8]$ to represent the board. (Here, note that the notation (and notion)

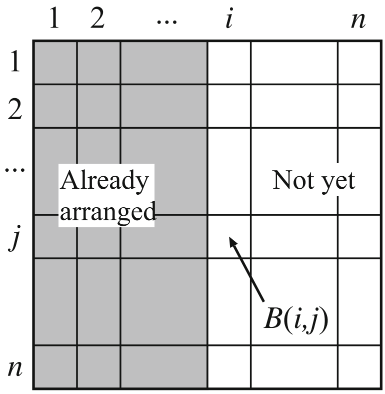


Fig. 5.4 Checking and pruning

differs slightly from that in the previous section: $B(i, j)$ indicates "the position on the board" and we use the value of the array $B[i, j]$ to maintain "the state" of the cell. To be precise, we distinguish $B[]$ from $B()$.) We let $B[i, j] = 0$ mean that there is no queen and $B[i, j] = 1$ that there is a queen. Then, we first consider the ith column for each $i = 1, 2, 3, \ldots, 8$. Next, we consider the jth row for each $j = 1, 2, \ldots, 8$ and check whether or not we can put the ith queen on $B(i, j)$. The important point is that, once we have found that we cannot put the ith queen at $B(i, j)$, we no longer need to consider the case $B[i, j] = 1$ (Fig. 5.4). Therefore, we can now halt the search. If we can put the ith queen at $B(i, j)$, we proceed for this case by letting $B[i, j] = 1$. If $i = 8$, this is a solution for the eight queens' problem, so we output it. When $i < 8$, we have to check further columns. Note that, after checking whether we can put the queen on $B(i, j)$, we have to reset $B(i, j)$ by letting $B[i, j] = 0$. It is more effective to arrange the process for each i into a subroutine. We name it QueenSub($B[], i$); we postpone giving the details of its implementation at a later time. Here, since we can put the first queen at any position in the first column, the algorithm for solving the eight queens' problem can be described as follows.

Algorithm 33: 8Queen(B) for the eight queens problem

```
Input  : 2-dimensional array $B[]$
Output: All solutions $B[]$ satisfying the condition for the eight queens problem
for $i \leftarrow 1, 2, \ldots, 8$ do
    for $j \leftarrow 1, 2, \ldots, 8$ do
        $B[i, j] \leftarrow 0$ ;                        /* Initialize the board */
    end
end
for $j \leftarrow 1, 2, \ldots, 8$ do
    $B[1, j] \leftarrow 1$ ;                   /* Put the first queen at $B(1, j)$ */
    QueenSub($B$, 2) ;                  /* Proceed to the second column */
    $B[1, j] \leftarrow 0$ ;             /* Remove the first queen from $B(1, j)$ */
end
```

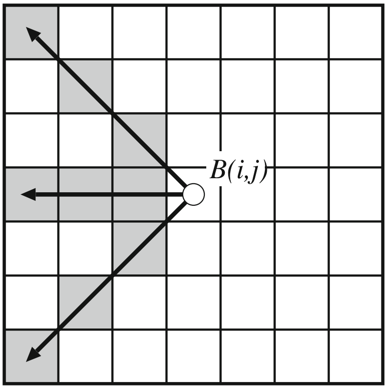


Fig. 5.5 Checking whether a queen threatens position $B(i, j)$. If a queen exists at one of the shaded positions, we cannot put a queen at $B(i, j)$

Here, we consider the problem that the subroutine QueenSub(B, i) should solve. When QueenSub(B, i) is invoked, we already have queens on $B(1, *)$ to $B(i - 1, *)$ ("$*$" means "some reasonable value"). Therefore, the algorithm has to check, for each $j = 1, 2, \dots, 8$, whether it can place a new queen at $B(i, j)$, and determines whether it can proceed or must backtrack. More precisely, the algorithm performs the following steps.

- If a queen cannot be placed at $B(i, j)$, halt the checking for this j.
- If a queen can be placed at $B(i, j)$, after placing the queen (by setting $B[i, j] \leftarrow 1$), perform the following.

  – when $i = 8$, output the current $B[]$ as a solution, and
  – when $i < 8$, call QueenSub($B[], i + 1$).

  After the process, reset $B[i, j]$ by setting $B[i, j] \leftarrow 0$.

Now, the last step is to check whether we can place a queen at $B(i, j)$ for $i > 1$. To do this, it is sufficient to check whether the other queens are arranged at the following positions (Fig. 5.5):

- $B(i - 1, j - 1) \cdot B(i - 1, j) \cdot B(i - 1, j + 1)$: the $i - 1$st column
- $B(i - 2, j - 2) \cdot B(i - 2, j) \cdot B(i - 2, j + 2)$: the $i - 2$nd column
  $\cdots$
- $B(1, j - (i - 1)) \cdot B(1, j) \cdot B(1, j + (i - 1))$: the first column.

To check this backwardly, we use a variable k. If the value of the row becomes 0 or 9, the algorithm can quit checking, because otherwise it will attempt to check areas outside the board.[1] If there are some queens in this row, the ith queen cannot be put on $B(i, j)$. We define that the array $B[i, j]$ takes 1 if there is a queen on $B(i, j)$. Therefore, we can place the queen on $B(i, j)$ if the summation of all the values of the corresponding positions is 0; otherwise, we cannot. (If we are attempting to create

[1] **Application of sentinel**: We can omit this check of "outside the board" using the technique called sentinel. If you have enough energy, consider how you can realize this.

the fastest possible algorithm, instead of computing the summation, we can quit the check as soon as we find an element 1. However, here we take an easier route to make it simple.) We now give the description of the algorithm:

```
Algorithm 34: Subroutine QueenSub($B, i$)
  Input  : 2-dimensional array $B[]$, the number $i$ of column
  Output: Each $B[]$ that satisfies the conditions of the eight queens problem
for $j \leftarrow 1, 2, \ldots, 8$ do
    $c \leftarrow 0$ ;                            /* The number of threatening queens */
    for $k = 1, \ldots, i - 1$ do
        if $j - (i - k) > 0$ then /* Queens on the left upper diagonal? */
            $c \leftarrow c + B[k, j - (i - k)]$;
        end
        $c \leftarrow c + B[k, j]$ ;               /* Queens on the left of $B(i, j)$? */
        if $j + (i - k) < 9$ then /* Queens on the left lower diagonal? */
            $c \leftarrow c + B[k, j + (i - k)]$;
        end
    end
    if $c = 0$ then                               /* No queen threatening here */
        $B[i, j] \leftarrow 1$ ;                        /* Put a queen at $B(i, j)$ */
        if $i = 8$ then                                   /* The last column? */
            output $B[]$ ;                               /* This is a solution */
        else
            QueenSub($B, i + 1$) ;             /* Proceed to the next column */
        end
        $B[i, j] \leftarrow 0$ ;                /* Remove the queen from $B(i, j)$ */
    end
end
```

We ran the above algorithm and obtained all the 92 solutions *immediately*. Since the solutions obtained by rotation and reflection are not regarded as distinct, we have essentially 12 different solutions in total, as shown in Fig. 5.6.

How can we reduce the number of redundant solutions?

It is not difficult to remove the redundant solutions obtained by rotation and reflection, although it is rather complicated. We first represent an 8×8 board with eight queens by a binary number of 64 bits. This can be achieved by introducing an ordering of the positions, e.g., $B(1, 1)$, $B(1, 2)$, ..., $B(1, 8)$, $B(2, 1)$, $B(2, 2)$, ..., $B(8, 1)$, ..., and $B(8, 8)$, and then estimating the values (0 or 1) of these variables as the binary representation. Now, for a given solution, we can obtain at most seven more variants by rotation and reflection. For each of these eight solutions, we estimate the corresponding numbers. Then, we output the solution if the corresponding number is the smallest of these numbers. It is easy to see that the redundant solutions are removed from the output. We omit the details of the algorithm in this book; however, it is a good exercise.

Exercise 35 ☕☕☕☕ Solve the n queen problem for $n = 4, 5, 6, 7$, and values greater than 8. As the value of n increases, how does the number of solutions change?

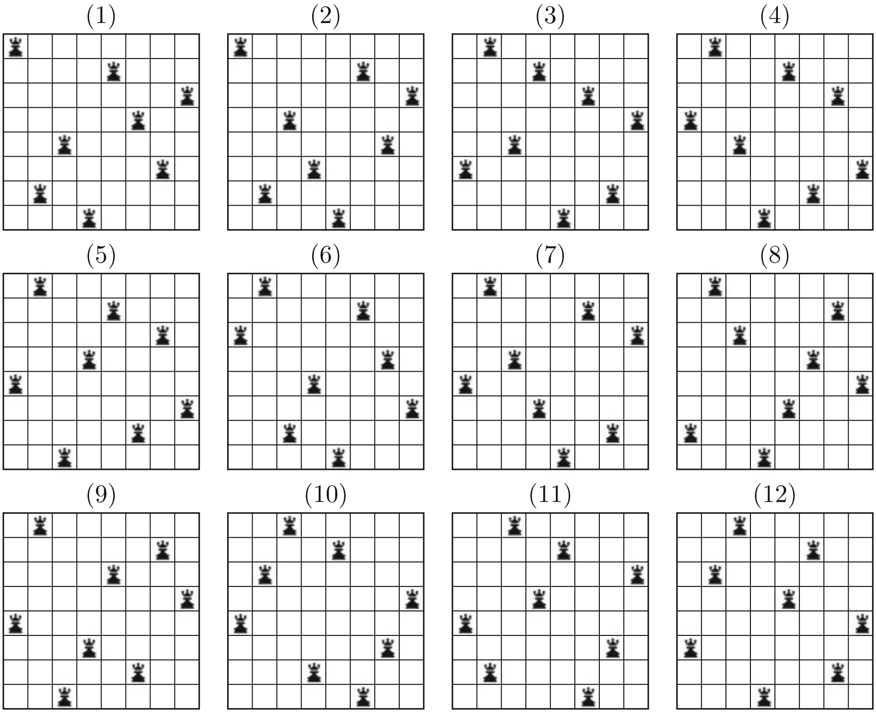


Fig. 5.6 Twelve different solutions of the eight queens' problem

Backtrack?

Although we are learning the method called "backtracking," even when you follow the algorithm, it may not seem to you to be "turning back." The turning back occurs at the part where recursive calls are made. After making recursive calling of the subroutine QueenSub, when the algorithm eventually can find no solution, it returns to the parent subroutines in order. When this occurs after making a long chain of recursive calls, the algorithm should turn back to a shallow node in the search tree from a deep node. This situation is called "backtracking." When you can grasp this operating principle and imagine the phenomenon vividly, you are no longer a beginner.

5.2 Knight's Tour

Now, we turn from the queen to the knight. In the game of chess, the knight's moves are somewhat different from those of the other pieces: It can move to a square that is two squares in the horizontal direction and one square in the vertical direction, or

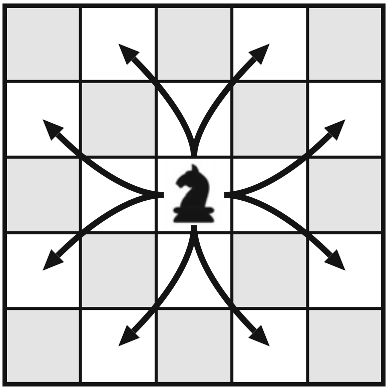


Fig. 5.7 Rule for the knight

two squares in the vertical direction and one square in the horizontal direction. There are eight possible squares to which it can move (Fig. 5.7). The knight tour problem consists of finding a route for a knight on a chessboard of size 8 × 8 such that the knight returns to the first position after visiting every square, except the first, exactly once. We here again consider this problem in a general style.

The knight's tour on a board of size $n \times n$:

On a chessboard of size $n \times n$, place a knight. Find a route such that the knight returns to the first position after visiting every square, except the first, exactly once. How many routes are there?

For the moment, we aim to solve this problem for $n = 8$. As in the n queen problem, we indicate each square by $B(1, 1)$ to $B(n, n)$. However, we may have to reconsider whether this notation is reasonable or not. In any case, the knight's tour requires a cyclic route such that the knight visits every square exactly once. Therefore, if there is a solution, it should visit square $B(1, 1)$, which means that we can assume that the solution starts from $B(1, 1)$ and ends at $B(1, 1)$ without loss of generality.

5.2.1 Case Study 1

We first consider some small cases to obtain a sense of the problem.

Case $n = 1$: In this case, the board consists of one square $B(1, 1)$, and we consider that we have a trivial solution in a route of length 0. That is, there is a unique solution for the problem when $n = 1$.

Case $n = 2$: In this case, the knight cannot move to any square from the start position. Therefore, we have no solution.

Case $n = 3$: If the knight is in the first position, $B(1, 1)$, we have two choices. However, they are essentially the same because of the symmetricity. Once we have

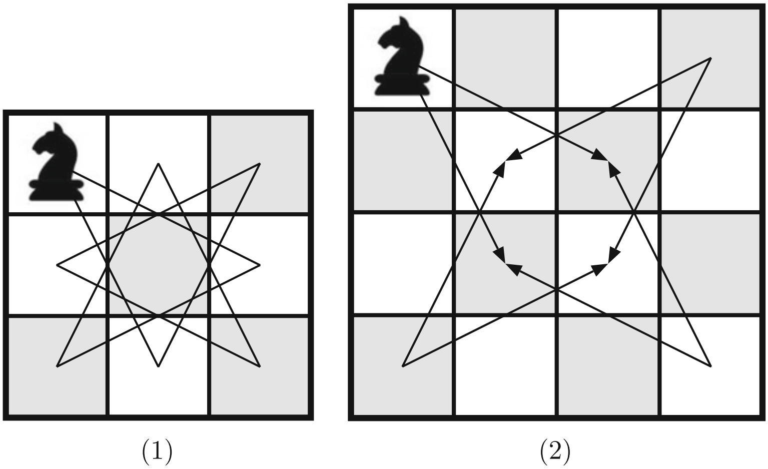


Fig. 5.8 Two cases of boards of size 3 × 3 and 4 × 4

decided the direction, the route is automatically determined since we have no other choice. Then, the knight returns to the start position after moving, as shown in Fig. 5.8(1). At a glance, this seems to be a solution; however, this is not so, since this route does not visit the central square $B(2, 2)$. That is, we have no solution for $n = 3$.

Case $n = 4$: As in the case where $n = 3$, from the knight's start position $B(1, 1)$, we have two choices that are essentially the same. On the other hand, since we are considering a cyclic route, these two moves should be a part of the solution. We draw them at all corners as a part of the solution on the board, as shown in Fig. 5.8(2). Then, we obtain two isolated cyclic routes on the board constructed from these unavoidable moves. Therefore, since a solution requires that the knight visits every square exactly once, we have no solution for this case either.

5.2.2 Case Study 2

From the case where $n = 5$, the problem becomes too complicated to solve by hand. Therefore, now we give our hands a rest and use our heads instead. When $n = 3$, the route cannot include the central square in any way. There is something we should mention about this property. A chessboard is usually colored in a checkered pattern, as shown in the figure. Let us consider this property, which is called **parity**. It is easy to observe that any series of movements of a knight consists of

$$\text{Black} \rightarrow \text{White} \rightarrow \text{Black} \rightarrow \text{White} \rightarrow \text{Black} \rightarrow \text{White} \rightarrow \text{Black} \rightarrow \text{White} \rightarrow \cdots,$$

that is, the knight should visit black and white squares alternately. Therefore, in any route, the number of black squares should be equal to the number of white ones (expect for the trivial case $n = 1$). That is, if the number of the black and white squares on the checkerboard is different, we have no solution of the knight tour problem on this board. Considering this property, when $n = 3$, we can immediately conclude that we have no solution on this board since we can see that there are five black squares and four white squares without performing a case analysis.

In general, when the length of the edge of a board is odd, namely the size is $(2k + 1) \times (2k + 1)$ for some positive integer k, the number of squares is odd since $(2k + 1) \times (2k + 1) = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$. We cannot split an odd number into two equal numbers. Thus, we obtain the following theorem.

