Algorithms
for Competitive Programming

David Esparza Alba
Juan Antonio Ruiz Leal

January 26, 2021

To Junue and Leonardo
– David Esparza Alba

To my parents Tito and Ma. del Refugio
– Juan Antonio Ruiz Leal

Contents

List of Figures

Listings

About the Authors

David Esparza Alba (Cheeto) got his bachelor degree in Electronic Engineering from Universidad Panamericana, and a master degree in Computer Science and Mathematics from the Mathematics Research Center (CIMAT) where he focused his research in artificial intelligence, particularly in evolutionary algorithms. Inside the industry he has more than 10 years of experience as a software engineer, having worked at Oracle and Amazon. In the academy he has done research stays at Ritsumeikan University and CIMAT, focusing his investigations in the fields of Machine Learning and Bayesian statistics. He frequently collaborates as a professor at Universidad Panamericana, teaching mainly the courses of Computer Programming, and Algorithm Design and Analysis.

Juan Antonio Ruiz Leal (Toño) got involved in competitive programming during high school, he obtained silver and bronze medals in the Mexican Informatics Olympiad (Olimpiada Mexicana de Informática - OMI) in 2009 and 2010 respectively. Then, as a Bachelor Student he continued in the field and was part of one of the teams that represented Mexico in the ACM ICPC World Finals in 2014 and 2015. He studied Computer Engineering at Universidad Autónoma de Aguascalientes where he graduated with honors. He has coached students for the Mexican Informatics Olympiad and for the Mexican Mathematics Olympiad. In the industry he has worked as a Software Engineer at Oracle, and currently he is pursuing his master degree at Heidelberg University in the Interdisciplinary Center for Scientific Computing focusing on Deep Learning algorithms.

Preface

Computer Programming is something that has become part of our daily lives in such a way that it results natural to us, it is present in our smart phones, computers, TV, automobiles, etc. Some years ago the only ones that needed to know how to program were software engineers, but today, a vast of professions are linked to computer programming, and in years to come that link will become stronger, and those who know how to program will have more opportunities to develop their talent.

Nowadays more and more companies, and not only in the software industry, need to develop mobile applications, or create ways to improve their communications channels, or analyze great amount of data in order to offer their customers a better service or a new product. Well, in order to do that, computer programming plays an indispensable role, and we are not talking about knowing just the commands of a programming language, but being able to think and analyze what is the best way to solve a problem, and in this way transmit those ideas into a computer in order to create something that can help people.

The objective of this book is to show both sides of the coin, on one side give a simple explanation of some of the most popular algorithms in different topics, and on the other side show a computer program containing the basic structure of each algorithm.

Who Should Read this Book?

The origin of this book started some years ago, when we used to save files with algorithms used in programming competitions for

problems that we considered interesting. After that, these same algorithms that we learnt were useful to get jobs in the software industry. Thus, based on our experience, this book is intended for anyone looking to learn some of the algorithms used in programming contests, coding interviews, and in the industry.

Competitive Programming is another tool for software engineers, there are developers that excel in their jobs, but find difficult to answer algorithms questions. If you feel identified with this, then this book can help you improve your problem solving skills.

Prerequisites

This book assumes previous knowledge on any programming language. For each algorithm it is given a brief description of how it works and a source code, this with the intention to put on practice the theory behind the algorithms, it doesn't contain any explanation of the programming language used, with the exception of some built-in functions.

Some sections contain college-level math (algebra, geometry and combinatorics), a reasonable knowledge of math concepts can help the reader to understand some of the algorithms in those sections.

Structure of the book

The book consists on 10 chapters, and with the exception of the first chapter, the rest contains a section with exercises, and the solutions for those exercises are located at the end of the book. Also at the end of each chapter there is a section called *"Chapter Notes"*, where we mention some references and bibliography related to the content of the corresponding chapter.

The first chapter is just a small description of some of the different programming contests and online judges that are available. Some of those contests are directed to a specific audience and all of them have different rules. On the other hand, online judges are a place to improve your programming skills, some of them are oriented to some specific area, there are some that focus more in mathematics, others in logical thinking, etc. We encourage the

reader to take a look to all of the online judges listed in the chapter and try to solve at least one problem in each one of them.

Chapter 2 is an introduction of fundamental topics, such as Computer Complexity, Recursion, and Bitwise operations, which are frequently used in the rest of the book. If the reader is already familiar with these topics, then this chapter can be skipped.

Chapter 3 covers different data structures, explaining their properties, advantages and how to implement them. Depending on the problem definition some data structures fit better than others. The content in this chapter is of great importance for the rest of the chapters and we recommend to read this chapter first before moving to others.

Chapter 4 is about *Sorting Algorithms*, containing some of the most popular algorithms, like Bubble Sort, Selection Sort, and others that run faster, such as Heap Sort, and Merge Sort. We also mention other methods like Counting Sort, which is an algorithm to sort integer numbers in linear time.

Chapter 5 talks about an important technique called *Divide and Conquer*, which allow us to divide a problem in easier subproblems. An useful tool when dealing with large amount of data.

In chapter 6 we review some of the most popular problems in *Dynamic Programming*, which more than a tool, is an ability that is developed by practice, and is closely related to recursion.

Chapter 7 is all about *Graph Theory*, which is one of the areas with more applications, from social networks to robotics. Many of the problems we face daily can be transformed to graphs. For example, to identify the best route to go from home to the office. We can apply the algorithms in this chapter to solve these kind of problems and more.

Chapter 8 focuses on mathematical algorithms, specially on geometric algorithms. Here we explain algorithms to solve some of the most frequent problems, like finding the intersection of two lines, or identifying if a point is inside a polygon, but also we describe more complex algorithms like finding the convex hull of a cloud of points.

The 9^{th} chapter is about *Number Theory* and *Combinatorics*, two of the most fascinating topics in mathematics and also in algorithms. Number theory deals with properties of numbers, like divisibility, prime numbers, sequences, etc. On the other hand, combinatorics is more about counting in how many ways we can obtain a result for a specific problem. Some applications of these topics are password security and analysis of computational complexity.

The last chapter is dedicated to *String Processing*, or *String Manipulation*. Which consists on given a string or a set of strings, manipulating the characters of those strings in order to solve a problem. Some examples are: find a word in a text, check whether a word is a palindrome or not, find the palindrome of maximum length inside a string, etc.

Online Content

The source code of the exercises and appendices can be found in the GitHub page:

`https://github.com/Cheetos/afcp`

The content of the repository will be updated periodically with new algorithms and solutions to problems, but feel free to contact us if you think that a specific algorithm should be added, or if you have an interesting problem that you want to share.

1

Introduction

1.1 Programming Contests

Programming contests are great places to get involved in algorithms and programming, and there is a contest for everyone. For people from 12 to 18 years old, perhaps the most important contest is the *Olympiad of Informatics*, which consists in solving different problems using logic and computers. Each country organizes preliminaries and a national contest, then they select four students to participate in the *International Olympiad of Informatics* that takes place every year. The following link contains the results, problems, and solutions of all contests that have taken place since 1989.

`http://www.ioinformatics.org/history.shtml`

For college students there is the *ACM-ICPC (ACM - International Collegiate Programming Contest)*. Here, each team consists of three students and one coach. There is only one computer for each team, and they have five hours to solve a set of problems. The team that solves more problems is the winner, and in case of a tie, the amount of time they needed to solve the problems is used to break the tie. There is a penalty for each incorrect submission, so be careful to check every detail before sending a solution.

As in the Olympiad of Informatics, in the *ACM-ICPC* there are regional contests, and the best teams of each region participate in the international contest. Results from previous contests and problems sets can be consulted in the official website of the contest.

`https://icpc.baylor.edu/`

For graduate students or professionals, there are other options to continue participating in programming contests. Nowadays some of the biggest companies in the software industry and other organizations do their own contests, each one with their own rules and prizes. Some of these contests are the **Facebook Hacker Cup**, and the **Google Code Jam**. Also **Topcoder** organizes some contests that include monetary prizes.

1.2 Coding Interviews

Is well known that the most important companies in the software industry have a well structured and high-selective hiring process, which involves testing the candidate's knowledge of algorithms. This has caused an increase in the amount of material intended to help future candidates to succeed in technical interviews. This book differentiates in the aspect that its content is written by software engineers with teaching experience that have been trough multiple recruiting processes, not only as candidates, but also as interviewers.

Some advice that we can give to anyone planning to enter in an interview process are:

- Prepare for your interview, each company has its own culture, its own hiring processes, so try to learn about this before the interview.
- If you are going to study for an interview, be confident on your strengths and focus on your weaknesses.
- Moments before the interview try to relax. At the end you should be able to enjoy the interview no matter the outcome.
- Independently from the verdict, the result from an interview is always positive.