Theorem 14 *If the length n of the edge of a board is odd, the knight tour problem on a board of size n × n has no solution.*

5.2.3 Case Study 3

In the n queen problem, we dealt with a board by using a two-dimensional array; however, in the knight tour problem, it is worth considering a more reasonable model to clarify the problem. Now, we represent the board by a graph. We model it as follows. Each square corresponds to a vertex of a graph, and each pair of two vertices is joined by an edge if and only if the knight can move from one corresponding square to the other. Then, for a board of size 3 × 3, we obtain the corresponding graph shown in Fig. 5.9. From the board given in Fig. 5.9(1), we obtain the graph in Fig. 5.9(2) naturally; however, we can modify it as in Fig. 5.9(3) by moving the white vertices. Now, it is quite easy to see that the route is uniquely determined and there is no solution.

Moreover, when we represent the boards by graphs, we can estimate roughly the size of the problem beforehand. For example, let us look at the two graphs shown in Fig. 5.10. The graph representing the board of size 6 × 6 consists of 4 vertices

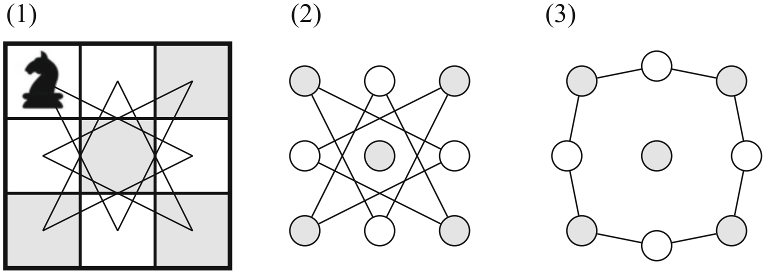
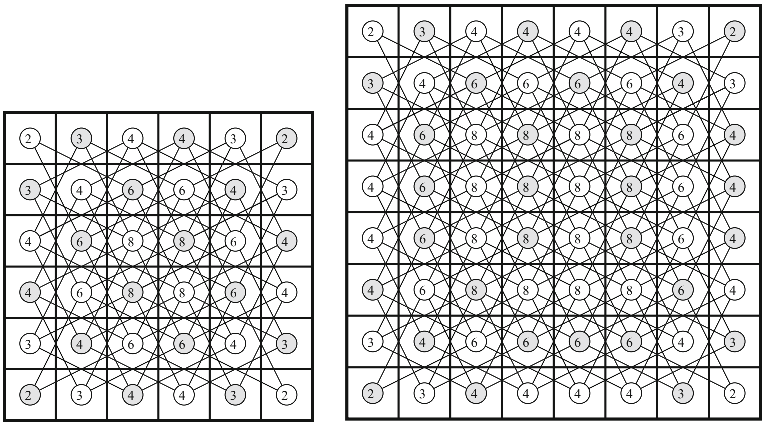


Fig. 5.9 Representation of a board of size 3 × 3 by a graph



(1) Board of size 6 × 6

(2) Board of size 8 × 8

Fig. 5.10 Representations of boards of size 6 × 6 and 8 × 8 by graphs

of degree 2, 8 vertices of degree 3, 12 vertices of degree 4, 8 vertices of degree 6, and 4 vertices of degree 8. From these data, we can obtain a rough estimate of the running time of the algorithm. In the search tree, consider a vertex. This vertex is visited from another vertex, and then the algorithm will visit a next vertex. Therefore, in the worst case, the number of candidates of the "next" vertices from this vertex is at most $d - 1$, where d is the number of degree of this vertex. In the search tree, the algorithm may find some vertices that have already been visited. Therefore, in particular in the latter part of the searching, the number of candidates is less than this value. From this observation, we can estimate roughly the number of vertices visited in the search tree of the knight tour problem on a board of size 6 × 6. More precisely, the following equation gives us an upper bound of the number of vertices that the search algorithm will visit:

$$(2-1)^4 \times (3-1)^8 \times (4-1)^{12} \times (6-1)^8 \times (8-1)^4 = 2^8 3^{12} 5^8 7^4 \sim 1.3 \times 10^{17}.$$

How fast can an up-to-date computer check all the vertices? Let us assume that your PC has a CPU with a 3 GHz clock. Then, roughly speaking, it can perform 3×10^9 basic operations per second. (In fact, some basic statements require few clocks. Therefore, the number of possible operations is smaller. However, this does not concern us since it is a rough estimation.) Then, roughly, it takes $(1.3 \times 10^{17})/(3 \times 10^9) = 4.3 \times 10^7$ seconds, which implies around 11,800 hours. In general, a backtrack algorithm drastically reduces redundant searching, and hence, we can hope that it will run much faster than that rough estimation in the worst-case scenario.

Next, we turn to the graph corresponding to the board of size 8 × 8. In this graph, there are 4 vertices of degree 2, 8 vertices of degree 3, 20 vertices of degree 4, 16 vertices of degree 6, and 16 vertices of degree 8. In the same way, we can compute the upper bound of the number of vertices that will be visited in the search as

$$(2-1)^4 \times (3-1)^8 \times (4-1)^{20} \times (6-1)^{16} \times (8-1)^{16} = 2^8 3^{20} 5^{16} 7^{16} \sim 4.5 \times 10^{36}.$$

Assuming we are using the CPU with a 3 GHz clock, we obtain $(4.5 \times 10^{36})/(3 \times 10^9) = 1.5 \times 10^{27}$ seconds, which implies around 5×10^{19} years. It seems that we have no hope of completing the computation even if we use some tricks to improve the running time.

Therefore, we first aim to find all the solutions of the knight tour problem on a board of size 6 × 6.

Search space and board size

In the eight queen problem, our algorithm finds all the solutions immediately, even without careful analysis. We are now considering a board of the same size; however, the algorithm's running time is very different. Why is it so different? The crucial point of the eight queen problem is that we can put only one queen in each column. That is, the depth of the search tree is only eight levels, and we have only eight choices at each level. Therefore, the size of the search tree is bounded above by, roughly, $8^8 = 16777216$, and this implies that the running time on the same CPU model is estimated to be $8^8/(3 \times 10^9) = 0.00559$ seconds. Thus, we can hope that the running time is within one second, and it is.
On the other hand, in the knight tour problem, the depth of the search tree is the size of the board. As mentioned in the analysis, the depth of the search tree corresponds to the exponent that is on the shoulder of some number. In an exponential function, a large exponent means that the resulting value is of astronomical size.
To gain general know-how, we have to consider the case where a search tree is deep. Naive implementation can take an enormously long computational time.

5.2.4 *Knight Tours on Board of Size* 6 × 6

We first represent the board by a graph structure. In this case, using a two-dimensional array $B[i, j]$ somewhat complicates the problem. Therefore, instead of the array, we label each cell of the board with a distinct number, as shown in Fig. 5.11. Hereafter, we call each cell labeled by the number i vertex i. (This is the same correspondence between one- and two-dimensional arrays as that introduced in Sect. 3.3.)

Now, we can represent the board given in Fig. 5.10(1) by a graph, as shown in Fig. 5.12. We here represent this graph by the adjacency set. The number of vertices in the graph is 36, each vertex is labeled by a unique number from 0 to 35, and the maximum degree is 8. To represent these data, we use an array $A[]$. The element $A[i, j]$ represents the jth neighbor of vertex i. Therefore, we have $0 \leq i \leq 35$ and

Fig. 5.11 Indices of the positions on the board of size 6×6

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

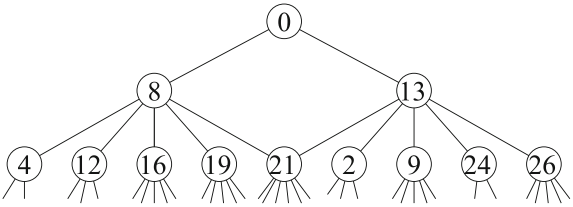


Fig. 5.12 Part of the graph corresponding to the board of size 6×6. Only the area around the vertex 0 is depicted

$0 \le j \le 8$. The end of the neighbors is represented by -1 (otherwise, we cannot distinguish it from vertex 0). For example, the representation of the neighbors of vertex 0 is given by

$$A[0,0] = 8, A[0,1] = 13, A[0,2] = -1, A[0,3] = -1, A[0,4] = -1,$$
$$A[0,5] = -1, A[0,6] = -1, A[0,7] = -1, A[0,8] = -1,$$

and the neighbor set of vertex 14 is given by

$$A[14,0] = 1, A[14,1] = 3, A[14,2] = 6, A[14,3] = 10, A[14,4] = 18,$$
$$A[14,5] = 22, A[14,6] = 25, A[14,7] = 27, A[14,8] = -1.$$

On the other hand, we also need an array $V[]$ to record the visited vertices. That is, $V[i] = 0$ means the route has not yet visited vertex i, and $V[i] = 1$ means that vertex i has already been visited, and hence, the route can no longer jump to this vertex. We also prepare a variable c to record the current vertex and another variable p to count the number of steps of the route.

When we start from vertex 0, the next possible vertex is either vertex 8 or another vertex 13. To simplify the algorithm, we fix that the first step is from vertex 0 to vertex 8. Then, we obtain a solution if and only if the route visits vertex 13 at the 35th step.

Based on this idea with data structure, the main part of the algorithm is described as follows.

```
Algorithm 35: 6Knight($A$) for the knight tour problem of size $6 \times 6$
  Input  : Adjacency set $A[]$
  Output: Output all solutions of the knight tour problem
for $i \leftarrow 0, 1, 2, \ldots, 36$ do
    $V[i] \leftarrow 0$ ;                    /* all vertices are unvisited */
end
$c \leftarrow 0$ ;                                 /* start from vertex 0 */
$V[0] \leftarrow 1$ ;                              /* vertex 0 is visited */
$p \leftarrow 1$ ;                               /* this is the first step */
check(8, $p$) ;                                /* proceed from vertex 8 */
```

The subroutine check, which searches and visits each unvisited vertex, is not very difficult.

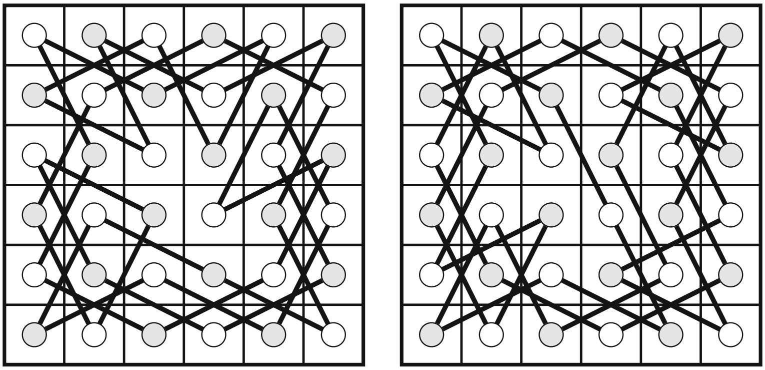
```
Algorithm 36: Subroutine check($i$, $p$) for the next possible position for the knight
  Input  : Next vertex $i$ and the number $p$ of steps
  Output: Output all solutions of the knight tour problem
$V[i] \leftarrow 1$ ;                             /* vertex $i$ is visited now */
if $p = 35$ and $i = 13$ then                       /* reach a solution! */
    output solution;
else
    $j \leftarrow 0$;
    while $A[i, j] > -1$ do          /* check each unvisited neighbor */
        if $V[A[i, j]] = 0$ then check($A[i, j]$, $p + 1$);
        $j \leftarrow j + 1$;
    end
end
$V[i] \leftarrow 0$ ;                        /* change vertex $i$ to unvisited */
```

Although it is a rather tedious task to initialize the array $A[]$ that represents the structure of the graph, we can obtain the algorithm for solving the knight tour problem for a board of size 6×6. Then, will the algorithm halt in reasonable time? In other words, does the cut off of the redundant searching in the backtracking, which is called **pruning**, work correctly? We cannot check this without conducting an experiment.

Results on a board of size 6×6. I implemented the algorithm for solving the knight tour problem on a board of size 6×6 on a standard desktop PC using the programming language C. It ran in a reasonable time, and I obtained all the solutions. Concerning the algorithm, we note the following two points.

- It does not remove the redundant solutions obtained by rotation and reflection.
- It finds only all the routes that start from vertex 0, move to vertex 8, and return to vertex 0 from vertex 13.

Therefore, the precise number of solutions needs to be discussed. However, this point no longer concerns us. I list some data here for your information:



(1) First solution

(2) Last, 9862nd, solution.

Fig. 5.13 First and last solutions on the board of size 6 × 6

Running time: The PC used in this experiment was built in 2004, the clock of the CPU is 3 GHz, and the size of the main memory is 1 GB. It is not now considered a fast machine, but it took 10 minutes to complete the computation.
The number of solutions: The number of solutions is 9862. The first and last solutions are shown in Fig. 5.13.
The number of visited vertices: We can obtain the number of visited vertices in the algorithm by counting how many times the statement $V[i] \leftarrow 1$ is performed. In the algorithm, the number of visited vertices is 1,125,977,888.

We here focus on the last number, which is the number of visited vertices. The first approximate estimation gives us the upper bound 1.3×10^{17}, where the real value is 1,125,977,888. Dividing the upper bound by the real value, we obtain $1.3 \times 10^{17}/1125977888 \sim 1.1 \times 10^{8}$. That is, the algorithm runs a billion times faster than the approximate estimation. The source of this drastic improvement is the pruning, i.e., removing the redundant searching. On the other hand, such an improvement is not sufficient to allow us to solve the knight tour problem on a board of size 8 × 8.

Exercise 36 ☕☕☕☕ In the subroutine check above, only "output solution" is written, without the details. In this algorithm, the route itself is not recorded when it moves to the next vertex. Therefore, it cannot output the solution as can the current algorithm. How can we modify the algorithm to output the solution?

Knight's tour on board of size 8 × 8. The approximate estimate of the size of the search tree of a board of size 6 × 6 is 1.3×10^{17}. On the other hand, it is 4.5×10^{36} for a board of size 8 × 8. Taking the ratio, we obtain $(4.5 \times 10^{36})/(1.3 \times 10^{17}) = 3.5 \times 10^{19}$. From the fact that the board of size 6 × 6 takes 10 minutes, we can conclude that it is intractable to solve the problem that would take 10^{19} times more. However, the board of size 6 × 6 has 9862 solutions. Therefore, for the board of size 8 × 8, even if it is impossible to enumerate all the solutions, it may, however, be possible to find some solutions. If you agree with this positive idea, tackle the following exercise.

Exercise 37 ☕☕☕☕☕ Design an algorithm for finding solutions for a board of size 8 × 8 by backtracking. However, modify the algorithm to halt when it finds one or some solutions. Is it possible to find a solution by using this algorithm?

How many hours does it take to find the first solution? Guess the number of hours beforehand and then try it. Was your guess correct?

Chapter 6
Randomized Algorithms

Abstract In this chapter, we learn some basic **randomized algorithms** and their analyses. Randomized algorithms are a type of probabilistic algorithm. In some contexts, the term probabilistic algorithms may be used in two different meanings: The algorithm itself uses random bits or probabilistic data is input to some deterministic algorithms. In general, when we refer to randomized algorithms, we mean that we are considering the algorithms on a computational model that has some access to random bits. There is a random machine model that assumes that it can obtain one random bit of 0 or 1 uniformly at random in one step and another random machine model that can obtain one random real integer number in some range uniformly at random in one step. Using this feature neatly, we can design simple and powerful algorithms. In this chapter, we will touch on some of their charm.

What you will learn:

- Random number and randomness
- Pseudo-randomness
- Shuffle problem
- Coupon collector's problem

6.1 Random Numbers

First, what is a "random number"? It is a question that has a philosophical flavor and is quite difficult to answer accurately. For example, is 100 a random number? How about 11.235813, or the ratio π of the circumference of a circle to its diameter? To begin with, is it strange to discuss whether one specific value is random or not? Then, which is more random, 1, 0, 1, 0, 1 or 1, 1, 1, 1, 1? We can say that we have no certain answers to these questions. In this book, we do not get involved with

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_6

this problem. We assume that we have a perfect black box. Putting some range into this box, we can obtain one random value in the range uniformly at random.[1] This obtained value is our **random number**.