1.3 Online Judges

Online judges are websites where you can find problems from different categories, and where you can submit your solution for any of

those problems, which then is evaluated by comparing it with test inputs and outputs, and finally a result is given back to you. Some of these judges contain previous problems from ACM competitions, from Olympiads of Informatics, and from other contests. Without any doubt, online judges are one of the best places to practice and improve your coding skills. Some of the most popular online judges are:

- **Leetcode.** One of the most popular sites to train for job interviews in top tech companies. Contains problems from different categories.
- **Project Euler.** Focused on mathematics, there is no need to send a source code, only the answer to the problem.
- **Codeforces.** One of the best online judges to practice, contains problems from different categories, and they frequently schedule competitions. There are also great tutorials and discussions about solutions.
- **UVA Online Judge.** Here you can find problems of any kind, and there are more than 4000 problems to choose from.
- **HackerRank.** A popular online judge to start coding, they have a path so you can start with easy problems and then move to more complicated ones.
- **Timus.** This page does not have as many problems as other online judges, but the quality and complexity of the problems make it an excellent option to improve your math and programming skills.
- **CodeChef.** This platform is useful to prepare students for programming competitions, and for professionals to improve their coding skills. It has a large community of developers and supports more than 50 programming languages.
- **Topcoder.** Contains tutorials explaining with great detail different algorithms. When solving a problem the points gained from solving it decrease as the time goes by, so if you want to improve your coding speed, this is the right place.
- **OmegaUP.** Excellent tool to teach computer programming. It is very easy to create your own problems and there is also a large data base of problems to solve.

1.4 C and C++

The programming language chosen to write the source code of the algorithms in this book is `C++`, along with some functions of `C`. Even when there are other popular languages like Java and Python, we chose `C++` because it is still one of the more robust languages in the market, and with the use of the *Standard Template Library* (STL), we can implement more complex data structures, manipulate strings, use predefined algorithms, etc.

Another reason we chose `C++` is because it runs faster than other languages, for example, in some cases `Java` can be 10 times slower, and Python is even more slow than that. In programming contests, speed is something that matters. Nevertheless, the optimal solution should run inside the time limits of any problem independently of the language used.

Something that the reader will notice is that in some occasions we use `C` functions to read the input data and write the output data. `C++` has `cin` and `cout` to handle the input and output respectively, but they are slow in comparison with `C` functions `scanf`, and `printf`. Even so, `C++` functions can be optimized by adding the following lines at the beginning of the `main` function

```
cin.tie(0);
ios_base::sync_with_stdio(0);
```

Unfortunately we already have the "bad" habit of using `C` functions for read and write, but it is not only that, `scanf` and `printf` provides great flexibility to handle the input and output of your program.

1.5 Chapter Notes

Every programming language has its own advantages and disadvantages, and it is good to know or at least have a notion of those. For this book we chose `C/C++` as our main language, because of its simplicity and all the libraries and capabilities it contains. One of the things that makes `C/C++` special is the way it handles the input and output. It is just amazing how easy we can read and write data using `scanf` and `printf` for `C`, and `cin` and `cout` for `C++`. We personally find reading and writing in other programming languages not as intuitive, at least at first.

`C++` also gives us all the advantages of Object Oriented Programming (OOP), and the **STL** library, which contains algorithms, data structures, string manipulation capability, etc.

Other programming languages have powerful tools for problem solving. For example in the case of *Java* we have found very useful the `BigInteger` class, which allows us to do operations with large numbers. Another example is *Python*, with its simple syntax, it allows to write solutions using few lines of code.

2

Fundamentals

"First, solve the problem. Then, write the code."

– John Johnson

In this chapter we will review important concepts and techniques that are indispensable to understand and implement most of the algorithms contained in next chapters. We make emphasis in three concepts. Recursion, Algorithm analysis, and bitwise operations.

Recursion is a vastly used technique that at first is not that easy to understand. In fact, it can look like some kind of magic, but the truth is that it is not that complex, and it is a very powerful tool, as some problems are impossible to solve without recursion.

Algorithm Analysis helps us to identify how good, or bad is an algorithm for a certain problem, because depending on the data some algorithms will fit better than others.

Finally, the section about bitwise operations will help us to understand how operations are made at bit level. Remember that all calculations in your computer are binary, so everything is translated to 0's and 1's, and there are operators that allow us to do operations directly on the bits, and as we will see, that can save us not only running time, but also coding time.

2.1 Recursion

Recursion is a very powerful tool, and many of the algorithms contained in this book use it. That is why we decided to add a brief description of it. If you are already familiar with the concept of recursion, then you can skip this section.

When a function is called inside the same function, we said that this function is a recursive function. Suppose there is a function $f(n)$, which returns the factorial of a given number n. We know that:

$$
\begin{aligned}
f(0) &= 1 \\
f(1) &= 1 \\
f(2) &= 1 \times 2 \\
f(3) &= 1 \times 2 \times 3 = f(2) \times 3 \\
f(3) &= 1 \times 2 \times 3 \times 4 = f(3) \times 4 \\
&\vdots \\
f(n) &= 1 \times 2 \times \ldots \times n = f(n-1) \times n
\end{aligned}
$$

We see that the factorial of n can be obtained by multiplying n and the factorial of $n-1$, and the factorial of $n-1$ can be obtained using the factorial of $n-2$, and so on, until we reach 0!. So we say that 0! is the base case, since it is generating the rest of the factorials. If we see this as a recursive function, it would look like 2.1.

Listing 2.1: Factorial by Recursion

```
1 int f(unsigned int n) {
2   if (n == 0) {
3     return 1;
4   }
5
6   return n * f(n - 1);
7 }
```

The validation to check if n is zero is critical, because without it the recursion would never stop, and the memory of our computer will eventually crash.

Every time that a function is recursively called, all the memory it uses is stored on a heap, and that memory is released until the

function ends. For that reason it is indispensable to add a stop condition, and avoid ending in an infinite process. Not really infinite, because our computer will crash before that.

To resume, there are two fundamental parts that every recursive function must have:

1. The function must call itself inside the function.
2. A stop condition or base case should be given in order to avoid an infinite process.

2.1.1 Memoization

Memoization is a way to improve recursion. It is a technique that consists in storing in memory values that we have already computed in order to avoid calculating them again, improving that way the running time of the algorithm.

Let's do an example in order to see the importance of using memoization. Consider the recursion function in 2.2, which computes the n^{th} Fibonacci number. The first two Fibonacci numbers are 1, and the n^{th} Fibonacci number is the sum of the two previous Fibonacci numbers, for $n \geq 2$.

Listing 2.2: Fibonacci with Recursion

```
1 int f(int n) {
2   if (n == 0 || n == 1) {
3     return 1;
4   }
5
6   return f(n - 1) + f(n - 2);
7 }
```

Now, suppose we want to obtain the 5^{th} Fibonacci number. The procedure of the recursion is shown in figure 2.1, and as we can see, some Fibonacci numbers are computed multiple times, for example, the 3^{rd} Fibonacci number is calculated twice, and the 2^{nd} Fibonacci number is calculated three times. In other words, we are doing the same calculations over and over. But if we use an array to store the values of all the Fibonacci numbers already computed and use those values instead of going deeper in the recursion tree, then we will avoid executing the same operations all over again and that will improve the running time of our code.

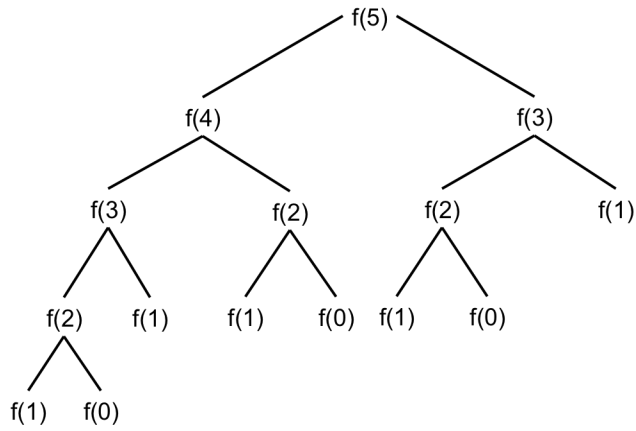


Figure 2.1: Fibonacci Recursion Tree

Consider the array `Fibo` which initially contains only 0's, and will be used to store the Fibonacci numbers. The recursion function using memoization would look as follows:

Listing 2.3: Fibonacci with Memoization

```
1  int f(int n) {
2    if (n == 0 || n == 1) {
3      return 1;
4    }
5
6    if (Fibo[n] == 0) {
7      Fibo[n] = f(n - 1) + f(n - 2);
8    }
9
10   return Fibo[n];
11 }
```

In the code showed in 2.3 we notice that we only call the recursive function if the value of `Fibo[n]` is zero, which means that we have not yet calculated the n^{th} Fibonacci number. Otherwise we return the already calculated value stored in `Fibo[n]`.