In up-to-date computers, we can use some mechanism as the black box to generate random numbers called **pseudo-random numbers**. In a standard computer language, for example, we can use a function random() and obtain a random value between 0 and 1 at random by calling this function. In this chapter, we follow this manner; when we call random(), we obtain a random value r with $0 \leq r < 1$. This function is realized as follows. The system first prepares some number called the **seed** and applies some function repeatedly, and then generates a sequence of numbers that are apparently random. Therefore, this system can reproduce the same sequence from the same seed. This property is important, since we sometimes need randomness and a reproductive property simultaneously when we run experiments on a computer. A huge amount of research has been conducted on pseudo-random generators, and some practical pseudo-random mechanisms exist that also have nice properties from the mathematical viewpoint. In this book, we assume that our system has already provided us with such a good (pseudo-)random generator.

Mersenne twister method

Mersenne twister is one of the good pseudo-random generators, which is practical and well known. Not only it has been shown that this algorithm has advantages as a pseudo-random generator from the mathematical point of view, but also several source codes in popular computer languages are public. These codes are short and fast, and the generated numbers are good as a series of random numbers. Information can be found at http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html (accessed in November, 2018) maintained by Makoto Matsumoto.

Once you have obtained a random real number r with $0 \leq r < 1$, you can generate a random integer from 0 to $k - 1$ by computing $\lfloor kr \rfloor$ for any positive integer k. If r is uniformly at random, so also is $\lfloor kr \rfloor$. We can also perform this procedure in the opposite direction. That is, once we obtain a mechanism for random generation in some range, we can map it to an arbitrary desired range.

Exercise 38 ☕☕ Let r be a random value that satisfies $0 \leq r < 1$ and is uniformly at random. Then, for a given positive integer k, show that $\lfloor kr \rfloor$ gives a random variable that satisfies $0 \leq \lfloor kr \rfloor < k$ and is uniformly at random.

In this chapter, we first consider algorithms for shuffling as representative randomized algorithms and analyze the coupon collector's problem as an analysis exercise of randomized algorithms.

[1] **Uniformly at random**:
The term **uniformly at random** is, of course, a technical term in mathematics, and we have an accurate definition. In this book, we do not discuss the definition of randomness. Intuitively, we can assume that each possible number appears with the same probability.

6.2 Shuffling Problem

The shuffling problem is aimed to arrange the elements of a given array randomly. Imagine that you want to give a deck of cards a good shuffle. In other words, this problem is the reverse of the sorting problem. The algorithm can be used for generating test data for another algorithm. More precisely, the shuffling problem can be defined as follows.

The shuffling problem:

Input: An array $a[1], a[2], \ldots, a[n]$
Output: The array $a[]$ such that its elements are reorganized uniformly at random.

How can we solve this problem efficiently using a randomized algorithm? First, we consider a brief strategy to solve it. We will adopt the following idea.

(1) For each $i = n, n-1, \ldots, 2, 1$,

  (a) generate a random number r with $1 \leq r \leq i$, and
  (b) output the rth not output element in $a[]$.

The key point is how we can find the rth element from the elements in $a[]$ that has not already been output. We consider each step for efficient implementation.

Exercise 39 ☕☕ Prove that the above strategy certainly works. More precisely, it is sufficient that any sequence of the numbers from 1 to n appears uniformly at random.

6.2.1 *Naive Algorithm*

Off the top of our head, the first naive algorithm can be as follows. Mark each element of $a[]$ when it is output. At each iteration, the algorithm counts up to the rth element among elements with no marks and outputs the rth element. For this purpose, we prepare another array $b[]$ in which to store the elements in $a[]$. We let that $a[i]$ is not output if $b[i] = 0$ and $a[i]$ has already been output when $b[i] = 1$. We use a small idea to find the rth element. After generating the random number r with $1 \leq r \leq i$, the algorithm counts up j and checks $b[j]$, and then, when $b[j] = 0$, it decreases r itself. When r becomes 0, $a[j]$ is the rth element. Using this idea, we obtain the following algorithm.

Algorithm 37: NaiveShuffle(a)

```
Input  : An array $a[]$
Output: The array $a[]$, which is randomly shuffled
for $i \leftarrow 1, 2, \dots, n$ do
    $b[i] \leftarrow 0$;
end
for $i \leftarrow n, n-1, \dots, 2, 1$ do
    $r \leftarrow \lfloor random() \times i \rfloor + 1$ ;            /* This $r$ satisfies $1 \le r \le i$ */
    $j \leftarrow 0$;
    while $r > 0$ do
        $j \leftarrow j + 1$;
        if $b[j] = 0$ then
            $r \leftarrow r - 1$;
        end
    end
    output $a[j]$;
    $b[j] \leftarrow 1$;
end
```

Example 16 Let $a[]$ be an array of size 10 and initialized by the data 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 in this order. For each $i = 10, 9, 8, 7, 6, 5, 4, 3, 2, 1$, we assume that the random numbers are obtained as $r = 8, 4, 6, 5, 5, 1, 2, 2, 1, 1$, respectively. Then, the algorithm shuffles this array as shown in Fig. 6.1. The elements in gray-shaded boxes have already been output. At each step, it skips some elements that have been output and outputs the rth element. For example, $a[8]$ and $a[4]$ are the 8th and 4th element as is; however, $a[7]$ is the 6th element since the algorithm skips $a[4]$ since it has already been output. As a result, we obtain the sequence 8, 4, 7, 6, 9, 1, 3, 5, 2, 10 as output.

This algorithm seems to be relatively complicated as described here although the basic idea is simple. Now, we analyze the computational complexity of this algorithm. Its space complexity is $O(n)$ since it uses an extra array $b[]$. (Precisely, each element of $b[]$ requires only 1 bit, so it can be reduced as compared to $a[]$ in some real system.) The time complexity is $O(n^2)$; it consists of n loops for the variable i, and for each i, it takes $O(n)$ steps for processing r and j.

Exercise 40 ☕☕ In NaiveShuffle(a), the algorithm outputs only the values of $a[]$ and does not write them back into $a[]$. That is, $a[]$ itself is not shuffled as required. Modify the algorithm so that the resulting $a[]$ is shuffled. (Hint: You can store the output and write it back at the last step. You may have some nice tools that you have already learnt.)

Now, we consider whether there is a redundant part in this naive algorithm. The algorithm should eventually output n elements, and hence, we cannot avoid the loop for i. The point that should concern us is "counting up" from the top "to find the rth element that has not yet been output." In the current implementation, the elements of "not yet output" and "already output" are mixed in the array, and we cannot avoid checking from the first element, and hence, we have to use $O(n)$ time. You possess

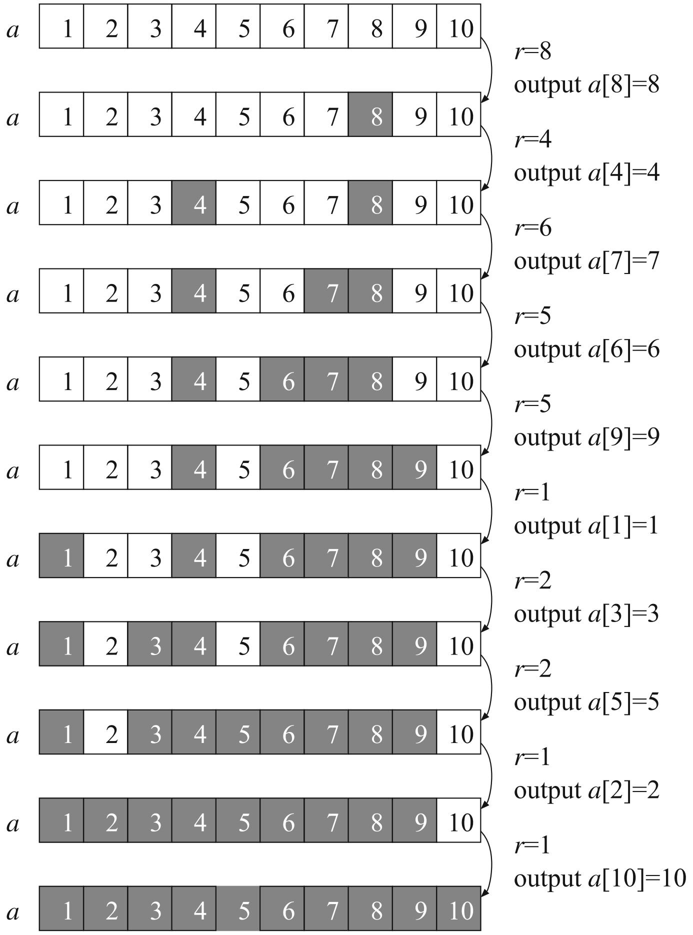


Fig. 6.1 Behavior of the naive shuffling algorithm

some sense of algorithms if you feel that this part may be reorganized in some smart manner.

In fact, we can use a smart data structure for “counting up” here and perform a type of binary search instead of this linear search. Thus, we can reduce the running time of this part from $O(n)$ to $O(\log n)$, and hence, the time complexity can be reduced to a total of $O(n \log n)$. However, in this case, we have to use a complicated data structure. Here, we would like to use a smart idea instead of the complicated data structure.

Smart data structure

I add some additional information for advanced readers. In this book, we do not use “pointers,” which is the next step of the data structure “array.” Using the data structure called **binary tree**, which requires the use of pointers, we can find the rth element among n data in $O(\log n)$ steps. We do not discuss binary trees since it is outside the topic of this book, but it is a nice idea for advanced readers to investigate.

Before going to the next section, let us stop and consider. Can you find an elegant algorithm by yourself? There is a hint in the above discussion. More precisely, let us consider carefully the following line: “In the current implementation, the elements of “not yet output” and “already output” are mixed in the array and we cannot avoid checking from the first element.” We can arrive at an elegant algorithm by considering this part in depth.

☕ Let us consider for a while ☕

6.2.2 Elegant Algorithm

First, why do we have to find the rth element from the first element by checking the elements one by one using the variables r and j? The reason is that in the array $a[]$, two different types of data are mixed: One group contains data already output, and the other contains data not output. We may find the element indicated by a random variable r with $1 \leq r \leq i$ in a constant time if we can handle this part appropriately. To find it in a constant time, we make the element $a[r]$ be exactly the rth element.

Now, suppose that we are looking at the array from $a[1]$ to $a[i]$. Then, the random variable r is generated to satisfy $1 \leq r \leq i$. After output $a[r]$, we would like to change the situation that the array from $a[1]$ to $a[i-1]$ consists of data not yet output. Can we achieve this? Yes, by swapping $a[r]$ and $a[i]$. Then, we can obtain an assertion that the data that have been output are packed into the latter part of the array and the former part always contains only fresh data that have not yet been output.

Using this nice idea, we can implement the algorithm for the shuffling problem as follows. We no longer need the array $b[]$ and the variable j. We are already familiar with the subroutine swap() that swaps two elements $a[i]$ and $a[r]$. Not surprisingly, the array $a[]$ after performing this algorithm is exactly the array that is shuffled uniformly at random, which we desired. (Precisely speaking, the array is in the reverse order of the sequence generated by the random values r; however, we desire a shuffled array, so this is not an issue!)

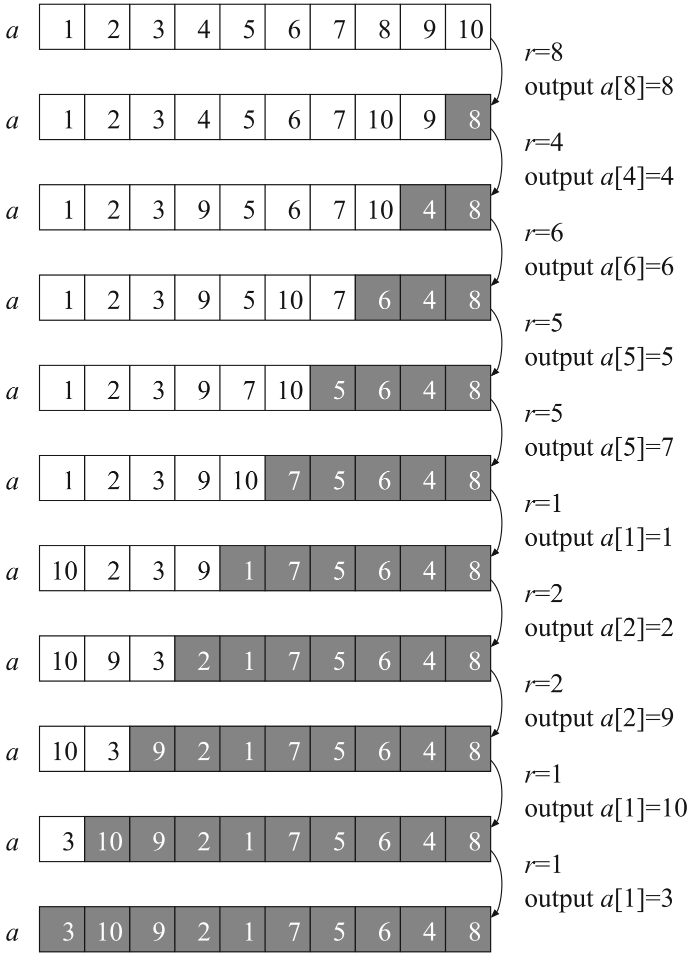


Fig. 6.2 Behavior of the elegant shuffling algorithm

```
Algorithm 38: ElegantShuffle($a$)
  Input  : An array $a[]$
  Output: Array $a[]$ shuffled in random ordering
for $i \leftarrow n, n-1, \ldots, 3, 2$ do
    $r \leftarrow \lfloor random() \times i \rfloor + 1$ ;          /* The case $i = 1$ is redundant */
    swap($a[i], a[r]$);
end
```

Example 17 Let $a[]$ be an array of size 10 and initialized by the data $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ in this order. For each $i = 10, 9, 8, 7, 6, 5, 4, 3, 2, 1$, we assume that the random numbers are obtained as $r = 8, 4, 6, 5, 5, 1, 2, 2, 1, 1$, respectively. Then, the algorithm shuffles this array as shown in Fig. 6.2. The elements in the gray-shaded boxes have been already output. These elements are always packed into the tail of the array, for any r with $1 \leq r \leq i$, $a[r]$ always comprises fresh data that have not yet been output.

It is almost trivial that this simple algorithm solves the shuffle problem correctly. The computational complexity of this algorithm is also easy to analyze. Its space complexity is only the two variables r and i, except the input $a[]$. (Using the technique in the solution of Exercise 3, we do not need to use additional variables in the swap.) Its time complexity is also trivially $\Theta(n)$, which means that this algorithm runs quite fast.

Comparing these two algorithms, one may notice that, for the same sequence of random numbers, the output sequences are different. In a naive algorithm, since it realizes the basic idea as is, we can apply the proof of the answer to Exercise 39. Then, how about the elegant shuffling algorithm? To confirm the correctness of the elegant algorithm, consider the following exercise.

Exercise 41 ☕☕☕ Prove that the elegant shuffling algorithm works correctly. That is, prove again that any sequence of the numbers from 1 to n appears uniformly at random.

6.3 Coupon Collector's Problem

There are n different types of coupon. The quantity of coupons is sufficiently huge that, when you buy one, you get any type of coupon with the same probability equal to $1/n$. One day, since you are a coupon collector, you decide to buy coupons until you have collected all the types of coupon. Then, how many coupons will you need to buy to complete the set of n different coupons? Let $C(n)$ be the expected number of coupons you buy. The **coupon collector's problem** is to compute this function $C(n)$.

If you are a collector of something, you are familiar with this problem. In this section, we give a theoretical answer to it. The coupon collector's problem itself is not an algorithmic problem; however, it sometimes appears when you analyze randomized algorithms and it may be useful in your daily life.

For the coupon collector's problem, the following theorem is known.

Theorem 15 *Assume that there is a huge amount n of types of coupon, and you draw each coupon with the same probability $1/n$. You collect all n types of coupon by repeating the trials until all types are obtained. Let $C(n)$ be the expected value of the number of trials until the coupon collection is completed. Then, $C(n) = nH(n)$, where $H(n)$ is the nth harmonic number.*

Before proving Theorem 15, we show one lemma. This lemma is, intuitively, quite natural, and many readers may use it as a fact unconsciously. For example, suppose that you roll a die until you obtain 1. On average, how many times do you have to roll the die until you obtain 1? Maybe, since we obtain 1 with probability 1/6, most readers will expect the answer to be around 6 even if you have no reason. The following lemma formalizes this intuition.

Lemma 16 *Suppose that we repeat a trial until it succeeds, and each trial succeeds with probability p. Then, the expected value $E(p)$ of the number of trials until success is 1/p.*

For example, when we regard 1 as a success for rolling a die, the probability is $1/6$, and the expected value of the number of rolls is $1/(1/6) = 6$, which agrees with our intuition.

Proof (of Lemma 16) By assumption, we obtain a success at the first trial with probability p. In other words, we obtain a fail at the first trial with probability $(1 - p)$. Therefore, we obtain a success at the second trial with probability $(1 - p)p$, we have two fails in the first two trials with probability $(1 - p)^2$, and so on. Thus, a success is obtained at the kth trial with probability $(1 - p)^{k-1}p$, and hence, the expected value $E(p)$ can be calculated by the equation

$$\begin{aligned} E(p) &= 1 \cdot p + 2 \cdot (1-p)p + 3 \cdot (1-p)^2 p + \cdots + k \cdot (1-p)^{k-1} p + \cdots && (6.1) \\ &= \sum_{i=1}^{\infty} i \cdot (1-p)^{i-1} p. && (6.2) \end{aligned}$$

Now, we multiply both sides by $(1 - p)$ and obtain

$$\begin{aligned} (1-p)E(p) &= 1 \cdot (1-p)p + 2 \cdot (1-p)^2 p + \cdots + k \cdot (1-p)^k p + \cdots && (6.3) \\ &= \sum_{i=1}^{\infty} i \cdot (1-p)^i p. && (6.4) \end{aligned}$$

Therefore, by subtracting the second equation from the first, we have

$$\begin{aligned} pE(p) &= 1 \cdot p + 1 \cdot (1-p)p + 1 \cdot (1-p)^2 p + \cdots + 1 \cdot (1-p)^{k-1} p + \cdots \\ &= p \sum_{i=1}^{\infty} (1-p)^{i-1}. \end{aligned}$$

The right-hand side of the equation above is a geometric series, and since $0 < (1 - p) < 1$, it converges. Specifically, since $\sum_{i=1}^{\infty}(1-p)^{i-1} = 1/(1-(1-p)) = 1/p$, we obtain $E(p) = 1/p$. □

Now, we turn to the proof of the main theorem.

Proof (of Theorem 15) We first define each state S_i by the state that we have exactly i types of coupon. That is, we start from the state S_0, and the state is changed to S_1 when we draw the first coupon. Eventually, when we reach to the state S_n, we finally complete the collection of all the types of coupon.

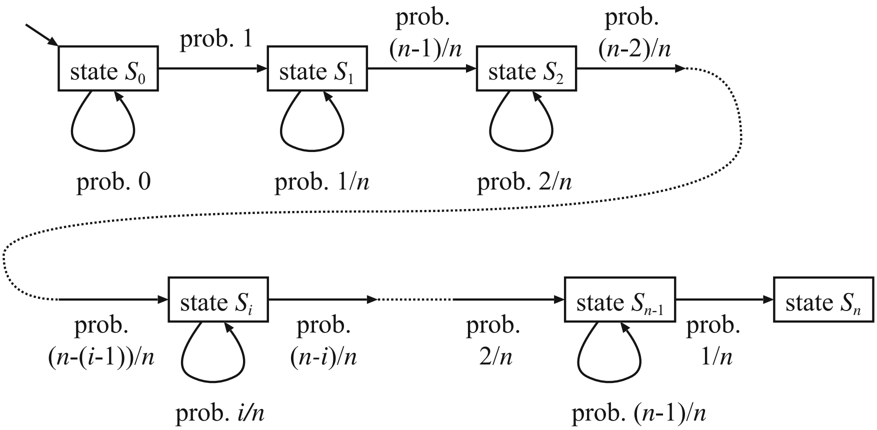


Fig. 6.3 Transition of states

Now, we assume that we are at state S_i and consider what happens when we draw one new coupon. This new coupon is redundant with probability i/n. That is, state S_i remains at S_i with probability i/n. On the other hand, we do not have this new coupon with probability $(n-i)/n$. Therefore, when we are at state S_i, we transfer to S_{i+1} with probability $(n-i)/n$. We consider this transition from S_i to S_{i+1} a "success." That is, from state S_i, we succeed with probability $(n-i)/n$ and transfer to state S_{i+1}. It is sufficient to repeat this process from the start state S_0 to the goal state S_n. This series of transitions is depicted in Fig. 6.3. It is consistent that we consider that state S_0 is transferred to state S_1 with probability 1.

By Lemma 16, the expected value of the number of trials is $n/(n-i)$ if each trial succeeds with probability $(n-i)/n$. In the current context, in the state S_i, the expected value of the number of coupons we will draw to transfer to state S_{i+1} is $n/(n-i)$. Therefore, the expected value $C(n)$ of the number of total trials from the start state S_0 to the goal state S_n is given by the summation of $n/(n-i)$ for each i. Therefore, we obtain the equation[2]

$$C(n) = \sum_{i=0}^{n-1} \frac{n}{n-i}.$$

When i changes from 0 to $n-1$, $n-i$ changes from n to 1. Thus, we introduce a new variable $i' = n - i$, and we obtain

$$E(n) = \sum_{i=0}^{n-1} \frac{n}{n-i} = \sum_{i'=1}^{n} \frac{n}{i'} = n \sum_{i'=1}^{n} \frac{1}{i'} = nH(n),$$

which completes the proof of the theorem. □

[2] **Linearity of expected value**:
As in the discussion in the proof of Theorem 15, we implicitly use the linearity of expected value.

Theorem 15 for the coupon collector's problem is beautiful, and its proof is elegant. The real calculation is easy and involves computing the harmonic number itself, or taking the logarithm as its approximation. For example, when you collect $n = 10$ coupons, the expected value $nH(n)$ is 29.3, and for $n = 100$ coupons, the expected value $nH(n)$ is 519.7. Regardless of whether these values are much less or more than your intuitive estimation, they give some criteria.

Exercise 42 ☕☕☕☕ In the naive algorithm for the shuffling problem, it takes time to find the rth element from the top. Here, we change the idea as follows.

- The variable r always takes a random value from 1 to n.
- If $a[r]$ has been output, we generate r again until we succeed in finding a new one.

We call this algorithm lazy shuffling. The precise algorithm is described as follows.

```
Algorithm 39: LazyShuffle($a$)
  Input  : An array $a[]$
  Output: The array $a[]$ that is randomly shuffled
for $i \leftarrow 1, 2, \dots, n$ do
    $b[i] \leftarrow 0$;
end
for $i \leftarrow n, n-1, \dots, 2, 1$ do
    $r \leftarrow \lfloor random() \times n \rfloor + 1$ ;          /* We have $1 \le r \le n$ */
    while $b[r] > 0$ do
        $r \leftarrow \lfloor random() \times n \rfloor + 1$ ;      /* Repeat if it fails */
    end
    output $a[r]$;
    $b[r] \leftarrow 1$;
end
```

The algorithm seems to be really lazy; it just picks up uniformly at random until it succeeds. It is not difficult to see that it eventually works correctly. However, how efficiently does this lazy algorithm work? Evaluate the expected value of the running time of this algorithm. Compare the result with the running time of the naive algorithm.

Chapter 7
References

Abstract In this chapter, I introduce some supplementary and useful books for readers who would like to study the subject in more depth.

7.1 For Beginners

"*Math Girls*," Hiroshi Yuki (Author), Tony Gonzalez (Translator), Bento Books, Inc., 2011.

In this book, the details of the analysis of Fibonacci numbers are presented. In the equation of the ith Fibonacci number, the expression $\sqrt{5}$s appears here and there, which dazzles us, where the final number is an integer. The in-depth answer to the natural question as to the source of the expression is given in this book. There exists a very large amount of the literature about Fibonacci numbers; this book gives comprehensive references to them

Readers who would like to know more about machine models and the principles of computation and would also like to gain information literacy will find the following book worth reading.

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_7

“*D is for Digital: What a Well-informed Person Should Know about Computers and Communications,*” Brian W. Kernighan, CreateSpace Independent Publishing Platform, 2011.

This book gives relatively realistic models with their details. Moreover, not only theoretical models, but also a “sense of manner” is presented.

7.2 For Intermediates

As described in the text, there are two major implementations of quick sorting. One of them is introduced in this book. In the following book, the second is described in detail.

“*Algorithms in C*,” Robert Sedgewick, Addison-Wesley, 1998.

Moreover, many other algorithms are also discussed in this book.

In this book, we consider several puzzles in order to learn algorithms. If you prefer such puzzles, you will definitely enjoy reading Martin Gardner’s books.[1] Presently, the writing of the new Martin Gardner mathematical library series is ongoing. The series will consist of fifteen volumes, the first four volumes of which have already been published:

“*The New Martin Gardner Mathematical Library*,” Martin Gardner, Mathematical Association of America, 2008.

For an intermediate programmer who aims at attaining a higher level of proficiency and learning better programming skills, I recommend reading “around” the subject of programming. The following books provide this.

“*Programming Pearls*,” Jon Bentley, Addison-Wesley, 1999.
“*The Mythical Man-Month: Essays on Software Engineering*,” Frederick P. Brooks Jr., Addison-Wesley, 1995.

These books are old, the programming languages discussed are out of date, and the writing style may also be obsolete. However, it is interesting to know that the principles of computation have not changed, and the knowledge and subjects in this field are timeless. Algorithms are a universal theme regardless of generations and machine architecture.

[1] **Martin Gardner (1914–2010)**:
He was an amateur mathematician of worldwide renown and wrote more than 70 books. The books about mathematics, puzzles, and magic are attractive. It has been said that “Martin has turned thousands of children into mathematicians, and thousands of mathematicians into children.” An international conference called “Gathering 4 Gardner” is held in alternate years. These conferences are wonderful meetings, and the invitees include people from world-renowned mathematicians to professional magicians.

7.3 For Experts

In each area, there are books called “bibles.” I introduce some of the bibles for algorithms and related mathematics. As is usual with bibles, they are not easy for beginners to read, but it is worth possessing them since you can refer to them over quite a long period.

> “*Introduction to Algorithms*,” Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson, MIT Press, 2009.

This book is suitable as the bible of algorithms. It is a textbook used at MIT for the course on algorithms. It is a thick book, but on the whole written conscientiously and sympathetically, and hence you can enhance your programming skills if you read it carefully. MIT also actively makes its courses public; this is the origin of OpenCourseWare (OCW).[2] The courses at MIT that use this textbook are also public at this site. It is a good idea to take these courses by online.

> “*The Art of Computer Programming*,” Donald E. Knuth, Addison-Wesley, 2011.

Don Knuth is definitely one of the leaders in (theoretical) computer science. As a lifework, he is writing a series of books about algorithms. He started this series of books, which is intended to consist of seven volumes, in the 1960s. Since then, he has continued to write and update them, and now a part of Volume 4 has been published. Volume 4 consists of some fascicles. Also, the title of Volume 3 is “Sorting and Searching.” The depth of this topic truly appears in this title.

> “*Concrete Mathematics: A Foundation for Computer Science*,” Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, Addison-Wesley, 1994.

Ron Graham is a mathematician credited by the American Mathematical Society as being “one of the principal architects of the rapid development worldwide of discrete mathematics in recent years.”[3] This book focuses on discrete mathematics, which is useful in computer science. In this book, Knuth also includes a careful commentary of the big-O notation. This is a nice textbook about discrete mathematics for readers who like an analysis of algorithms, discrete mathematics, and combinatorics.

[2] **OCW (OpenCourseWare)**:
OCW is one of the frameworks for publishing courses in universities. You can access the Web page of MIT OCW at http://ocw.mit.edu/index.htm and check the courses that use this book as a textbook.

[3] **Ronald Lewis Graham (1935–)**:
Ron Graham is one of the best known American mathematicians who wrote the greatest number of collaborative works with Paul Erdős. He is a specialist in discrete mathematics, and he is a good juggler.

Chapter 8
Answers to Exercises

Answer to Exercise 1: The numbers are expressed in the binary system by 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, 10001, 10010, 10011, 10100 and in the hexadecimal system by 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14. Let us consider the relation between these two expressions. We can present the above in the form of a table, filling the leftmost position with 0s for alignment purposes:

Decimal	Binary	Hexadecimal
00	00000	00
01	00001	01
02	00010	02
03	00011	03
04	00100	04
05	00101	05
06	00110	06
07	00111	07
08	01000	08
09	01001	09
10	01010	0A
11	01011	0B
12	01100	0C
13	01101	0D
14	01110	0E
15	01111	0F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6_8

A careful look reveals that when a carry is generated in the hexadecimal representation from 0F to 10, a carry is also generated in the binary representation from 01111 to 10000. In fact, binary and hexadecimal numbers can easily be converted to each other using the following rules.

> **Binary to Hexadecimal**: split binary digits into groups of four from the least significant bit and replace 0000 to 1111 by their respective hexadecimal equivalent numbers 0 to F.
>
> **Hexadecimal to Binary**: replace each digit from 0 to F with its equivalent 4-bit binary representation.

The internal representation of data and programs in a computer is entirely in the binary system; i.e., they are all expressed as sequences of 0s and 1s. However, this results in a large number of digits, which is difficult for human beings to handle. On the other hand, converting such data to decimal representation would be time-consuming, and the relation is difficult to understand. For these reasons, binary digits are often converted to the hexadecimal representation to facilitate human visualization. The resulting number of digits is small and their relation to binary numbers is straightforward.

Answer to Exercise 2: When our RAM model has n words in its memory, each memory cell consists of $\log n$ bits, which gives us the size of one word. Thus, this memory has $n \log n$ bits in total. Using $\log n$ bits, the CPU can distinguish n cases, which means that this CPU has n different operations. When $n = 65536$, $\log n = 16$. Thus, this CPU is called 16-bit CPU, that has 65536 operations and memory consists of 65536 words, which makes 65536 × 16 bits in total. Similarly, when $n = 2^{64}$, it models so-called 64-bit CPU. This 64-bit CPU can handle 2^{64} words, and each word consists of 64 bits.

Answer to Exercise 3: Follow carefully the contents of the variables throughout algorithm SW. To facilitate your understanding, let x_0 and y_0 be the values given to variables x and y at the start of the algorithm. Then, as a result of the first assignment $x \leftarrow x + y$, the value of variable x is replaced by $x_0 + y_0$. Let us call this value x_1. In other words, $x_1 = x_0 + y_0$. If y_1 is the result of the next assignment, $y \leftarrow x - y$, the value becomes $y_1 = x_1 - y_0 = (x_0 + y_0) - y_0 = x_0$. Therefore, the result of the last assignment $x \leftarrow x - y$ becomes $x_2 = x_1 - y_1 = (x_0 + y_0) - x_0 = y_0$. In other words, the final values of variables x and y are y_0 and x_0. Simply put, this algorithm "swaps the values of variables x and y."

Some readers may think that if we wish to swap the values of variables x, y, we can simply proceed as follows:

Algorithm 40: Algorithm to swap the values of variables x and y (?) Swap?(x, y)

```
Input  : Two data $x$ and $y$
Output: Swapped values of $x$ and $y$
$y \leftarrow x$;
$x \leftarrow y$;
output $x$ and $y$;
```

This is incorrect. If we proceed as above, both variables will contain the initial value x_0 of x. The normal procedure is to use an intermediate variable (e.g., t) to store a temporary value:

Algorithm 41: Algorithm to swap the values of two variables x and y, Swap(x, y)

```
Input  : Two data $x$ and $y$
Output: Swapped values of $x$ and $y$
$t \leftarrow x$ ;              /* temporarily save the value of $x$ in $t$ */
$x \leftarrow y$ ;                             /* put the value of $y$ in $x$ */
$y \leftarrow t$ ;                 /* restore to $y$ the value saved in $t$ */
output $x$ and $y$;
```

The procedure shown in this exercise has the advantage of not needing an intermediate variable. However, this algorithm does not seem to be widely used, maybe because of understandability issues. Moreover, it is worth noting that by using the exclusive-or logical sum (XOR) operation $\bigoplus$ on bits, the equations[1]

```
$x \leftarrow x + y$;
$y \leftarrow x - y$;
$x \leftarrow x - y$;
```

can be written as

```
$x \leftarrow x \bigoplus y$;
$y \leftarrow x \bigoplus y$;
$x \leftarrow x \bigoplus y$;
```

This is a very nice trick and beautiful.