2.2 Algorithm Analysis

The complexity of an algorithm can be defined as how the performance of the algorithm changes as we change the size of the input data, this is reflected in the running time and in the memory space of the algorithm. We can expect then that as we increase the size

of the input our algorithm will take more time to run and will need more memory to store the input data, which is certainly the case most of the time. It is extremely important to know the performance of an algorithm before implementing it, mainly to save time for the programmer. For example, an algorithm which is efficient to sort ten elements can perform poorly sorting one million elements and if we know that in advance we could search for other alternatives when the size of the input is large instead of going straight to the computer and realize that in fact this algorithm will take forever after spending valuable time in the implementation.

2.2.1 Asymptotic Notations

We use the so-called "big O notation" to measure the running time of an algorithm, we say that our algorithm runs in $O(f(n))$, if the algorithm executes *at most* $c \cdot f(n)$ operations to process an input of size n, where c is a positive constant and f is a function of n. We can assume that each operation takes one unit of time to execute in the computer, $10^{-9}s$ (one nanosecond) for example, and use number of operations and units of time interchangeably. For instance, we say that an algorithm runs in $O(n)$ time if it has linear complexity, i.e. the relation between the size of the input and the number of operations that the algorithm performs is linear, in other words, the algorithm executes *at most* $c \cdot n$ operations to finish, given an input of size n. Another typical example is $O(\log n)$, which means that the algorithm has logarithmic complexity, i.e. the algorithm takes *at most* $c \cdot log(n)$ operations to terminate when the input size is n. The O-notation is an asymptotic bound for the running time of an algorithm, but it is not the only way of describing asymptotic behaviors. Below we list three different types of notations used to describe running times of algorithms.

- **O-notation.** Used to define an asymptotic upper bound for the running time. It is employed to describe the worst case scenario.

- **Ω-notation.** Define an asymptotic lower bound for the running time of an algorithm. In other words, it is used to describe the best case scenario. We can get further intuition for the Ω-notation by replacing the words "*at most*" by "*at least*" in the previous paragraph.

- **Θ-notation.** Specifies both, an asymptotic lower bound, and

an asymptotic upper bound for the running time of an algorithm.

Most of the time we care more about the O-notation than about the other two because it tells us what will happen in the worst case scenario and if it covers the worst case it covers all the other cases. In practice we usually omit the words "*at most*" and simply say that our algorithm runs in $f(n)$.

Trough the rest of the book we will mostly use the O-notation when describing the time complexity of an algorithm. It is important to mention that even when asymptotic notations are commonly associated to the running time, they can also be used to describe other characteristics of the algorithm, such as, memory.

There are different kinds of algorithms depending on their complexity. Some of the most common belong to the following categories:

- **Constant Complexity.** Their performance does not change with the increase of the input size. We say these algorithms run in $O(1)$ time.

- **Linear Complexity.** The performance of these algorithms behaves linearly as we increase the size of the input data. For example, for 1 element, the algorithm executes c operations, for 10 elements, it makes $10c$ operations, for 100, it executes $100c$ operations, and so on. We say that these algorithms run in $O(n)$ time.

- **Logarithmic Complexity.** The performance increases in a logarithmic way as we increase the input data size. Meaning that the if we have 10 elements, the algorithm will execute $\log 10$ operations, which is around 2.3. For 1000 elements, it makes around of 7 operations, depending on the base of the logarithm. In most of the cases we deal with base-2 logarithms, where:

$$\log_2 n = \frac{\log n}{\log 2}$$

  Since $1/\log 2$ is a multiplicative factor, then we say that these algorithms run in $O(\log n)$ time.

- **Polynomial Complexity.** For this kind of algorithms their performance grows in a polynomial rate according to the input data. Some of the most famous sorting algorithms, the *Bubble Sort*, runs in quadratic time, $O(n^2)$. A well known algorithm that runs in cubic time, $O(n^3)$ is the matrix multiplication. More complex problems have polynomial solutions with greater degree. In programming contests it is common that a problem with an input of 100 elements still can be solved with a cubic approach, but at the end all depends on the time limits specified by the problem.

- **Exponential Complexity.** These are the worst cases and must be avoided if possible, since for a small increment in the input data, their performance grows considerably. Problems that require exponential solutions, can be considered as the hardest ones. We say that these algorithms run in (a^n) time, for some value of $a > 1$.

2.2.2 Master Theorem

As we have seen trough this chapter, recursion is a powerful tool that allows us to solve problems that would be impossible to solve with an iterative approach. But one inconvenient of using recursion is that sometimes it is not clear to see at first glance what would be the time complexity of an algorithm. For this case, the master theorem specifies three cases that can help us to identify the time complexity of a recursive algorithm.

Let $a \geq 1$ and $b \geq 1$ be constants, and let $f(n)$ be a function. If the time complexity, $T(n)$, of a recursive algorithm has the form

$$T(n) = aT(n/b) + f(n),$$

where a refers to the number of branches that will come out of the current node in the recursion tree and b refers to the data size on each of these branches. Then, it has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, then for all sufficiently large n we have that $T(n) = \Theta(f(n))$.

When we apply the master theorem, what we do is to compare the function $f(n)$ with the function $n^{\log_b a}$, and select the larger of the two. If $n^{\log_b a}$ is larger, then we are in the first case, and then $T(n) = \Theta(n^{\log_b a})$. If $f(n)$ is larger, then we are in case 3, and $T(n) = \Theta(f(n))$. If both functions have the same size, then we are in case 2, and we multiply the function by a logarithmic factor, and get $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.

When we say that $f(n)$ must be smaller or larger than $n^{\log_b a}$, we mean that $f(n)$ must be polynomially smaller or larger than $n^{\log_b a}$ by a factor of n^ϵ, for some $\epsilon > 0$.

For a better understanding of the master theorem let's see some examples.

- $T(n) = T(n/2) + 1$.

  This example represents a *Binary Search*, which is explained in chapter 5. Here $a = 1, b = 2$, and $f(n) = 1$. Then $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Since $f(n) = n^{\log_b a} = 1$, then we are in the second case of the master theorem, and $T(n) = \Theta(\log n)$.

- $T(n) = 2T(n/2) + n$.

  This case represents the behavior of the *Merge Sort*, which is explained in chapter 3. Here we have $a = 2, b = 2$, and $f(n) = n$. Then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Since $f(n) = n^{\log_b a}$, again we are in case 2 of the master theorem, and $T(n) = \Theta(n \log n)$.

- $T(n) = 4T(n/2) + n$.

  For this case we have $a = 4, b = 2$, and $f(n) = n$. Then $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$. Since $f(n) = O(n^{2-\epsilon})$, with $\epsilon = 1$, then we can apply case 1 of the master theorem and say that $T(n) = \Theta(n^2)$.

2.2.3 P and NP

We call P the set of problems that can be solved in polynomial time, and NP the set of problems whose solution can be verified in polynomial time. To illustrate a bit more the concept of verification of a solution, imagine that someone gives us a string s and tells us that the string s is a *palindrome*, i.e. it can be read in the same way from left to right as from right to left, for example the word "radar" is a palindrome, but we do not trust our source and we want to verify if in fact s is a palindrome. We can do it in $O(n)$, which is polynomial, and therefore this problem would be in NP. Now, it is clear that P is inside NP, since a problem which solution can be obtained in polynomial time, can be verified in polynomial time, see figure 2.2. However, it is not that clear for the other way around, meaning that a problem whose solution can be verified in polynomial time, it is uncertain that it can be solved in polynomial time. For example, consider the problem of selecting a sub-set of numbers from a given set, in such a way that the sum of the elements in the sub-set is equal to some given number K. If a solution is given to us, we can verify in linear time if that solution is correct, just sum all the numbers and check if the result is equal to K, but finding that solution is not that easy.

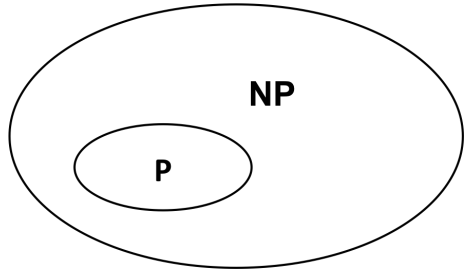


Figure 2.2: P and NP problems

The proof that a problem whose solution can be easily verified (in polynomial time) can or cannot be solved easily (in polynomial time) has not yet found, and it is one of the seven millennium problems.

2.3 Bitwise Operations

Bitwise operations are performed by the processor all the time, because they are the fundamental operations of the computer. Here, we will discuss these operations from the programmer perspective, in other words, to make our programs more efficient.

These types of operations are executed faster than the arithmetic operations, namely: the sum, the multiplication, the division or the modulo, the reason is basically because the arithmetic operations are a set of bitwise operations internally. Therefore whenever we can substitute an arithmetic operation by an bitwise operation we make our program faster in terms of execution time.

There are a variety of bitwise operators, but in this chapter we will tackle the most important ones.

2.3.1 AND (&) operator

AND operator, denoted commonly as as & or **and** in most programming languages, is defined as follows in a bitwise level: $0 \ \& \ 0 = 0$, $0 \ \& \ 1 = 0$, $1 \ \& \ 0 = 0$, $1 \ \& \ 1 = 1$.

The AND operator forces in some sense the two bits to be 1 to get 1 as a result. When we have two numbers formed by several bits each of them, the AND operation is executed in each pair of respective bits. For example, let's take two numbers based on 8 bits: 173 = 10101101 and 85 = 01010101, if we do 173 & 85 the result is 5 = 00000101, as we can see in figure 2.3.

```
  10101101
& 01010101
----------
  00000101
```

Figure 2.3: Example AND operator

An interesting application of this is when we want to know the modulo r of a number n modulo 2^m, because the modulo r can be obtained by doing $n \ \& \ (2^m - 1)$. For instance, $n = 27$ and $m = 2$, $27 \ \& \ 3 = 3$ and $27 \ mod \ 2^2 = 3$. The idea behind this is that $2^m - 1$ is a number formed by only 0's at the left and m 1's at the

right, so when we make an AND operation the last m bits at the right of n remain as they are, and the other ones become 0, and that is exactly what the modulo operation does when it is applied with a power of 2. Remember that the modulo operation, along with the division, is the most computationally expensive operation among the basic arithmetic operations, therefore this trick saves computational time whenever the module operation is executed in a loop and involves a power of 2.

2.3.2 OR (|) operator

OR operator, normally indicated as | or **or** in the majority of the programming languages, is defined in a bitwise level in this fashion: $0 \mid 0 = 0$, $0 \mid 1 = 1$, $1 \mid 0 = 1$, $1 \mid 1 = 1$. The OR operator returns 1 if least one bit is 1. For the case of numbers composed by several bits the OR operation is performed in the same way as the AND operation. For instance, let's take two numbers based on 8 bits: $51 = 00110011$ and $149 = 10010101$, if we perform $51 \mid 149$ the result is $183 = 10110111$, as we can see in figure 2.4.

```
  00110011
| 10010101
----------
  10110111
```

Figure 2.4: Example OR operator

Imagine that we want to sum integers such that all of them are powers of 2 and they are different from each other. Instead of using the sum operation we can perform an OR operation and the outcome will be the same. The summands are powers of 2, therefore they have one bit on (1) and the rest are off (0), since the bit that is activated is different in all the numbers by assumption, the OR operator just turn on the respective bit in the result and this is equivalent to the sum operation.

For instance, let's take the numbers $2 = 00000010$, $8 = 00001000$ and $32 = 01000000$, based on 8 bits. The sum is $42 = 01010010$, which is exactly the same as $00000010 \mid 00001000 \mid 01000000 = 01010010 = 42$.

A common use of the OR operator is to keep track of how many elements have been selected from a total of N possible elements. For instance, assume there are $N = 10$ different basketball teams numbered from 0 to $N - 1$, and each team plays only one game against each other team, we can keep track of which teams have already played against a specific team by using a number B, that initially has a value of zero, if the team plays against team i, then we set the i^{th} bit to 1. This means that if $B = 3$, then the team has played against teams 0 and 1, since the 1^{st} and 2^{nd} bit are 1. This technique is called **bit masking**.

In C/C++ to set the i^{th} bit to 1 we can do

```
B |= (1 << i),
```

and to know if we have played against team i

```
if ((B & (1 << i))) {
  cout << "We have played against team " << i << "\n";
}
```

2.3.3 XOR operator

XOR operator, usually written as $\wedge$ or **xor** in most programming languages, is defined in the following manner in a bitwise level: $0 \wedge 0 = 0$, $0 \wedge 1 = 1$, $1 \wedge 0 = 1$, $1 \wedge 1 = 0$.

The XOR operator indicates if the two bits are different no matter the order. When having numbers shaped by several bits, the operation is executed in the same fashion as the AND or OR operators. Figure 2.5 shows an example of doing $150 \wedge 76$, the result is 218, which in its binary representation has 1's where the bits were different, and 0's where the bits were equal.

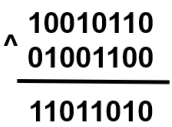


Figure 2.5: Example XOR operator

A common trick where the XOR operator comes handy is when we want to swap the value of two variables, usually we would use a

temporary or auxiliary variable to do this, but with XOR we don't need any extra variable. Suppose we want to swap the values of variables x and y, the trick is the following:

$$x = x \wedge y$$
$$y = y \wedge x$$
$$x = x \wedge y$$

2.3.4 Two's complement

Two's complement is commonly used to represent signed integers. Suppose we have a number of N bits and we add both, the number and its two's complement, then, we get 2^N, which is a $N + 1$ bit number, but if we consider only the first N bits we have the value of zero, which is what we expect when we add a number with its negative.

One way to obtain the two's complement of a number is to switch all its bits (one's complement) and add 1, as illustrated in 2.6.

```
  00101110
  11010001
+ 00000001
----------
  11010010
```

Figure 2.6: Two's complement of 46. The first line is the binary representation of 46, the second line is what we obtain after flipping all the bits. The third line is the binary representation of 1, and after adding line 2 and line 3, we get the two's complement of 46, which is 210.

Another way is to go trough all the bits starting from the less significant bit, and after passing the first 1 start flipping the rest of the bits. For the example in figure 2.6 we would keep unchanged the first two bits, and after that the remaining bits are flipped.

Even though it is uncommon to face a situation where you are asked to write a program that obtains the two's complement of a number, for either a contest or interview, it is still a tool which is

useful in some cases, like coding a *Binary Indexed Tree (BIT)*, that we will cover later in the book.

2.4 Chapter Notes

There are "famous" recursion problems, like the *Eight Queens Problem (A.1), Sudoku Puzzle*, among others. Some of them would be difficult to solve without recursion, or without using other kind of approaches like genetic algorithms, etc. But even when recursion is a powerful technique, we have to be careful when to use it. For the Fibonacci problem mentioned before for example, perhaps recursion here is not the best way to go, unless we use memoization. Even so, a simple array storing all the Fibonacci numbers and a loop cycle would be more than sufficient, and in that way we avoid the problems with memory occasioned by calculating the same Fibonacci number more than once. In few words, we have to choose wisely when to use recursion, as sometimes we do not have a choice, but if we do, then we have to analyze the cost-benefit factor.

In the case of algorithm analysis it is very important to know the basics of it, to have an idea of when an approach will be good or not depending on the input data. Always keep in mind the worst case scenario when you write a code. The book of *Introduction to Algorithms* [1] contains a detailed explanation about this subject.

Bitwise operations are always faster, since they are executed at bit level directly, and there a lot of applications that use them, like communications protocols, and security algorithms. Whenever it is possible, it is usually a good idea to use bitwise operations. Sometimes, they can make the code a little fuzzier because of the symbolic notations, but it is worth to try it.

In appendix A you can find the solutions to problems that exemplify the use of recursion and bitwise operations.

2.5 Exercises

1. Following the same idea of the *8-queen problem* (See appendix A.1), write a program that solves Sudokus. The input consists of a 9×9 matrix containing numbers in the range $[0, 9]$, where 0 means an empty square that needs to be filled. The rules of the Sudoku are simple, on every row and every column has to appear each digit once, and also in every 3×3 sub-matrix.

2. You have N friends, which are numbered from 0 to $N-1$. After your vacations on the Caribbean, you have bought them presents, and some of your friends may receive more than one present, write a program that determines if you have given a gift to a friend or not. The input consists on three numbers N ($2 \leq N \leq 20$), M ($1 \leq M \leq 10^5$), and Q ($1 \leq Q \leq 10^5$), indicating respectively the number of friends you have, the number of presents you have bought for your friends and the number of queries. The next line contains M numbers in the range $[0, N-1]$ indicating to which friend you have given a present. After that Q lines follow, each one with a number k in the range $[0, N-1]$, for each query your program should identify if the k^{th} friend has received a gift or not.

3

Data Structures

"Experience is the name everyone gives to their mistakes."
– Oscar Wilde

Data structures is a fundamental part of computer programming, since they are used to store our data. An array, a matrix, or any multi-dimensional array are examples of a data structure.

In this chapter we will review some of the most used data structures, we will analyze their properties and use cases, because depending on the circumstances some are more apt than others.

Let's analyze a simple array and possible use cases. Imagine that we have an array X of n elements, $X_0, X_1, \ldots, X_{n-1}$. Now, what if we want to extract the 10^{th} element of the array? Well, that is easy, we just go and retrieve the element X_{10}. That simple operation of retrieving certain element runs in $O(1)$ time. Now, suppose we want to find certain element in X, well, in that case we must go trough the whole array and check element by element until we find the one we are looking for. That task runs in $O(n)$ time. Finally, consider the case of removing one element from X. That is not a simple task, since we must remove the desired element and shift all the elements at its right. That task has a $O(n)$ running time.