Answer to Exercise 4: In order to show that arrays $p[i, j]$ and $a[k]$ are related by a one-to-one mapping, we will show that k is uniquely determined by i, j, and that the original i, j can be uniquely recovered by k. It is easy to see from equation $k = j \times n + i$ that k is uniquely determined by i, j. Here, when $i = 0$, $j = 0$, we have $k = 0$; moreover, when $i = 0$, $j = 0$, we have $k = 0$, and when $i = n - 1$, $j = m - 1$, we have $k = (m - 1) \times n + n - 1 = nm - 1$. Therefore, within the range where i and j are given $(0 \leq i \leq n - 1, 0 \leq j \leq m - 1)$, k is limited to a certain area $(0 \leq k \leq nm - 1)$. Then, it is sufficient to show that i and j can be uniquely determined from k. If k satisfies $0 \leq k \leq mn - 1$, suppose $j' \geq 0$ is the result of the integer division k/n, where the remainder is $i' \geq 0$. Since i' is the remainder, we have $0 \leq i' \leq n - 1$. Moreover, from $0 \leq k \leq mn - 1$, we have $0 \leq j' \leq m - 1$. Therefore, we have uniquely obtained i' and j' from k that satisfy our conditions.

[1] **XOR(exclusive-or logical sum)**:
This is an operation on 2 bits defined as $0 \bigoplus 0 = 0$, $0 \bigoplus 1 = 1$, $1 \bigoplus 0 = 1$, and $1 \bigoplus 1 = 0$. In the usage introduced in the text, it is applied on each digit of binary sequences that express x and y. The operation does not exert any effect on the neighboring digits and can be performed on each digit independently.

Thus, we have $i' = i$, $j' = j$. As a consequence, $p[i, j]$ and $a[k]$ are related by a one-to-one mapping.

Answer to Exercise 5: It is simply a matter of generalizing the method used for two-dimensional arrays. Let $q[]$ be a three-dimensional array with ranges $0 \leq i \leq n - 1$, $0 \leq j \leq m - 1, 0 \leq k \leq \ell - 1$, accessed as $q[i, j, k]$. On the other hand, let $a[]$ be a one-dimensional array with range $0 \leq h \leq nm\ell - 1$, accessed as $a[h]$. If we consider that this natural number x can be expressed as a number with "pseudo digits," we can consider the first digit i as a number of base n that varies from 0 to $n - 1$, and the next digit j as a number of base m that varies from 0 to $m - 1$. Here, the weight of digit j is n. In other words, if j is incremented by one, in terms of the lowest digit we can think of it as an increment by n. If we generalize this reasoning to one more digit, the last digit k can be thought of as a number of base ℓ that varies from 0 to $\ell - 1$. The point here is that the weight of this digit is nm, i.e., when k is incremented by 1, this is equivalent to an increment of nm on the lower digits. Therefore, as a whole, if we perform a mechanical replacement such as $a[i + j \times n + k \times n \times m]$, this one-dimensional array $a[]$ can be used to realize a three-dimensional array $q[]$. That is, one-dimensional array $a[]$ can be used to express a three-dimensional array $q[]$. We continue optimizing in this way. By continuing the generalization, we can construct a one-to-one relation between a c-dimensional array and a one-dimensional array, for any integer $c > 3$.

Answer to Exercise 6: Consider where the data are stored for $t - s < 0$, i.e., when $t < s$. s was at the top of the queue, and t was in the position where the next element must be stored. As can be seen in Fig. 1.5, in this case, we have that both $Q[s]$, $Q[s + 1], \ldots, Q[n]$ and $Q[1], Q[2], \ldots, Q[t - 1]$ contain data. Therefore, the number of elements in this array split into two is $(t - 1) + (n - s + 1) = n + t - s$. It is worth noting that in general, when data are stored in a continuous array $Q[a]$, $Q[a + 1], \ldots, Q[b - 1], Q[b]$, the number of data is not $b - a$, but rather $b - a + 1$.

Answer to Exercise 7: If we pay attention to the implementation of queues and make a simplification considering that the top element can be fixed, the implementation of stack S will become simpler than that of a queue. Remember that a stack S allows the following operations:

- Adding a new element to S
- Removing the last element from S
- Checking the number of elements that S contains

The difference between a stack and a queue is that in a stack the last element added is the first one to be removed; otherwise, it is the same as a queue. However, if we use the fact that the top element is always $S[1]$, it can be significantly simplified. Here, in order to implement stack S, we prepare an array S of size n ($S[1]$ to $S[n]$) and an entity t that shows the next location at which to store an element. Since the top element is always $S[1]$, no variable is needed to manage this. At this point, the implementation of the following two processes can be significantly simplified:

Initialization: just $t \leftarrow 1$ is sufficient.
sizeof(S) **function**: always returns $t - 1$.

Moreover, the function push(S, x) that adds a new element to the stack can be realized in the same way as in queues:

```
Algorithm 42: Function push(S, x) that adds a new element to the stack
  Input  : Stack S and element x
  Output: None
  if sizeof(S) = n then
      output an overflow alert and halt.
  else
      S[t] ← x;
      t ← t + 1;
  end
```

Since the top element in a stack is fixed, t exceeding n indicates an overflow. Taking this fact into account leads to further simplification. Specifically:

- In a queue, an array of size n accommodates just $n - 1$ elements. A stack does not need this kind of care, and thus, can contain up to n elements. Therefore, the overflow condition is decremented by one.
- Since an array does not need to be arranged in a circular structure, the procedure of "restoring the value t to 1" that is needed by queues becomes unnecessary.

Only the pop function, which retrieves an element from stack S, needs to be slightly changed. Specifically, the implementation must be as follows:

```
Algorithm 43: Function pop(S) that retrieves the last element of a stack
  Input  : Stack S
  Output: Last element y of stack S
  if sizeof(S) = 0 then
      output an alert signal that the queue is empty and halt.
  else
      t ← t − 1;
      return the value S[t];
  end
```

The essential change is that the value $S[t]$ is returned instead of $S[s]$. Since the value t does not need to be restored to 1, this part is simplified. Moreover, the value was saved in variable q; however, here we cut corners and simplified this part.

Answer to Exercise 8: Present the solution step by step, introducing comments after each step.

(1) $2n + 5 = O(n)$: correct. If constants are set as $c = 3, n_0 = 5$, we clearly have $0 \leq 2n + 5 \leq 3n = cn$ for all $n > n_0 = 5$. Therefore, $2n + 5 = O(n)$.

(2) $n = O(2n + 5)$: correct. If we consider $c = 1, n_0 = 1$, for all $n > n_0 = 1$ we have $0 \leq n \leq 2n + 5$, and therefore, $n = O(2n + 5)$.

Some readers may feel uncomfortable with the two statements above. In the O notation, we ignore multiplications by constants, and therefore, both are correct.

(3) $n^2 = O(n^3)$: correct. If we consider $c = 1, n_0 = 1$, we have $0 \leq n^2 \leq n^3$ for all $n > n_0 = 1$, and therefore, $n^2 = O(n^3)$.

(4) $n^3 = O(n^2)$: incorrect. This can be proven by contradiction. Suppose $n^3 = O(n^2)$. Then, there exist positive constants c, n_0, such that if $n > n_0$, then $0 \leq n^3 \leq cn^2$ holds.
However, if we put $n > c + n_0$, then $n > n_0$ and also $n^3 > (c + n_0)n^2 = cn^2 + n_0 n^2 > cn^2$ holds, which is a contradiction. Therefore, no c and n_0 satisfy the conditions.

Contradiction is a good method for demonstrating that something "does not belong to $O(f(n))$." We just conveniently use constants that appear in the definition and make n extremely large. A little experience is needed, but as in the solution above, in many cases it is sufficient to add all constants.

(5) $O(n^2) = O(n^3)$: incorrect. As a rule concerning the O expression, it is not correct to write O in the left-hand part. Apart from this, the sets of functions represented as $O(n^2)$ and $O(n^3)$ do not match, and that is an error in itself. For example, as shown in (4) above, n^3 does not belong to set $O(n^2)$, but does belong to set $O(n^3)$. (As in (3), it is sufficient to consider $c = 1, n_0 = 1$.)

(6) $5n^2 + 3 = O(2^n)$: correct. For $c = 1$ and $n_0 = 8$, using a little knowledge of calculus[2], we find that if $n > 8$, then $5n^2 + 3 < 2^n$.

A little later in this exercise, in Sect. 1.5, we will learn the relations between polynomial functions and exponential functions. This is only a warm-up. As we will note later, as compared to the growth of a polynomial $5n^2 + 3$, the growth of the exponential function 2^n is extremely fast. It is no exaggeration to say that 2^n grows explosively. Here, we will restrict ourselves to noting this fact.

(7) If $f(n) = O(n^2)$, then $f(n) = O(n^3)$: correct. If $f(n) = O(n^2)$, then, from the definition, positive constants c, n_0 exist such that, if $n > n_0$, then $f(n) \leq cn^2$. Here, for the same c, n_0, we have $f(n) \leq cn^2 \leq cn^3$. Therefore, $f(n) = O(n^3)$.

(8) If $f(n) = O(n^3)$, then $f(n) = O(n^2)$: incorrect. It suffices to find a function for which $f(n) = O(n^3)$ holds, but $f(n) = O(n^2)$ does not. Here, we show the above for $f(n) = n^3$. First, $f(n) = O(n^3)$ can be verified by considering $c = 1, n_0 = 1$. It has already been verified that $f(n) = O(n^2)$ is incorrect in (4).

Answer to Exercise 9: In order to prove $A = B$ for two sets A and B, the usual procedure is to prove that both $A \subseteq B$ and $B \subseteq A$ hold.[3] To prove that $A \subseteq B$, it

[2] **Some of information on calculus**:
Here, we vaguely refer to "knowledge of calculus," but what applies in fact is **l'Hospital's rule** For an explanation of l'Hospital's rule, refer to the column at Exercise 13.

[3] **Proof that two sets are equal**:
This general method may seem conventional and somewhat obvious. However, in my experience, most beginners do not know about it.

suffices to prove that when an arbitrary element x is taken from set A, we also have $x \in B$. The above will also be used in this exercise.

First, we prove that $\Theta(g(n)) \subseteq O(g(n)) \cap \Omega(g(n))$. For an arbitrary function $f(n)$ that satisfies $f(n) = \Theta(g(n))$, it suffices to prove that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. According to the definition of Θ notation, positive constants c_1, c_2, n_0 exist and satisfy $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. From $0 \leq c_1 g(n) \leq f(n)$, we have $f(n) = \Omega(g(n))$ and from $0 \leq f(n) \leq c_2 g(n)$, we have $f(n) = O(g(n))$. Therefore, $f(n)$ is an element of set $O(g(n)) \cap \Omega(g(n))$. Next, we prove that $O(g(n)) \cap \Omega(g(n)) \subseteq \Theta(g(n))$. Let function $f(n)$ be an arbitrary element of set $O(g(n)) \cap \Omega(g(n)$. Since $f(n) = \Omega(g(n))$, constants c_1, n_1 exist such that for an arbitrary n that satisfies $n_1 < n$, $0 \leq c_1 g(n) \leq f(n)$ holds. On the other hand, since $f(n) = O(g(n))$, there exist positive constants c_2, n_2 such that for any n that satisfies $n_2 < n, 0 \leq f(n) \leq c_2 g(n)$ holds. Therefore, using these positive constants c_1, c_2 and defining $n_0 = \max(n_1, n_2)$,[4] for these positive constants c_1, c_2, n_0, we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n > n_0$. Therefore, $f(n)$ is an element of set $\Theta(g(n))$. From the above, $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ holds.

Answer to Exercise 10: The computation time expressed in seconds is $n^{50}/(10 \times 10^9)$. If we consider a "realistic time" as "one day," one day contains 60s × 60min × 24 h, and therefore, the solution can be derived as $n^{50}/(10 \times 10^9) = 60 \times 60 \times 24$. In other words, $n^{50} = 60 \times 60 \times 24 \times 10 \times 10^9 = 8.64 \times 10^{14}$. We can calculate it using trial and error on a calculator, but the correct way to solve this type of problem is by calculating the logarithm.[5] Calculating the logarithm of both sides yields

$$\log n^{50} = \log(8.64 \times 10^{14}).$$

Using a property of logarithms, we have

$$50 \log n = \log 8.64 + 14 \log 10.$$

The base of the logarithm can be anything. Here, we consider the natural logarithm with base $e = 2.718\ldots$. A scientific calculator gives us the following numbers: $\ln 8.64 = 2.1564 : \ln 10 = 2.30259$. Therefore, we have

$$\ln n = 34.3927/50 = 0.687854.$$

n is $e^{0.687}$, which is approximately 2. That is, in the case of an algorithm that takes n^{50} steps, for $n = 2$, the computations take approximately one day. On the other hand, as an example, let's estimate the calculation time for $n = 10$. This time, consider x to be the number of years that the computation takes. Then, we can perform the

[4] $\max(a, b)$:
$\max(a, b)$ is a function that takes a and b and returns the larger of them. In other words, if $a > b$, then $\max(a, b) = a$. Otherwise, $max(a, b) = b$.

[5] **Logarithmic function**:
Refer to Sect. 1.5.

calculation

$$n^{50}/(10 \times 10^9) = 365 \times 60 \times 60 \times 24 \times x, \text{ with } n = 10.$$

The result is simply

$$x = 10^{50}/(10 \times 10^9 \times 60 \times 60 \times 24 \times 365).$$

The denominator is

$$10 \times 10^9 \times 60 \times 60 \times 24 \times 365 = 3.1536 \times 10^{17}.$$

Therefore, we can roughly estimate that it takes 3×10^{33} years. It is worth mentioning that the age of the universe is 13.7 billion years, i.e., 1.37×10^{10} years. Even if a polynomial time algorithm can be developed, if the order of the polynomial time is prohibitively large, in practice, it will be of no use.

Answer to Exercise 11: In the naive method, the computation of $a_i x^i$ involves i multiplications. Therefore, the number of additions is d, and the number of multiplications is $\sum_{i=0}^{d} i = d(d+1)/2$. Therefore, the total number of computations is $d + (d(d+1)/2) = d(d+3)/2 = \Theta(d^2)$. On the other hand, if the equation is changed, the number of additions remains d, but the number of multiplications is drastically reduced to d. Therefore, the total number of computations is only $2d$. This kind of simple improvement is the most important part of an algorithm. It is worth noting that this method is called the **Horner method**.[6] There is, however, some controversy as to whether Horner was actually the inventor of this method.

Answer to Exercise 12: Folding a newspaper in half 10 times is impossible using conventional methods.

Folding toilet paper in half

In 2002, American high-school student Britney Gallivan managed to fold a specially prepared 1-km-long piece of toilet paper 12 times. In 2012, in a TV program, high-school students in Massachusetts broke this record, folding the paper 13 times. The piece of paper used on this occasion was 16 km long. After folding, the width of the folded paper was 1.5 m, and its thickness was 76 cm (and of course $2^{13} = 8192$ overlapped layers).

In any rate, it can be understood easily by actually trying to fold a newspaper in half 10 times. According to my measurement, the thickness of a newspaper sheet is approximately 0.1 mm. If we fold it 10 times, the number of layers must be $2^{10} = 1024$. In other words, the thickness becomes approximately 10 cm. It is very difficult to fold a sheet of newspaper successively until its thickness reaches 10 cm.

[6] **William George Horner:1786–1837**:
British mathematician, lecturer.

By the way, how many times must a sheet of paper 0.1 mm thick be folded so that its thickness becomes 3776 m? It is sufficient to solve an integer number n that satisfies

$$2^{n-1} < 3776 \times 1000 \times 10 \leq 2^{n}.$$

We can use logarithms, but even if we calculate based on trial and error, it is not difficult to obtain the result

$$2^{25} = 33554432 < 37760000 < 67108864 = 2^{26}.$$

In other words, just folding a newspaper sheet 26 times makes its thickness exceed the height of Mount Fuji. Furthermore, it is not difficult to apply the above to the distance of 380 thousand km to the moon. Using the magic of exponential explosion, we have that the value satisfying the equation

$$2^{n-1} < 380000 \times 1000 \times 1000 \times 10 \leq 2^{n}$$

is

$$2^{41} = 2199023255552 < 3800000000000 < 4398046511104 = 2^{42}.$$

After folding it such that its thickness exceeds the height of Mount Fuji, 16 more folds make the thickness of the newspaper equal to the distance to the moon. To continue folding a sheet of paper in half is more difficult than developing space rockets or space elevators. This hopefully illustrates the weirdness of exponential explosion.

Answer to Exercise 13: If you use actual graph paper to make the drawings, you will probably fail very soon. The problem is that the increase in $f(x) = 2^x$ is prohibitively fast. Here, we divide the graph into two parts. Figure 8.1 contains the overlapped plots of $f(x) = \log x$ and $f(x) = x$, and Fig. 8.2 contains the overlapped plots of $f(x) = 2^x$ and $f(x) = x$. The objective of this exercise is that you will notice that function $\log x$ remains close to the x axis even when x is increased. On the other hand, function 2^x increases significantly, even for small changes in x. Could you observe this?

Answer to Exercise 14: Let us first consider the case for a distance equal to 4. As in the main text, we need to find n that satisfies

$$1 + \frac{1}{2} + \cdots + \frac{1}{n-1} < 4 \leq 1 + \frac{1}{2} + \cdots + \frac{1}{n}.$$

I reached the following for $n = 31$ after laborious work with a calculator:

$$1 + \frac{1}{2} + \cdots + \frac{1}{30} = 3.99499\ldots < 4 < 4.02725\ldots = 1 + \frac{1}{2} + \cdots + \frac{1}{30} + \frac{1}{31}.$$

Therefore, carefully overlapping 31 cards results in 2 cards being eliminated. How many cards must be eliminated in order to increase the distance to 10? In principle,

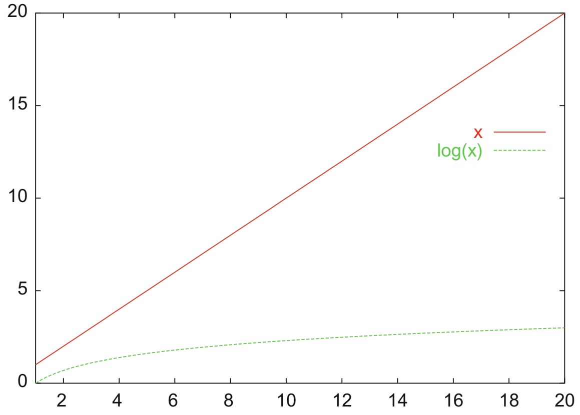


Fig. 8.1 Overlapped plots of functions $f(x) = x$ and $f(x) = \log x$

for such n, we have

$$1 + \frac{1}{2} + \cdots + \frac{1}{n-1} < 10 \leq 1 + \frac{1}{2} + \cdots + \frac{1}{n}.$$

However, it seems that simply using the calculator is not an intelligent approach. Let us perform the estimation using a smarter method. As previously mentioned in the present text, function $H(n)$ is roughly $\ln n$ in a harmonic series. In other words, the number we are considering is such that $H(n)$ is 10; that is, we are looking for n such that $\ln n$ is approximately 10. If we solve the equation $\ln n \approx 10$ for integer n, we have $n \approx e^{10} \approx 22026$. (Here, e is the base of the natural logarithm, $e = 2.71828\ldots$.) In other words, approximately 22000 cards must be stacked. It would be a wise decision

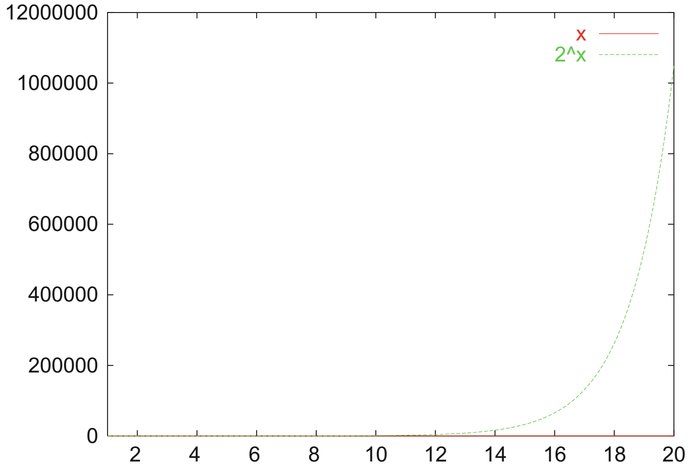


Fig. 8.2 Overlap graphs of functions $f(x) = 2^x$ and $f(x) = x$

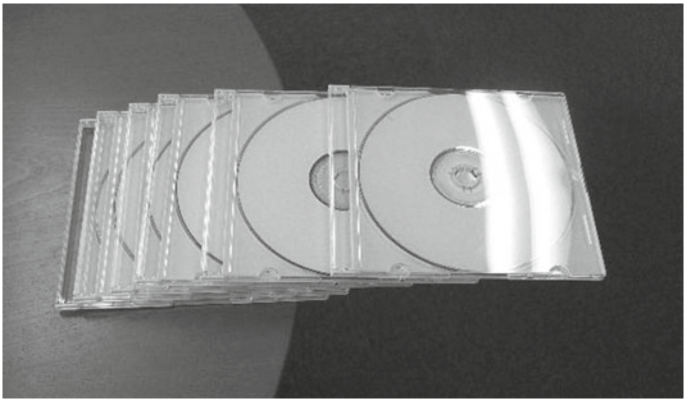


Fig. 8.3 DVD’s stacked and projected outward from a table

to give up both on performing the numerical calculation and actually preparing the cards. Both tasks seem too difficult.

However, the result mentioned at the beginning of this text, according to which four cards can be used to eliminate one, is worth trying. It is very hard to achieve this using four cards, but using five or six cards, it can be done easily, as shown in Fig. 8.3. It is fun and, I recommend it.

Answer to Exercise 15: It is not easily noticed, but in fact these two graphs are exactly the same but drawn in different ways. It is easy to verify this fact by writing down the edges. The set of edges for both is {{1, 2}, {1, 5}, {1, 6}, {1, 7}, {2, 7}, {3, 4}, {3, 6}, {4, 5}, {4, 6}, {4, 7}, {5, 7}}.

Answer to Exercise 16: If the degree of vertex i is denoted by $d(i)$, the sum of the degrees of the graph in Fig. 1.9 is $d(1) + d(2) + d(3) + d(4) + d(5) + d(6) + d(7) = 4 + 2 + 2 + 4 + 3 + 3 + 4 = 22$. On the other hand, the number of edges is 11, as can be seen in the answer to Exercise 15. Therefore, we have $22 = 2 \times 11$.

Answer to Exercise 17: If you understood the proof of Theorem 3 correctly, the directed graph version of Theorem 3 can be easily understood. When the degree of each vertex is incremented, each directed edge is counted once as an in-degree and once as an out-degree. Therefore, in the directed graph, the following beautiful equality holds:

$$\text{Sum of in-degrees of vertices} = \text{sum of out-degree vertices} = \text{number of edges.}$$

Answer to Exercise 18: Denote again the operation of moving a disc k from rod i to rod j as $(k; i \to j)$. When Hanoi(4, 1, 3) is called, Hanoi(3, 1, 2), (4; 1 → 3), Hanoi(3, 2, 3) are executed in order during the algorithm. In the above, Hanoi(3, 1, 2) corresponds to Hanoi(2, 1, 3), (3; 1 → 2), Hanoi(2, 3, 2). In the above, Hanoi(3, 2, 3) corresponds to Hanoi(2, 2, 1), (3; 2 → 3), Hanoi(2, 1, 3). For example, Hanoi(2, 1, 3) corresponds to Hanoi(1, 1, 2), (2; 1 → 3), Hanoi(1, 2, 3), which in turn is (1; 1 → 2), (2; 1 → 3), (1; 2 → 3). If we manage to put everything together, Hanoi(4, 1, 3) can be written as follows: (1; 1 → 2):(2; 1 → 3):(1; 2 → 3): (3; 1 → 2): (1; 3 → 1):(2; 3 → 2):(1; 1 → 2): (4; 1 → 3): (1; 2 → 3):(2; 2 → 1):(1; 3 → 1): (3; 2 → 3): (1; 1 → 2):(2; 1 → 3):(1; 2 → 3): In other words, $H(4) = 15$.

Arranging the values obtained so far, we have: $H(1) = 1$, $H(2) = 3$, $H(3) = 7$, and $H(4) = 15$. A careful analysis of the Hanoi(n, 1, 3) algorithm yields that Hanoi($n - 1, i, j$) is internally called twice, and the disc is moved once. Therefore, it seems reasonable to multiply the previous value by 2 and add 1. In other words, the conclusion is that it is something close to 2^n. In that sense, if we look again at $H(1)$ to $H(4)$, we have the impression that, in general, $H(n) = 2^n - 1$ holds. Please refer to the main text for a proof.

Answer to Exercise 19: From the previous estimation and the proof in the main text, we learnt that $H(n) = 2^n - 1$. Therefore, the number of moves required to move 64 discs is $H(64) = 2^{64} - 1$. If it takes 1 s to move a single disc, the total necessary time is $2^{64} - 1$ s. Since one year contains $60 \times 60 \times 24 \times 365 = 31536000$ s, dividing the above by this number yields approximately 5.85×10^{11} years. In other words, this amounts to about 6 trillion years. Considering that the age of the universe is 13.7 billion years, regardless of the year in which we start moving the discs around, the end of the world seems to be quite far away.

Answer to Exercise 20: Using my laptop PC, the execution took a long time for $n = 40$, and for $n = 42$ we can still tolerate the wait. The level $n = 45$ seems to be the limit of our patience. For $n = 50$, it seems that the answer will never emerge.

Answer to Exercise 21: Using my laptop PC, the answer appeared immediately for n in the order of a few thousands. Instead of the computation time, the numerical overflow in the variables became an issue sooner.

Answer to Exercise 22: This works well with a standard mathematical induction, but you must note the fact that there are two basic steps. The following proof puts some emphasis on this point.

First, the proposition to be proven is that for a sequence, the general term $F(n)$ of which is defined as

$$\begin{aligned} F(1) &= 1 \\ F(2) &= 1 \\ F(n) &= F(n-1) + F(n-2) \quad (\text{for} n > 2) \end{aligned}$$

the following equation holds:

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right).$$

As the basic steps, verify that for $i = 1$ and $i = 2$, $F(1) = 1$ and $F(2) = 1$ hold. **For** $i = 1$: $F(1) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1\right) = \frac{1}{\sqrt{5}}\sqrt{5} = 1$, and therefore it holds. **For** $i = 2$: we have $F(2) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^2\right) = \frac{1}{\sqrt{5}}\left(\left(\frac{6+2\sqrt{5}}{4}\right) - \left(\frac{6-2\sqrt{5}}{4}\right)\right) = \frac{1}{\sqrt{5}}\sqrt{5} = 1$, and therefore it holds.

Then we will go to the general case. Here, for $n > 2$, we hypothesize that both the following equations hold:

$$F(n-1) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right)$$

and

$$F(n-2) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right).$$

It is important here to emphasize that both equations are considered to hold; otherwise, the induction will not work properly. If we use this hypothesis, we transform it into the following equation, using the definition first, and then $F(n) = F(n-1) + F(n-2)$. Therefore, from the hypothesis above, we can place each of them in a closed equation, obtaining the following:

$$\begin{aligned} F(n) &= F(n-1) + F(n-2) \\ &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right) \\ &\quad + \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right). \end{aligned}$$

After some work, the above can be arranged as

$$\begin{aligned} F(n) &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right) \\ &= \frac{1}{\sqrt{5}}\left(\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\right) - \left(\left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right)\right). \end{aligned}$$

Here, some work has to be done in view of our goal. Eventually, we want to obtain $\left(\frac{1+\sqrt{5}}{2}\right)^n$ and $\left(\frac{1-\sqrt{5}}{2}\right)^n$, but we cannot obtain it in this way. As an example, since the above equations contain the term $\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}$, we can make a guess and try to calculate $\left(\frac{1+\sqrt{5}}{2}\right)^2$ and $\left(\frac{1-\sqrt{5}}{2}\right)^2$. We obtain the following strange equations:

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = 1 + \frac{1+\sqrt{5}}{2}$$

and

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2} = 1 + \frac{1-\sqrt{5}}{2}.$$

Here, using the above the equation can be changed as follows:

$$\begin{aligned}
F(n) &= \frac{1}{\sqrt{5}}\left(\left(\left(\tfrac{1+\sqrt{5}}{2}\right)^{n-1} + \left(\tfrac{1+\sqrt{5}}{2}\right)^{n-2}\right) - \left(\left(\tfrac{1-\sqrt{5}}{2}\right)^{n-1} + \left(\tfrac{1-\sqrt{5}}{2}\right)^{n-2}\right)\right) \\
&= \tfrac{1}{\sqrt{5}}\left(\left(1 + \left(\tfrac{1+\sqrt{5}}{2}\right)\right)\left(\tfrac{1+\sqrt{5}}{2}\right)^{n-2} - \left(1 + \left(\tfrac{1-\sqrt{5}}{2}\right)\right)\left(\tfrac{1-\sqrt{5}}{2}\right)^{n-2}\right) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^2\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^2\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right).
\end{aligned}$$

Thus, we have proven the desired relation. Note that an equation manipulation that looks like magic took place in the second half. In fact, this keeps a close relation with the Fibonacci sequence and golden ratio. We will leave this topic for the next answer.

Answer to Exercise 23: The golden ratio ϕ is a constant defined as $\phi = \frac{1+\sqrt{5}}{2}$, which in practice is about 1.61803. It is also one of the solutions of equation $x^2 - x - 1 = 0$. A vast number of studies in the literature and prior research have been devoted to the relation between the Fibonacci sequence $F(n)$ and the golden ratio ϕ. Many books have addressed the fascination and magic of this subject. For details, please refer to Chap. 7.

Here, we focus, in particular, on the relation to the answer to Exercise 22. ϕ is one solution of the equation $x^2 - x - 1 = 0$. The second solution is $\frac{1-\sqrt{5}}{2}$. That is, both solutions satisfy $x^2 = x + 1$. Note that this is the explanation of the trick that appeared at the end of Exercise 22, where $\phi + 1$ is replaced by ϕ^2.

Regarding Fibonacci sequences $F(n)$, it is also known that $\lim_{n\to\infty} \frac{F(n)}{F(n-1)}$ converges to ϕ. This is the basis for the statement in the text when n is sufficiently large, $F(n)$ becomes $\Theta(\phi^n)$.

Answer to Exercise 24: The execution time of the block search was $\Theta(1/d + dn)$, in other words, the sum of two terms, $1/d$ and dn. If we ignore the multiplication by a constant, we can think of it as the average of the sum of two terms (or $(1/d + dn)/2$).

For a given constant c, if we consider $d = n^c$, $1/d + dn$ is $n^{-c} + n^{1+c}$, where the former is a decreasing function of c and the latter is an increasing function of c. Careful analysis leads to the conclusion that the balance point, i.e., when $-c = 1 + c$, is where the minimum of this function occurs. Solving this yields $c = -1/2$, which taken to the original equation results in $d = 1/\sqrt{n}$. In terms of algorithmic operations, the above means finding the balance between the guessing operation in the initial part and the one-by-one verification of the latter part. The conclusion is that the best efficiency is achieved when adjustments are made so that both take approximately the same time. If this balance is disrupted, the total performance tends to deteriorate. As extreme examples, we can perform calculations for the cases $d = 1$ and $d = n/3$ and confirm whether either of the functions becomes a bottleneck, so that the total result does not become smaller than $\Theta(\sqrt{n})$.