The time complexity for insertion, extraction, searching, and deletion operations varies depending on the data structure that is being utilized. Some are fast to extract information like vector or

arrays, other are faster to insert or remove data like lists, other like trees are more suitable to find elements. This will be the purpose of this chapter, to analyze the pros and cons of each data structure and how they work.

3.1 Linear Data Structures

In this section we will review the basic data structures. Let's begin by defining what is a data structure. A data structure is basically a tool to storage data in memory with a specific purpose when you are coding. Inherently associated with the data structure are the operations that allows to perform. These operations are normally insert, delete and find one element or a set of elements inside the data structure.

For the code in this section, we will separate the overall Memory Complexity and the Time Complexity for each of the operations over the data structure.

3.1.1 Stack

This data structure is used under the principle *last in first out (LIFO)*, that means the last element inserted will be the first element to be removed. A daily life case to exemplify how a stack works is how to dry a stack of washed dishes. We stack the dishes as we wash them, so the last plate washed will be the first to be dried, and so on until we have no plates anymore.

Stacks are plenty used in Computer Science, they are the basis of recursion, and they are heavily utilized in Graph Theory, as we will see in chapter 7. In programming contests, it is common to use stacks when you are solving problems related to the evaluation of mathematical expressions and parenthesis balance. Code 3.1 shows a custom implementation of a stack with capacity of 100 elements.

Memory Complexity: $O(n)$
***Insert* Time Complexity:** $O(1)$
***Delete* Time Complexity:** $O(1)$
***Find* Time Complexity:** $O(n)$

Input:
List of numbers(it can be any abstract data type) that will be

stored in the stack.

Output:

It depends on the operation and the current state of the stack. The operations performed will be insert, known as push, and delete, known as pop.

Listing 3.1: Custom Implementation of a Stack

```
// Stack simulated with array
// Stack simulated with array
#include <iostream>
#include <stdio.h>
using namespace std;

int s[100];
int l; // index l (last) to manage the stack

void push(int x) {
  s[l++] = x;
}

int pop() {
  int x = s[l--];
  return x;
}

bool isEmpty() {
  return l == 0;
}

void print() {
  for (int i = 0; i < l; i++) {
    printf("%d ", s[i]);
  }
  printf("\n");
}

int main() {
  // Insert numbers from 1 to 5
  printf("First stack state after insert[1-5]:\n");
  for (int i = 1; i <= 5; i++) {
    push(i);
  }

  print(); // See the state of the stack

  // Delete the last 2 elements
  pop();
  pop();

  printf("Stack state after 2 deletes:\n");
  print(); // See the state of the stack

  push(4); // Insert one element
  printf("Stack state after 1 insert (number 4):\n");

  print(); // See the state of the stack

  // Clear the stack
  while (!isEmpty()) {
```

```
53      pop();
54    }
55
56    printf("Stack empty\n");
57    return 0;
58  }
```

A stack also can be simulated with a linked list, but the concept is the same. In section 3.3 we are going to see how to use a stack from the *Standard Template Library (STL)*.

3.1.2 Queue

This data structure is used under the principle of *first in first out (FIFO)*, that means that the first element inserted will be the first element to be removed. A daily life case to illustrate how a queue works is the line formed in a supermarket checkout, the first person who arrives is the first person to be served. Queues are perhaps as common as stacks in Computer Science. In a contest, it is common to implement a queue to do a *Breadth First Search (BFS)* over a graph, as we will see in chapter 7, and it is useful to solve problems where it is asked the state of certain list that is modified over time. Program 3.2 implements a queue using a static array of 100 elements to exemplify its functionality.

Memory Complexity: $O(n)$
***Insert* Time Complexity:** $O(1)$
***Delete* Time Complexity:** $O(1)$
***Search* Time Complexity:** $O(n)$

Input:
List of numbers(it can be any abstract data type) that will be stored in the queue.
Output:
It depends on the operation and the current state of the queue. The operations performed will be insert, known as push, and delete, known as pop.

Listing 3.2: Custom Implementation of a Queue

```
1  // Queue simulated with array
2  #include <iostream>
3  #include <stdio.h>
4
5  using namespace std;
6  int q[100];
7  int f, b; // index f(front) and b(back) to manage the queue
```

```
8
9   void push(int x) { q[f++] = x; }
10
11  int pop() {
12    int x = q[b++];
13    return x;
14  }
15
16  bool isEmpty() { return (b == f); }
17
18  void print() {
19    for (int i = b; i < f; i++) {
20      printf("%d ", q[i]);
21    }
22    printf("\n");
23  }
24
25  int main() {
26    // Insert numbers from 1 to 5
27    printf("First queue state after insert [1-5]:\n");
28    for (int i = 1; i <= 5; i++) {
29      push(i);
30    }
31
32    print(); // See the state of the queue
33
34    // Delete the first 2 elements
35    pop();
36    pop();
37    printf("Queue state after 2 deletes:\n");
38    print(); // See the state of the queue
39
40    push(4); // Insert one element
41    printf("Queue state after 1 insert (number 4):\n");
42    print(); // See the state of the stack
43
44    // Clear the stack
45    while (!isEmpty()) {
46      pop();
47    }
48
49    printf("Queue empty\n");
50    return 0;
51  }
```

As a stack, a queue can also be simulated with a linked list and there would not be any change conceptually speaking. The STL library has a queue implemented too, and we will see it in section 3.3.

3.1.3 Linked List

A linked list can be thought of as an array, but the difference is that it allows us to insert and delete in constant time $O(1)$. There are various types of linked lists, some of them are: simple linked list, double linked list, circular linked list, double circular linked

list, among others.

A linked list is composed of nodes, these nodes store data, just like arrays, but they also store the memory address of nodes related to this node. The relationship between nodes can be the next node, previous node, child node, parent node, sibling node, etc.

In a linked list, it is necessary to know where is the beginning and the end of the list, known as head and tail of the list. Typically, the last node points to `NULL`.

In a simple linked list, doubly linked list, circular linked list and doubly circular linked list, the operations has the same time and memory complexity.

Memory Complexity: $O(n)$
***Insert* Time Complexity:** $O(1)$
***Delete* Time Complexity:** $O(1)$
***Search* Time Complexity:** $O(n)$

Simple Linked List

A *simple linked list* is a linked list where a node stores only the information about the next node address, and the main information of the node. Figure 3.1 shows a conceptualization of a linked list.

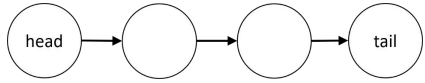


Figure 3.1: Visualization of a linked list

The source code in 3.3 define the class `Node`, which contains an integer value to store the information associated to that node, and a pointer to the next node in the list, except for the last node, which points to `NULL`. There are two more pointers that we need to store, `head`, which points to the first node of the list, and `tail` which points to the last node of the list. The `main` function add elements to the list and then remove them from the list.

Input:
List of numbers (it can be any data abstract type) that will be stored in the simple linked list.

Output:

It depends on the operation and the current state of the simple linked list. The operations performed will be insert, delete and search.

Listing 3.3: Simple Linked List

```
#include <cstdio>
using namespace std;

class Node {
 public:
  int val;
  Node *next;

  Node(int val, Node *nextNode) {
    this->val = val;
    this->next = nextNode;
  }
};

Node *head = NULL;
Node *tail = NULL;

void pushBack(int);
void popFront();
void printList();

int main() {
  // Insert three elements in the list
  pushBack(2);
  pushBack(3);
  pushBack(5);
  printList();

  // Remove the first element
  popFront();
  printList();

  // Remove the other two elements
  popFront();
  popFront();
  printList();

  return 0;
}
```

The `pushBack` function add a node at the end of the list. The new node becomes the tail, and the `next` pointer of the node that was previously the tail is modified and points to the new node.

```
void pushBack(int val) {
  Node *newNode = new Node(val, NULL);

  if (tail == NULL) {
    head = newNode;
  } else {
    tail->next = newNode;
```

```
  }

  tail = newNode;
}
```

The `popFront` function removes the first element of the list, if there is any. The node associated to the `next` pointer of the head becomes the head, and the node that was previously the head is removed.

```
void popFront() {
  if (head != NULL) {
    Node *nextNode = head->next;

    delete (head);

    head = nextNode;
    if (head == NULL) {
      tail = NULL;
    }
  }
}
```

The function `printList` as its name says print the whole list by iterating trough all its elements printing them one by one. If the list is empty prints the message *"– empty list –"*.

```
void printList() {
  Node *curNode = head;

  if (curNode == NULL) {
    printf("-- empty list --\n");
    return;
  }

  while (curNode != NULL) {
    printf("%d", curNode->val);
    curNode = curNode->next;

    if (curNode != NULL) {
      printf(" -> ");
    }
  }

  printf("\n");
}
```

Doubly Linked List

A doubly linked list is a linked list where a node stores the memory address about the next node and the previous one. The link to the next node of the last node points to `NULL`, and the link to the previous node of the first node points to `NULL` as well. See figure

3.2.