Answer to Exercise 25: Make sure that you actually implement it as a program in a real programming language. Following the description in this book, the implementation will be as follows:[7]

```
Algorithm 44: Binary search BinarySearch($a, i, j, x$)
  Input  : Check whether $x$ is in the range from $a[i]$ through $a[j]$ in array $a[]$
  Output: Output the index if it is found. If not, output a message saying so
 1 if $i \geq j$ then            /* in case only up to one element exists */
 2     if $a[i] = x$ then
 3         output index $i$ and halt;
 4     else
 5         output “not found” and halt;
 6     end
 7 end
 8 $m \leftarrow \lfloor (i + j)/2 \rfloor$; ;   /* if $(i + j)$ is odd, throw the fraction away */
 9 if $a[m] = x$ then
10     output index $m$ and halt;
11 else
12     if $a[m] > x$ then
13         BinarySearch($a, i, m - 1, x$);
14     else
15         BinarySearch($a, m + 1, j, x$);
16     end
17 end
```

If the subroutine above is implemented prior to the execution of BinarySearch $(A, 1, n, x)$, checking whether element x is in array A and outputting the corresponding index can be performed very quickly.

Note that in the process of finding m, any fractional part is truncated. In fact, rounding up would be essentially equivalent. For instance, if we have an odd number

[7] **Reason why $i \geq j$ in line 1**:
If there is only one data item, $i = j$ holds. Therefore, $i = j$ instead of $i \geq j$ apparently works. However, this is not true. Consider carefully the case where only two data exist from $a[i]$ to $a[j]$, i.e., $j = i + 1$. In this case, we have $m = i$, and for $a[m] > x$, we would call BinarySearch$(a, i, i - 1, x)$. In other words, when the number of data is 0, a function call occurs with $j = i - 1$. The condition $i \geq j$ is checked in order to enable correct processing in this case, too.

of elements such as $i = 1$, $j = 5$, the equation yields $(i + j)/2 = 3$, which is the central value with no fractional part. On the other hand, if we have an even number of elements such as $i = 1$, $j = 6$, truncating would yield $m = 3$, and rounding up would yield $m = 4$, and both cases would work well. In C language, for example, truncating is easier to program, and thus in this text, we opted for truncating.

Answer to Exercise 26: Since $\log 240000 = 17.8727$, if we consider that the number of words in the dictionary is 240,000, we can reach any word by looking up 18 words. When looking up a word, it is an efficient method to open a page located exactly in the middle every time. A binary search takes care of reducing the time it takes. It is also fun to compete with an electronic dictionary in terms of speed.

Answer to Exercise 27: In merge sort, two arrays are merged every time. In this case, the two arrays belong to a continuous area within the original array. Let $a[i], \ldots, a[j]$ and $a[j + 1], \ldots, a[k]$ express this case. When they are merged, suppose two equal elements $a[i']$ and $a[j']$ exist. Also, suppose that $a[i']$ is an element in $a[i], \ldots, a[j]$, and $a[j']$ is an element in $a[j + 1], \ldots, a[k]$. Here, if $a[i']$ is merged with higher priority (i.e., so that it comes first), it becomes a stable sort. (Going into details, if the value immediately after $a[i']$, $a[i' + 1]$ is also equal to $a[j']$, $a[i' + 1]$ must also be merged with a higher priority than $a[j']$.)

Answer to Exercise 28: Quick sort is a kind of divide and conquer approach, where sorting is performed by dividing the problem into small subproblems. This means that solving the subproblem itself does not have to be done using quick sort. Then, we can perform divide and conquer using quick sort if the number of elements of the given array is large, and then, switch to bubble sort when the number of elements becomes small. The original implementation of QuickSort (a, s, t) was "do nothing if it contains 0 or 1 element." We can change this step to "do bubble sort if the number of elements is k or less." The actual value of k may vary depending on the implementation. In my experiments, the value was around 20 or 30. In a recursive call, the last step always involves splitting the problem into small problems. Therefore, the global effect of improving this part is considerable.

The fact that this type of "best of both worlds" is effective is a characteristic of the divide-and-conquer method.

Answer to Exercise 29: We just need to modify the last output part and write back to $a[]$. Namely, in the following part:

```
for $j \leftarrow 1, 2, \ldots, m$ do
    output $j$ $b[j]$ times;
end
```

another counter k is set in addition to index i of $a[]$, and

```
$i \leftarrow 1$;
for $j \leftarrow 1, 2, \ldots, m$ do
    for $k \leftarrow 1, \ldots, b[j]$ do
        $a[i] \leftarrow j$;
        $i \leftarrow i + 1$;
    end
end
```

Then, instead of being displayed as the output, the value is written back to $a[]$.

Answer to Exercise 30: Evaluate the time computation of this algorithm. Dry spaghetti can be cut in a fixed amount of time per unit. Bunching the strands and standing them on a table takes another fixed amount of time, and removing them one by one also takes a fixed amount of time per unit. Therefore, this is a $\Theta(n)$ time algorithm.

However, it seems that there is no chance in frequency with which this algorithm is used in practice. Nobody will build robots to speed up the operation either. This is probably due to the following factors:

- Execution is theoretically performed in linear time $\Theta(n)$, but in practice, the number of constant coefficients is too large.
- Implementation becomes difficult if the number of data or the absolute value of each data item is large.
- One should not waste food in this way.

It should be noted that calculation methods based on this type of physical measurement seem to be breaking the rules, but they deserve attention this kind of problem. For instance, if a metallic frame is dumped into soap water, a soap membrane is formed. This membrane is an optimal one that satisfies certain conditions. As another example, if a large number of nails are hammered onto a wooden board and they are wrapped in a rubber band, a geometrical figure called a **convex hull** can be easily obtained. Geometrical computation of the shapes of membranes and convex hulls is not a trivial matter. In recent years, new machine models that differ from the previously existing ones, such as DNA computers and quantum computers, have been proposed in an active research trend. Research on new machine models can be viewed as an attempt to actively incorporate "rule-breaking" approaches to exceed the boundaries of existing computers.

Answer to Exercise 31: The problem is to find from which vertex the remaining vertex 5 is visited. In the original DFS algorithm, vertices are visited in the following order before stopping:

Vertex 1 → Vertex 2 → Vertex 4 → Vertex 3 → Vertex 6 → Vertex 7.

The algorithm without the [Yes] line also follows the same vertex visiting order. After visiting vertex 3 and then vertex 6 and vertex 7, the algorithm returns to vertex 4 and looks for unvisited vertices. More exactly, the set of neighboring vertices of vertex 4 is $A_1[4, 1] = 2$, $A_1[4, 2] = 3$, $A_1[4, 3] = 5$, $A_1[4, 4] = 6$. After returning here, since vertex 2 and vertex 3 have already been checked, we now check vertex $A_1[4, 3] = 5$. Then, since $V_1[5] = 0$, we realize that this vertex has not yet been visited and decide to visit vertex 5. In other words, the search tree is obtained by attaching edge $\{4, 5\}$ to the tree of Fig. 4.3.

Answer to Exercise 32: For instance, if we change the meaning of array $C[]$ as follows, array $D[]$ becomes unnecessary.

If $C[i] = -1$: the vertex i has never entered the queue.
If $C[i] = j \geq 0$: the vertex i has already entered the queue, and the distance from the vertex 1 to the vertex i is j.

In this case, the BFS algorithm can be rewritten as follows.

```
2 if $i = n$ then
3     output the value of $C[i]$ and halt;
4 end

6 if $C[A[i, j]] = -1$ then
7     push($A[i, j]$);
8     $C[A[i, j]] \leftarrow C[i] + 1$;
9 end
```

Moreover, line 5 of the main body (the assignment operation to array $D[]$) becomes unnecessary. It is true that the clarity of the algorithm is slightly sacrificed because $C[]$ has two meanings. If clarity is to be improved at the expense of increasing waste, this improvement may turn out to be a degradation, and care must be taken. The improvement proposed here does in fact result in simplification and can truly be considered an improvement, provided that the meanings of the variables are well documented.

Answer to Exercise 33: If we examine the original algorithm carefully, we can observe that a new vertex is visited (along the shortest path) when array $C[]$ or $D[]$ is updated. Therefore, the path can be memorized at this point. Then, how should the path be memorized? we will introduce an array $F[]$ that stores the piece of information regarding "from which vertex," whenever a vertex is visited for the first time. In other words, it means that if $F[i] = j$, we know that in the shortest path to vertex i, the immediately previous vertex is j. Vertex 1 is the starting point, and no immediately previous vertex exists. For practical purposes, consider $F[1] = 0$. In this algorithm, array $D[]$ is unnecessary, and the available space can be used to update array $F[]$. Thus, information related to the path itself instead of the length can be stored.

First, the following line of the main body of the algorithm

```
5 $D[1] \leftarrow 0$;
```

can be replaced by the initialization of $F[1]$:

```
5 $F[1] \leftarrow 0$;
```

Next, in the same way, the statement

```
9 $D[A[i, j]] \leftarrow D[i] + 1$;
```

of the BFS(A, Q) algorithm is replaced by

```
9 $F[j] \leftarrow i$;
```

It is easy to see that with the above changes it is possible to store the displacement path from vertex i to vertex j. The problem now is how to change the following line of the BFS(A, Q) algorithm:

```
3 output the value of $D[i]$ and halt;
```

What can we do? At this point, we have that:

- In $F[i] = F[n]$, the value of the previous vertex before arriving at vertex n (e.g., i') is recorded;
- In $F[i'] = F[F[i]] = F[F[n]]$, the value of the vertex immediately before arriving at vertex i' (e.g., i'') is recorded.
- In $F[i''] = F[F[i']]$, the value of the vertex before arriving at vertex i'' is recorded.

That is, the vertices are arranged in reverse order starting from the end. When $F[i] = 0$, eventually, we have $i = 1$, and this should correspond to the starting point. Therefore, if we want to output the path in reverse order, we have to change this line in the following way (note that the numbering of lines will change from now on, but ignore that).

```
3 output $n$, which is the last vertex;
4 while $F[i] > 0$ do
5     output $F[i]$;
6     $i \leftarrow F[i]$;
7 end
8 output 1, which is the first vertex.
```

With these changes, the followed path can be output in reverse order, i.e., the shortest path from the target to the starting point.

The problem can be considered to be solved, but also consider how to reverse this path for the output. Readers who immediately imagined a stack structure are now familiar with the topic. Using a stack structure, the order can be reversed very simple as explained below. Let S be a stack.

```
push($S$, $n$);
while $F[i] > 0$ do
    push($S$, $F[i]$);
    $i \leftarrow F[i]$;
end
push($S$, 1);
while sizeof($S$) $> 0$ do
    $i \leftarrow$ pop($S$);
    output $i$;
end
```

I expect that the validity of this algorithm became clear for the readers at this point.

Answer to Exercise 34: First we demonstrate the validity of the algorithm to find the shortest path. We use the induction method for the length of the shortest path. A shortest path length equal to 0 means that vertex $1 =$ vertex n, which corresponds to the trivial case $n = 1$. Here, first consider the case where the path length is 1, which happens when vertex n is a neighbor of vertex 1. Let's study the validity of the algorithm by carefully following it step by step. First, initialization is done in BFS-main. Here, vertex 1 is placed in queue Q, and BFS(A, Q) is called. In BFS, i is set to 1 by $i \leftarrow \text{pop}(Q)$. Currently, $i = n$ is not true, and therefore, lines 2 to 4 are not executed. The execution then jumps to line 5 onwards. Inside the for loop in lines 5 to 11, all vertices that are adjacent to vertex $i = 1$ go to queue Q. Vertex n is also placed in queue Q at this time. Then, returning to BFS-main, the same processing is repeated again and again from the beginning until Q becomes empty. Therefore, vertex n ends up being removed from Q and becomes caught in line 2 of BFS. When that occurs, $D[i] = 1$ is output and the process stops.

Next, let's suppose that the length of the shortest path is correctly obtained until $k(\geq 1)$, and that the distance up to vertex n is $k + 1$. In this case, at least one vertex n' of distance k which is adjacent to vertex n must exist. By applying the induction hypothesis to vertex n', this vertex n' will eventually come out of Q in one of the calls to BFS(A, Q) and will be placed in i. In the same way, if we continue examining the execution of the algorithm for i, we notice that vertex n is placed in queue Q, and that it will eventually be output and undergo comparison in one of the following calls. Distance $k + 1$ is correctly calculated in this case as well.

From the discussion above, we conclude that the validity of the algorithm has been demonstrated.

Next, consider the execution time of this algorithm. As in the DFS demonstration, if we consider each edge individually, we observe that each edge is taken at most twice. Regarding vertices, once a given vertex enters queue Q, it enters the processing when its turn arrives, and thereafter does not enter the scene again. Therefore, the total execution time can be kept at $O(n + m)$, where n is the number of vertices and m is the number of edges.

Table 8.1 Number of solutions to the n queen problem

n	1	2	3	4	5	6	7	8	9	10	11	12	13
$Q(n)$	1	0	0	2	10	4	40	92	352	724	2680	14200	73712

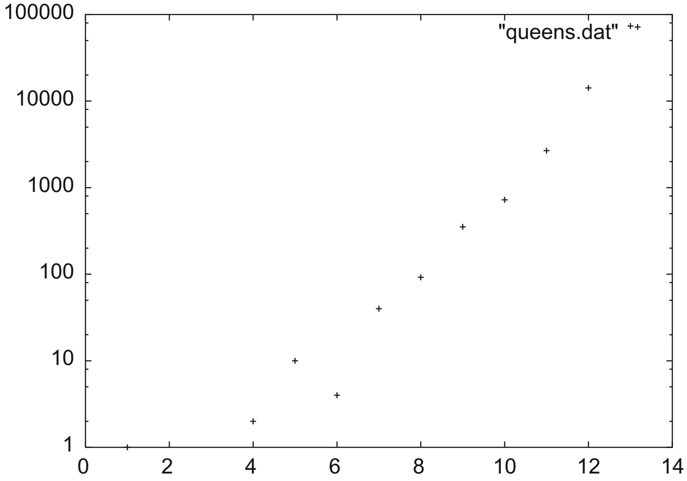


Fig. 8.4 Graph showing the number of solutions to the n queen problem: note that the vertical axis is in logarithmic scale

Now, we add a few comments about the **for** loop in line 5 of BFS(A, Q). In this loop, the neighboring vertices of vertex i are taken out one by one in order as $A[i, j]$. Since we were considering data structures of "neighboring sets," taking out a single vertex takes only a fixed amount of time. If we were considering "neighboring arrays" instead, this part could not be processed with such efficiency, and therefore, the upper limit of computation time would deteriorate. The importance of improving data structures becomes evident in this case.

Answer to Exercise 35: Regarding the n queen problem for a board of size $n \times n$, designate the number of solutions as function $Q(n)$. From the observations and execution results in the main text, we obtain $Q(1) = 1$, $Q(2) = 0$, $Q(3) = 0$, and $Q(8) = 92$. If a program that takes the size of the board as a parameter is written, other values besides the ones above can easily be calculated. For instance, $Q(4) = 2$. This is the solution shown in Fig. 5.3, as well as a flipped version of it. Likewise, $Q(5) = 10$, $Q(6) = 4$ and $Q(7) = 40$. Other values were calculated as well and are shown in Table 8.1. They are plotted in the graph of Fig. 8.4. The graph is difficult to interpret if we use conventional scales for the x and y axes. For this reason, the y axis is in logarithmic scale. The fact that the curve of this graph is a straight line indicates that the original function behaves like an exponential. Therefore, we conjecture that $Q(n)$ is an exponential function of n.

Answer to Exercise 36: If we look at the flow of the check subroutine, we realize that a new vertex is visited upon every recursive call. When the search finishes, this fact is forgotten. Therefore, we just have to record or forget the vertices visited at that particular time. A careful consideration of the flow of this type of recursive call leads to the conclusion that the data structure to be used should be a stack. Using a stack S, when the check subroutine performs $b[i] \leftarrow 1$, it also does push(S, i), and when it performs $b[i] \leftarrow 0$, it does pop(S) as well. Moreover, after $b[0] \leftarrow 1$ in 6Knight, making it perform push$(S, 0)$, our recording task is perfectly accomplished. For actually outputting the solution, we need to execute the following process:

- Remove all the contents of the stack and output it in reverse order.
- Restore the contents of the stack.