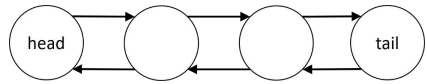


Figure 3.2: Visualization of a doubly linked list

The program in 3.4 defines a class `Node` similar to the one used in the *Simple Linked List*, but it adds a pointer to the previous node. The functions `pushBack` and `popFront` are similar to the ones in the *Simple Linked List* implementation, but include some changes that modify the `next` and `prev` links of the nodes involved in the insertion or deletion process.

Input:

List of numbers(it can be anything) that will be stored in the doubly linked list.

Output:

It depends on the operation, the current state of the doubly linked list. The operations performed will be insert, delete and search.

Listing 3.4: Doubly Linked List

```
#include <cstdio>
using namespace std;

class Node {
 public:
  int val;
  Node *next;
  Node *prev;

  Node(int val, Node *nextNode, Node *prevNode) {
    this->val = val;
    this->next = nextNode;
    this->prev = prevNode;
  }
};

Node *head = NULL;
Node *tail = NULL;

void pushBack(int);
void pushFront(int);
void popFront();
void popBack();
void printList();

int main() {
  // Add two elements at the end of the list
  pushBack(5);
```

```
29    pushBack(7);
30    printList();
31
32    // Removes the last element
33    popBack();
34    printList();
35
36    // Removes the last element and the list is empty
37    popBack();
38    printList();
39
40    // Add five elements at the top of the list
41    pushFront(6);
42    pushFront(8);
43    pushFront(3);
44    pushFront(1);
45    pushFront(4);
46    printList();
47
48    / Removes the last element and the two first elements popBack();
49    popFront();
50    popFront();
51    printList();
52
53    return 0;
54  }
```

Let t be the current tail. When we add an element to the back of the list, the new node becomes the tail with its `prev` link pointing to t, and the `next` link of t pointing to the new tail.

```
void pushBack(int val) {
  Node *newNode = new Node(val, NULL, NULL);

  if (tail == NULL) {
    head = newNode;
  } else {
    newNode->prev = tail;
    tail->next = newNode;
  }

  tail = newNode;
}
```

The `pushFront` function inserts an element at the beginning of the list. The new node becomes the head with its `next` link pointing to the node that was previously the head.

```
void pushFront(int val) {
  Node *newNode = new Node(val, NULL, NULL);

  if (head == NULL) {
    tail = newNode;
  } else {
    newNode->next = head;
    head->prev = newNode;
  }
```

```
  head = newNode;
}
```

The `popFront` function removes the first element of the list. The node associated to the `next` link of the head becomes the new head, and the previous head is removed from the list.

```
void popFront() {
  if (head != NULL) {
    Node *nextNode = head->next;
    if (nextNode != NULL) {
      nextNode->prev = NULL;
    }

    delete (head);

    head = nextNode;
    if (head == NULL) {
      tail = NULL;
  }
}
```

Similar to the `popFront` function, the `popBack` function removes the last element of the list. The tail is removed and node associated to its `prev` link becomes the new tail.

```
void popBack() {
  if (tail != NULL) {
    Node *prevNode = tail->prev;
    if (prevNode != NULL) {
      prevNode->next = NULL;
    }

    delete (tail);

    tail = prevNode;
    if (tail == NULL) {
      head = NULL;
    }
  }
}
```

Circular Linked List

A circular linked list is almost exactly the same as a simple linked list, but there is a slight difference, and is that instead of having the last node pointing to `NULL`, it points to the first node, and hence the name. See figure 3.3.

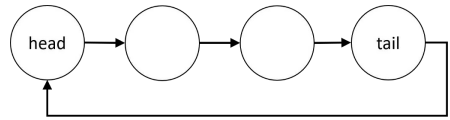


Figure 3.3: Visualization of a circular linked list

The code for the *Circular Linked List* is almost the same that the one in 3.3, with just one difference, every time that the list is modified, because of an insertion or a deletion, we have to make sure that the tail is linked to the head. Basically we must add the following statement.

```
if (tail != NULL) {
  tail->next = head;
}
```

Doubly circular Linked List

A doubly circular linked list is practically the same as a doubly linked list, but with a tiny difference, the last node has a link to the first node, and conversely the first node has a link that points to the last one. See figure 3.4.

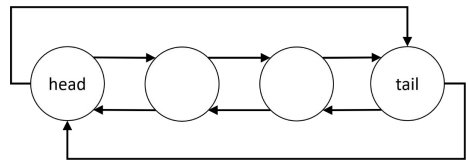


Figure 3.4: Visualization of a circular linked list

The implementation is very similar to the program 3.4, but again, we must be careful when the list is modified by an insertion or a deletion by ensuring that the first and last element of the list are linked to each other. We can do that by adding the following sentences:

```
if (head != NULL) {
  head->prev = tail;
  tail->next = head;
}
```

3.2 Trees

So far, we have reviewed linear data structures, but there is more, in this section we will discuss tree-shaped data structures. Let's start by defining a tree from the graph theory perspective. A tree is a simple graph (graph without multiple edges) with no cycles. This definition might be not too useful to picture a tree as a data structure, perhaps it is better to think of it as a multilevel structure to store information. One of the main applications of Trees is in file systems, when we create a file or a folder, somewhere internally we are creating a child from a specific node, which represents the folder. Other applications are related to keeping elements ordered and apply queries over it, one example of this is an index in some `SQL` package.

Before continuing, it is necessary to understand the following terminology:

Root: Top node of the tree.
Child: Node pointed by other node in a previous level.
Parent: Node that points to a node at the next level.
Siblings: Nodes that comes from same parent.
Leaf: Node with no children.
Internal Node: Any node that is neither the leaf nor the root.
Ancestor: Node that is before another node coming from root.
Descendant: Node that is after another node coming from root.
Path: Sequence of nodes connected.
Height: The height of a node is the maximum length of a path from that node to a leaf node. The height of a tree is the height of the root node.

3.2.1 Tree Traversal

The most common type of tree is the binary tree, on which each node has at most two children, and every node except for the root has exactly one parent. Figure 3.5 shows an example of a binary tree.

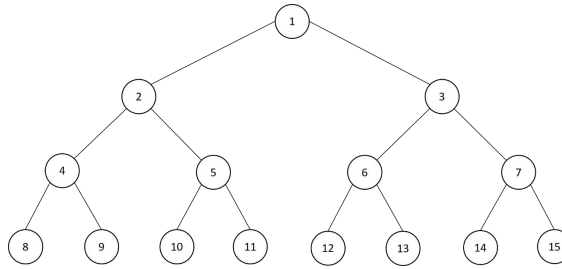


Figure 3.5: Example of a full binary tree

A common way to represent a node in a program is by using a class or structure that stores the value of the node, as well as the left and right pointers to the node's children. See code in 3.5

Listing 3.5: A simple Node class

```
class Node {
  public:
  Node *left;
  Node *right;
  int value;
};
```

As its name indicates, tree traversal is a way of going trough all the nodes of a tree, there can be multiple ways of doing that, as long as all the nodes are visited once. Here, we will cover the three most common ways to traverse trees: pre-order, in-order and post-order.

Pre-order Traversal

1. Visit the current node
2. Traverse left sub-tree
3. Traverse right sub-tree

For the tree in figure 3.5 the order in which the nodes are visited using pre-order traversal starting from the root is: $1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15$.

Listing 3.6: Pre-order Traversal

```
void preOrder(Node *node) {
  if (node == NULL) {
    return;
  }
```

```
5
6     printf("%d, ", node->value);
7     preOrder(node->left);
8     preOrder(node->right);
9   }
```

In-order Traversal

1. Traverse left sub-tree
2. Visit the current node
3. Traverse right sub-tree

Using in-order traversal to traverse the tree in 3.5, the nodes are visited as follows: $8, 4, 9, 2, 10, 5, 11, 1, 12, 6, 13, 3, 14, 7, 15$.

Listing 3.7: In-order Traversal

```
1   void inOrder(Node *node) {
2     if (node == NULL) {
3       return;
4     }
5
6     inOrder(node->left);
7     printf("%d, ", node->value);
8     inOrder(node->right);
9   }
```

Post-order Traversal

1. Traverse left sub-tree
2. Traverse right sub-tree
3. Visit the current node

For tree in 3.5, using post-order traversal the nodes are visited in the following order: $8, 9, 4, 10, 11, 5, 2, 12, 13, 6, 14, 15, 7, 3, 1$.

Listing 3.8: Post-order Traversal

```
1   void postOrder(Node *node) {
2     if (node == NULL) {
3       return;
4     }
5
6     postOrder(node->left);
7     postOrder(node->right);
8     printf("%d, ", node->value);
9   }
10  }
```

3.2.2 Heap

A heap is binary tree where the value associated to node k is larger or equal than the value associated to its children. Figure 3.6 represents a valid heap with integer values. As we can notice, the greatest element is always at the root.

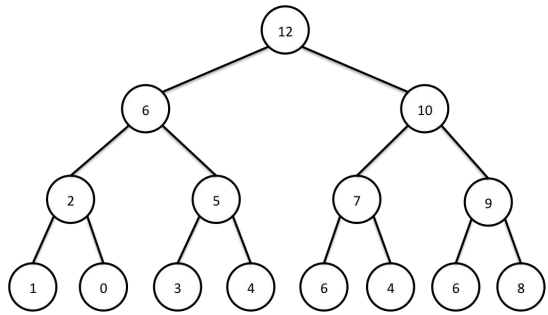


Figure 3.6: Example of a heap

One common situation is to transform an array of numbers into a heap. For that, we can use another array to represent a heap, for element in position k, its left child will be located in position $2k$, and its right child in position $2k + 1$, being the root the element at index 1. The array representation of the heap in figure 3.6 would be the following.

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]	X[9]	X[10]	X[11]	X[12]	X[13]	X[14]	X[15]
12	6	10	2	5	7	9	1	0	3	4	6	4	6	8

The method `downHeap` in 3.9 has the purpose of placing in the correct position the k^{th} element. The idea is to continue moving the given element down trough the heap until its children are smaller or equal or until we reach a leaf node. In any other case the element is swapped with the largest of its two children. The method receives the array representation of the heap, the number of elements in the heap, and the index of the element to be placed in the correct position.

Listing 3.9: Place element in a heap

```
1 void downHeap(int *H, int n, int k) {
2   int maxChild, temp;
```

```
  int leftChild = 2 * k;
  int rightChild = 2 * k + 1;

  while (leftChild <= n) {
    maxChild = leftChild;

    if (rightChild <= n && H[rightChild] > H[maxChild]) {
      maxChild = rightChild;
    }

    if (H[k] < H[maxChild]) {
      temp = H[k];
      H[k] = H[maxChild];
      H[maxChild] = temp;
    } else {
      break;
    }

    k = maxChild;
    leftChild = 2 * k;
    rightChild = 2 * k + 1;
  }
}
```

To create a valid heap we just need to call the `downHeap` method for every node, starting from node n (the last leaf node) down to node 1 (the root). This bottom-up strategy is necessary, because in order to place node k, all its descendants must be already in the right position.

Heaps are particularly useful when there are multiple queries asking for the largest or smallest value (min heaps) on a dataset. Since the largest element is always at the root we can answer those queries in $O(1)$, meanwhile update queries, insertions and deletions take $O(\log n)$. So if you are asked to write a program that finds the maximum or minimum value of some given data, specially if that operation is repeated multiple times, then a heap is perhaps the right data structure for you. It is not uncommon to face these kind of questions on an interview or contest. Keep that in mind and practice, so whenever you hear the phrase "find the maximum value in ...", your brain automatically switches to heaps.

Later, in this chapter we will see how to implement a heap easily using the STL library so you don't have to write the `downHeap` function every time you need to use a heap, but it is still important to understand its internal functionality, so we recommend, for educational purposes, to implement both, your own heap from scratch and then using the STL library, but in real-life situations try to

use libraries if possible, there is no need to re-invent the wheel.

3.2.3 Binary Search Tree (BST)

Binary Search Trees, or BSTs have existed for many years, their main goal is to be able to process search queries fast, and in the average case they can perform well, but still in the worst scenario they behave in the same way as a linked lists.

Each node in a BST has at most two children nodes, the left node stores a smaller value than the parent node, and the right node stores a larger value than the parent node. In this way we know, for a certain node, that all elements at its left are smaller, and all elements at its right are larger. In the case for elements with the same value, they can be placed either in the left sub-tree or in the right sub-tree, but not in both, at the moment of implementation it must be decided to which side they will go and maintain that behavior all the time.

Let's put into practice what we just said, suppose we want to store the following numbers in a BST: 5, 2, 9, 1, 3, 7, 8, 4, 6 (in this order). The root of tree will be 5, then, number 2 will be the left child of 5, since it is smaller than 5. Number 9 becomes the right child of 5, since is larger than 5, next comes number 1, which will become the left child of 2, due to is smaller than 5 and smaller than 2. If we continue doing the same process we end up with the tree in figure 3.7.

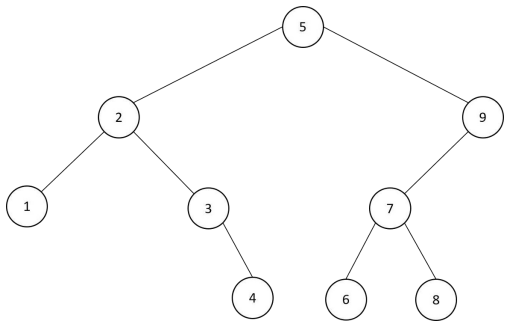


Figure 3.7: Example of a BST

By looking at figure 3.7, we can understand the benefits of a

tree structure to store our data. For example, if we want to search for number 6, we only need three comparisons. On the contrary, in a linked list, 6 would be the last element and we would need to go trough all elements before finding it.

The main problem with BST is that its performance depends on how elements are being inserted, for example, if we store numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, we would have 1 as the root, 2 as its right child, then, 3 would be the right child of 2, and so on, ending up with a linked list. Then, why should we store our data in a tree? Well, as we seen in figure 3.7, using trees have its advantages, but we need the tree to be *balanced*. We say that a tree is balanced when for any node the absolute difference between the height of the left sub-tree and the height of the right sub-tree is not greater than 1. An AVL tree is an example of a balanced tree and we will go deeper into this in 3.2.4, but first let's take a look into an implementation of the BST.

First, we need a class to store the information of a Node, and for this example, a Node is composed of a link to its left child, a link to its right child, a link to its parent node, and a numeric value. When we create a new node it is important to set the links to its children to NULL, since we will use that to identify when to stop when going through the tree.

Listing 3.10: BST

```
class Node {
 public:
  Node *left;
  Node *right;
  Node *parent;
  int value;

  Node(Node *left = NULL,
       Node *right = NULL,
       Node *parent = NULL,
       int value = 0) {
    this->left = left;
    this->right = right;
    this->parent = parent;
    this->value = value;
  }
};
```

The `insertNode` function in 3.11 takes in an integer value, which will be the value of a new node in the BST, it also receives a pointer to the parent of the current node, and a pointer to the cur-

rent node. Then, starting from the root, it continues moving down recursively in the BST depending on the value to be inserted, if it is less or equal we move left, otherwise we move right, this keeps going until we reach a `NULL`, finally, we append the new node to the last element traversed in the tree.

Listing 3.11: BST: Insertion

```
void insertNode(int val, Node *parent, Node *curNode) {
  if (curNode == NULL) {
    curNode = new Node(NULL, NULL, parent, val);
    connectNode(curNode);
    return;
  }

  if (val <= curNode->value) {
    insertNode(val, curNode, curNode->left);
  } else {
    insertNode(val, curNode, curNode->right);
  }
}
```

The `connectNode` function receives a node and connects its parent to the node given based on the node's value. If the parent is `NULL` it means that the node is the root. The implementation of this function is shown in 3.12.

Listing 3.12: BST: Node Connection

```
void connectNode(Node *curNode) {
  Node *parent = curNode->parent;
  if (parent == NULL) {
    root = curNode;
  } else if (parent->value >= curNode->value) {
    parent->left = curNode;
  } else {
    parent->right = curNode;
  }
}
```

So far, we have seen how to insert elements into a BST, but what happens if we need to delete an element? Here is when things turn a little bit more interesting, since we need to do some tricks with our pointers. When an element is removed from the tree, the rule is to replace it either with the largest value of its left sub-tree, or with the smallest value of its right sub-tree. Let's see an example for the tree in figure 3.7, suppose we want to remove the node with value 5, which is the root, well, according to the rule we have to replace it with the largest value of the left sub-tree (4) or with the smallest value of the right sub-tree (6). After replacing it with the node with value 4 we end up with the tree in figure 3.8.

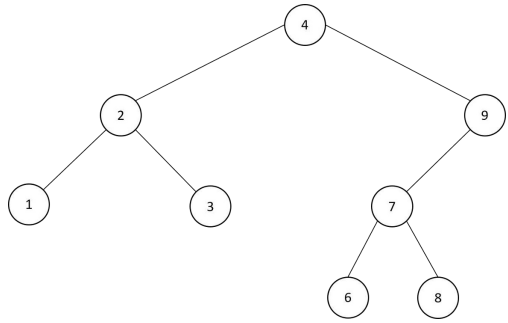


Figure 3.8: BST after removal

For this example that would be enough, but what happens if the node with value 4 has a left sub-tree, in this case, the left sub-tree becomes the right sub-tree of the parent node (2), this will ensure that the properties of the BST are maintained. See figure 3.9.