This can be easily implemented if we use an additional stack. Another alternative, which in fact is an irregular or illegal way of using stacks, is to output the contents of array S from the beginning of S. This is an easy way of outputting the solution.

Answer to Exercise 37: I could not predict the results of this exercise before attempting it. The initial, optimistic, thought was that at least one solution could be found in a realistic time. An initial warning is that preparing data is a bit painful. The former part was input by hand while checking several times. The latter part was performed more easily using symmetry. The following process was performed.

```
Algorithm 45: Initializing an 8 × 8 board Init(a)
  Input  : 2 dimensional array a[]
  Output: Initialize it so that it forms correct neighboring sets
  for all $0 \le i < 64, 0 \le j < 9$, set $a[i, j] \leftarrow -1$;
  initialize from $a[0, *]$ to $a[31, *]$ with data prepared by hand;
  for all combinations $32 \le i < 64, 0 \le j < 9$ if $a[63 - i, j]$ is not $-1$, do
    $a[i, j] \leftarrow 63 - a[63 - i, j]$;
```

A careful look at the data shows us that the data in the first half are reversed and copied to the second half. Thanks to this operation, the data input work was reduced by half. After initialization has been completed, all solutions can, in principle, be obtained by expanding to 8 × 8 boards, the same program used for 6 × 6 boards. To my surprise, the first result was found in less than a minute. This memorable first solution is shown in Fig. 8.5(1).

Then, the PC was left processing the program overnight, and the obtained solutions were recorded. In approximately 18 hours, 62596 solutions had been obtained. At this time, I decided to stop the program. The solution is shown in Fig. 8.5(2).

A careful looking at the two solutions reveals that most of the initial parts are essentially the same. The initial common part is shown in Fig. 8.5(3). We can see that for a total number of 64 steps, the steps are the same up to number 22. In other

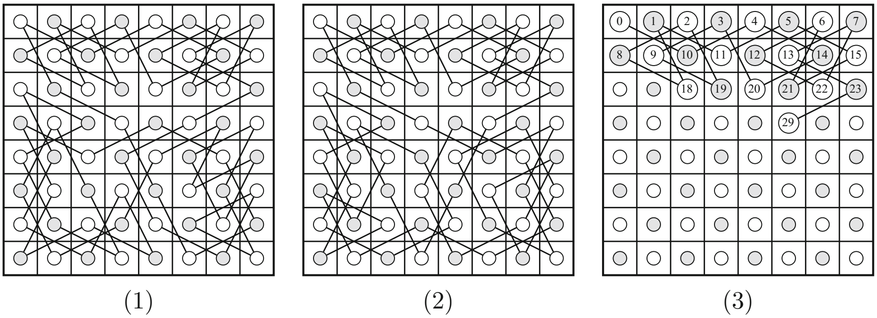


Fig. 8.5 (1) The first solution found in less than 1 minute; (2) the 62596th solution found after about 18 hours; (3) the part shared by the two solutions

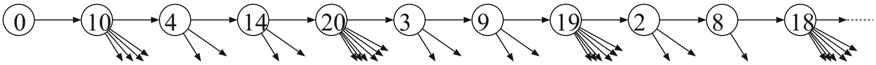


Fig. 8.6 Structure of the initial part of the search tree

words, it shows that the search trees are the same up to depth 22. An aspect of this search tree is shown in Fig. 8.6.

We will now try to roughly estimate the total amount of computation. Even considering that the edges from the first vertex 0 to vertex 10 are fixed, if we look at the neighboring vertices of vertex 10, i.e., vertices 4, 16, 20, 25, and 27, the search related to vertex 4 (the first in the list) has not been finished yet. That is, $1/5$ of the search has not been finished for this part. As to the next vertex 4, the search of the first in the group of neighboring vertices 14, 18, 20 has not yet been finished, which corresponds to $1/3$ of the search. In the same way, focusing on vertex 14, the first vertex in the group of the neighboring vertices 20, 29, and 31 has not been searched, and focusing on the next vertex 20, the first vertex in the group of neighboring vertices 3, 5, 10, 26, 30, 35, and 37 has not been searched, and so on. Roughly, consider that each time we face three alternatives, and that the search related to the first alternative up to depth 20 has not been finished. In this case, we can say that roughly about $(1/3)^{20}$ of the entire search has been finished. Considering that thus far 18 hours have been spent, the total search time is 18×3^{20} hours, which converted into years yields $(18 \times 3^{20})/(24 \times 365) = 7.16 \times 10^6$ years, that is, approximately 7 million years. It is true that this represents a significant improvement over the first rough estimation, but this is still too long to wait.

Great algorithms!

Do you know the **OEIS (On-Line Encyclopedia of Integer Sequences)** Web site[a]? This site is a database of known integer sequences. This interesting Web site contains many data related to integer sequences, and one can perform a search on several types of sequences. For instance, if we input 0, 0, 9862 in this database, we learn that it has already been stored as the number of solutions to the knight's tour problem. Moreover, we even learn that the number of solutions to the 8 × 8 board problem was already calculated back in 1997. The actual number is 13267364410532. How could a calculation that would take millions of years using a modern computer be done in 1997? This is the power of algorithms. Thinking of ways to speed up calculations is a good exercise for the brain.

[a] http://oeis.org.

Answer to Exercise 38: Suppose a real number r is a nonnegative uniformly random value smaller than 1. That is, we have $0 \leq r < 1$. In this case, the statement $0 \leq kr < k$ regarding a natural number k holds. If we truncate the fractional part, we obtain value 0 for $0 \leq kr < 1$, value 1 for $1 \leq kr < 2$, …, and value $k - 1$ for $k - 1 \leq kr < k$. For instance, for the $k = 9$ case depicted in Fig. 8.7, the interval $[0, 1)$ where the initial value r exists is amplified $k = 9$ times to become $[0, k)$.[8] If we consider $[i, i + 1)$ as an interval where the leftmost point is included and the rightmost point is not, all intervals will be of the same size. Therefore, if r is a uniform random variable, the probability that kr is included in interval $[i, i + 1)$ in the figure is also uniform.

Answer to Exercise 39: Despite being trivial, this problem is worth careful consideration. First, suppose there are several ways of arranging the numbers from 1 to n. This is a factorial problem and can be calculated as $n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$. In other words, it is just a question of showing that a sequence of numbers from 1 to n appears with probability $1/(n!)$. There is no special meaning in the order in which numbers are shown, and we can choose the tangible example $1, 2, 3, 4, \ldots, (n - 1), n$. Consider the probability of outputting these values. The probability of outputting 1 in the beginning corresponds to $r = 1$. This outcome is one out of uniform probabilities from 1 to n, and therefore, the probability of $r = 1$ is $1/n$. Next, the probability of outputting the smallest number 2 out of the numbers that have not already been output also corresponds to $r = 1$. This time, the outcome is selected from the uniform probabilities 1 to $n - 1$, and therefore, the probabil-

[8] **Notation to designate intervals**:
An important point when it comes to expressing intervals along a number line is whether or not the extremities are included. The standard mathematical notation is to use [] for intervals that contain the extremities and () for those that do not. In other words, [0, 1] means that this interval contains numbers greater than or equal to 0 and smaller than or equal to 1. On the other hand, (0, 1) means that the interval contains numbers greater than 0 and smaller than 1. Likewise, the interval $[i, j)$ means that its values are equal to or greater than i, and smaller than j.

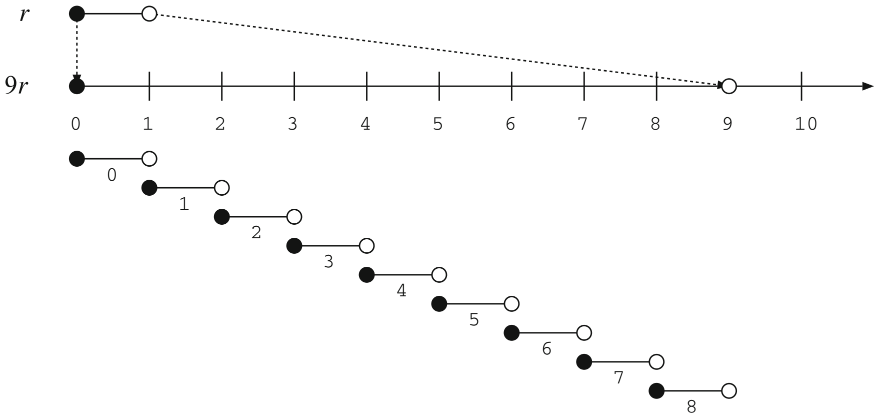


Fig. 8.7 Random numbers in the $0 \leq r < 1$ interval made nine times larger and split into nine intervals

ity corresponding to $r = 1$ is $1/(n-1)$. Likewise, the probability of outputting 3 in the third place also corresponds to $r = 1$, that is, the probability is $1/(n-2)$. If we continue this process, the probability of obtaining the sequence of numbers $1, 2, 3, 4, ..., (n-1), n$ is

$$\frac{1}{n} \times \frac{1}{n-1} \times \cdots \times \frac{1}{3} \times \frac{1}{2} \times \frac{1}{1} = \frac{1}{n!}.$$

Note that the choice of strings of characters is arbitrary. If we consider a specific sequence of numbers $n, (n-1), ..., 3, 2, 1$, even though the values of r in $r = n$, $r = n - 1, \ldots$ may change, their occurrence probability will not. Then, it is possible to see that any sequence of numbers will appear with the same probability $1/n!$.

Answer to Exercise 40: In this book, when we mention storing data, we refer to arrays, queues, or stacks. If the task is to store data instead of outputting them and to write them back at the end, queues are convenient, because they retain the original order. Modify actually the algorithm using queue Q. The last two lines of

```
13 output $a[j]$;
14 $b[j] \leftarrow 1$;
```

can be changed in the following way.

```
13 push($Q, a[j]$);
14 $b[j] \leftarrow 1$;
```

Moreover, since it is necessary to write data back at the end, we just have to add the following lines after the **for** loop in lines 4 to 15 (after overwriting 13):

```
for $i \leftarrow 1, 2, \ldots, n$ do
    $a[i] \leftarrow \mathrm{pop}(Q)$;
end
```

Take this opportunity to evaluate the computational cost of this algorithm. First, let's consider time complexity. Computation time does not change up to the first **for** loop. In other words, this algorithm is $O(n^2)$ until that part. On the other hand, since the computation time in the loop added at the end is $O(n)$, the total amount is $O(n^2)$ and remains unchanged. In terms of space complexity, even though a queue Q is used, space complexity does not change when it comes to considering overall order $O(n)$.

Answer to Exercise 41: As in Exercise 39, fix sequence $1, 2, 3, 4, \ldots, (n-1), n$ as a concrete example. Then, consider the probabilities of outcome by an elegant algorithm. A careful consideration indicates that this is quite similar to the calculation of Exercise 39, leading to the same results.

Then, why is it that a naive method and an elegant method yield different outputs for the same input and the same random sequence? There are two reasons. First, the meaning of r differs. In the naive method, we search for the r-th element in array $a[]$ that has not yet been output. In the elegant method, on the other hand, $a[r]$ is directly designated. This difference leads to the different results. Another point is that in the elegant method the elements in the array are swapped. In other words, swapping the last element causes a disruption in the order of elements. The data stored in $a[i]$ in the first place may be moved to another place. For the two reasons above, the data extracted for the same r may differ, leading to different final results.

Answer to Exercise 42: In LazyShuffle, if we view "elements that have not been output yet" as "coupons not obtained yet," it can be directly applied to the coupon collector problem. Therefore, the expectation value of computation time is $O(nH(n))$. The naive solution required $O(n^2)$ using a simple program. In comparison, $O(nH(n))$, that is, $O(n \log n)$ represents a significant improvement. This difference is comparable to the difference between bubble sort and quick sort. With the naive solution, it is possible to achieve $O(n \log n)$ using sophisticated data structures, but the program would become too complex and would not be very advantageous. In comparison, achieving such a significant speedup with such a simple algorithm is amazing. It is worth noting, however, that probability-based algorithms may, in some cases, require computation times larger than the expected value.

Afterword

Computers have become tools we cannot ignore in our lives in modern society, whether we like them or not. Even though some people may say "we don't use computers," but a person "living without a mobile phone" is a rarity. Mobile phones have a computation power equivalent to that of a supercomputer twenty years ago. If we think about it, we realize that we live surrounded by an unprecedented number of computers running sophisticated programs everywhere. In a train, for example, it is commonplace to see seven computers being used on a seat for seven people in Japan.

These programs are written by programmers. It is a known fact that the productivity of programmers shows large variations. It is not rare that a program written by a good programmer is 10 times more productive than one written by an ordinary programmer. What then is the secret for writing good programs? Is it a matter of coding techniques? Yes, it is. Is it the quality of tools? This does not seem to be exactly the point. There are actually two means that allow anyone to write better programs.

The first is to read good programs. This is equivalent to the maxim that "to write good texts, it is necessary to read good texts." It gives the programmer the ability to achieve improved superficial readability of the program and the facility to debug, but further improvement becomes difficult. This is parallel to the situation where just learning the format does not enable a person to write good texts. A good format without substance does not make a text attractive. Being able to write programs according to a grammar is pointless if the programs lack substance. What then is the substance of a program?, It is the algorithm. Behind a good algorithm is a great idea. For instance, the mechanism of recursive calls and the divide-and-conquer idea dramatically simplify certain problems and facilitate their solution. This is because these algorithms correctly express the essence of a problem, enabling us to correctly grasp it.

The second means of learning to write good programs is to study algorithms. Computer science is a relatively new field of science. Algorithms and the surrounding subject areas constitute topics for active research, and advances are occurring at a

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6

fast pace. Numerous algorithms are proposed in worldwide to deal with a variety of problems. These algorithms, we can say, are tools for solving problems. When we face a problem and find a tool that fits it, the problem can be solved straightforwardly. Furniture can be assembled with pliers, but a set of screwdrivers of various sizes allows us to do it in a smarter, faster, and more beautiful way. Preparing tools is the shortest path for solving problems.

In this book, which is just a preliminary text, we give only a short introduction to the subject. Numerous other excellent algorithms have been proposed. Learning a large number of algorithms following the references for further reading shown in Chap. 7 will enable you to solve several problems in a smart way.

When we think about solving a problem efficiently, the "essence" of the problem, which had not been seen until then, becomes clearly visible. This is similar to the moment when we solve a puzzle and it gives us a peculiar feeling of satisfaction. Above all, in good algorithms a useful idea is embedded and they have a kind of beauty that is not immediately understood. If this book helps you to cultivate an admiration for this beauty, it would be an unexpected joy for me.

In spite of my best effort, a few mistakes may still remain in this book, just as in any other book. Amendments and supplementary information will be made public on the Web site

http://www.jaist.ac.jp/~uehara/books/IntroAlg/index.html.

Moreover, programs that involve data collection, which are often found to be more exhausting than expected, are also available at the above site. Please copy and use these data as appropriate. Besides solving a problem and creating a good algorithm, another peculiar pleasure is seeing your program running.

My friend, Erik D. Demaine, obtained his Ph.D. at the age of only 20 and became a professor at the world's top-class university, MIT. His father, Martin L. Demaine, did not receive a higher education, but he educated his son at home. This is the so-called home schooling, which is sometimes seen in North America. At a banquet of a certain international conference, I had a chance to drink and chat with Martin about child education, and I asked him the secret to bringing up a genius. There are two secrets according to him. The first is that he taught Erik that "learning is fun." Yes, learning is fun. It is my hope that you had a joyful learning experience through reading this book.[1]

October 1st, 2018 Ryuhei Uehara

[1] **The secret of bringing up a genius**:
What about the second secret? I'll keep it as a secret in this book. It is indeed a secret. I'll whisper the answer only those who come to ask me face-to-face.

Index

R. Uehara, *First Course in Algorithms Through Puzzles*,
https://doi.org/10.1007/978-981-13-3188-6