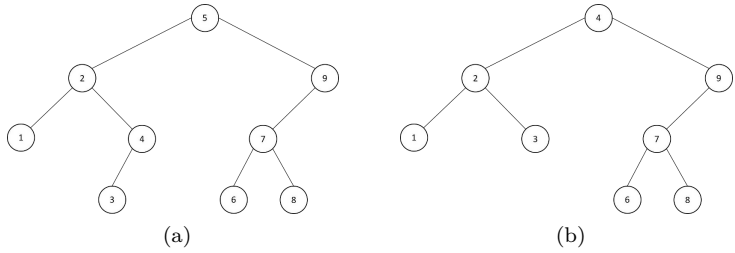


Figure 3.9: a) A BST with 9 nodes. b) The same BST after removing node 5 and replacing it with the node 4, notice that the left child of node 4 becomes the right child of node 2.

Another case we need to consider is to delete a node and replace it with one if its direct children. See figure 3.10, node a is removed and node b takes its place maintaining its left child and acquiring node c as its right child.

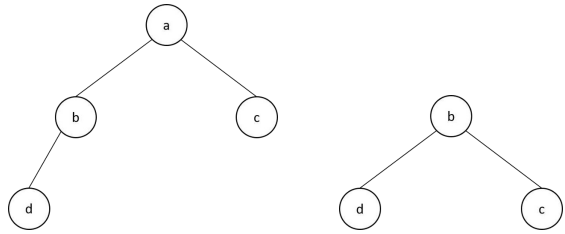


Figure 3.10: a) BST with four nodes. b) BST after node a is removed and replaced by node b.

To make things easier for the implementation of the deletion process, we will use three auxiliary functions that will also be useful when we deal with balanced binary trees. The first of those functions is `replace`, which receives two nodes A and B, and copy all edges from node A into node B.

Listing 3.13: BST: Replace nodes

```
1  void replaceNode(Node *oldNode, Node *newNode) {
2    newNode->left = oldNode->left;
3    newNode->right = oldNode->right;
4    newNode->parent = oldNode->parent;
5
6    if (newNode->left != NULL) {
7      newNode->left->parent = newNode;
8    }
9    if (newNode->right != NULL) {
10     newNode->right->parent = newNode;
11   }
12 }
```

The other two auxiliary functions are `getLargest` and `getSmallest`, the first finds the node with the largest value of the left sub-tree of a node, which we call `pivot`, and the second finds the node with the smallest value of the right sub-tree of node `pivot`. Both functions receives a pointer to some node in the BST and the node `pivot`. `getLargest` will continue moving right and it returns a pointer to the last node, while `getSmallest` moves left until the last node is reached and it returns a pointer to that node. Once a node is found in any of the two functions, we disconnect it from the tree as explained in figures 3.9 and 3.10.

Listing 3.14: BST: Largest element from a node

```
1  Node *getLargest(Node *curNode, Node *pivot) {
2    if (curNode->right != NULL) {
```

```
    Node *largest = getLargest(curNode->right, pivot);
    return largest;
  } else if (curNode->parent != pivot) {
    curNode->parent->right = curNode->left;
  } else {
    curNode->parent->left = curNode->left;
  }

  if (curNode->left != NULL) {
    curNode->left->parent = curNode->parent;
  }

  return curNode;
}
```

Listing 3.15: BST: Smallest element from a node

```
Node *getSmallest(Node *curNode, Node *pivot) {
  if (curNode->left != NULL) {
    Node *smallest = getSmallest(curNode->left, pivot);
    return smallest;
  } else if (curNode->parent != pivot) {
    curNode->parent->left = curNode->right;
  } else {
    curNode->parent->right = curNode->right;
  }

  if (curNode->right != NULL) {
    curNode->right->parent = curNode->parent;
  }

  return curNode;
}
```

The recursive implementation will become handy when we deal with balanced trees, as we will see later in this chapter. Now we have everything we need to remove a node from a BST. The function `removeNode` in 3.16 receives an integer specifying the value to be removed from the tree, and a pointer to a node in the BST, the function will move recursively until it finds the value we are looking for, or until it reaches a `NULL` node. Here is a summary of what the functions does.

1. Check if we have reached a `NULL` node, if it does, stop the search and print a message indicating that the value is not in the tree, otherwise move to step 2.

2. If the current node has a value equal to the value we are looking for, then, proceed with the removal of the node in step 3, otherwise move to step 7.

3. If the current node has a left child, get the node with the largest value in its left sub-tree using the `getLargest` func-

tion and replace the current node with the node found and go to step 6. If the current node does not have a left child then move to step 4.

4. Check if the current node has a right child, and if it does, get the node with the smallest value in its right sub-tree with the `getSmallest` function, and replace the current node with it, then move to step 6. If there is no right child, go to step 5.

5. If the current node does not have any children, we will proceed to remove that node, but first, we will make its parent to point to `NULL`, which means we are disconnecting the current node from the tree. Go to step 6.

6. Delete the current node from memory.

7. Continue looking up in the tree, if the value we are looking for is less or equal than the value of the current node, move to the left sub-tree, otherwise move to the right sub-tree.

Some extra considerations have to be made, like what happens if the root is removed, in that case, we need to set a new node as the root, or avoid the case when the parameter received is a `NULL` pointer.

Listing 3.16: BST: Node removal

```
void removeNode(int val, Node *curNode) {
  if (curNode == NULL) {
    printf("Value %d not found\n", val);
    return;
  } else if (curNode->value == val) {
    Node *endNode = NULL;
    if (curNode->left != NULL) {
      endNode = getLargest(curNode->left, curNode);
    } else if (curNode->right != NULL) {
      endNode = getSmallest(curNode->right, curNode);
    } else {
      disconnectNode(curNode);
    }

    if (endNode != NULL) {
      // Replace the node to be removed
      replaceNode(curNode, endNode);
      connectNode(endNode);
    }

    // Delete the node and return
    delete curNode;
    return;
  } else if (curNode->value >= val) {
    removeNode(val, curNode->left);
  } else {
```

```
    removeNode(val, curNode->right);
  }
}
```

The `disconnectNode` function is similar to the code in 3.12, and its purpose is to disconnect a node from the tree by making its parent to point to `NULL`. See code 3.17.

Listing 3.17: BST: Node disconnection

```
void disconnectNode(Node *curNode) {
  Node *parent = curNode->parent;
  if (parent == NULL) {
    root = NULL;
  } else if (parent->value >= curNode->value) {
    parent->left = NULL;
  } else {
    parent->right = NULL;
  }
}
```

All these functions will be useful for the implementation of balanced binary trees, since the process is almost the same, but an extra step is made, the *balancing* step, that ensures that our search queries run in $O(\log n)$ time.

To make sure everything works fine try to run the following code:

Listing 3.18: BST Testing

```
int main() {
  int x[] = {3, 8, 9, 2, 1, 5, 6, 4, 7};
  int z[] = {3, 0, 4, 8, 6, 7, 9, 2, 2, 5, 1};

  // Insert elements into the tree
  printf("Inserting elements into the tree.....\n");
  for (int i = 0; i < 9; i++) {
    insertNode(x[i], NULL, root);
    printBST(root);
    printf("\n");
  }

  // Remove elements from the tree
  printf("\nRemoving elements from the tree.....\n");
  for (int i = 0; i < 11; i++) {
    removeNode(z[i], root);
    printBST(root);
    printf("\n");
  }

  return 0;
}
```

The function `printBST` prints the value of all nodes in pre-order

traversal. This means that we will print the value of a node, then, move to the left child, and then, move to the right child.

Listing 3.19: BST Print

```
void printBST(Node *node) {
  if (node == NULL) {
    return;
  }

  printf("%d ", node->value);
  printBST(node->left);
  printBST(node->right);
}
```

The output of the program should be:

```
Inserting elements into the tree.....
3
3 8
3 8 9
3 2 8 9
3 2 1 8 9
3 2 1 8 5 9
3 2 1 8 5 6 9
3 2 1 8 5 4 6 9
3 2 1 8 5 4 6 7 9

Removing elements from the tree.....
2 1 8 5 4 6 7 9
Value 0 not found
2 1 8 5 4 6 7 9
2 1 8 5 6 7 9
2 1 7 5 6 9
2 1 7 5 9
2 1 5 9
2 1 5
1 5
Value 2 not found
1 5
1
```

In real-life situations, it is possible that you will not need to implement a BST, even in programming contests, since there are libraries for data structures that internally store data in a tree, but it is not uncommon to see interview questions about binary search trees or balanced trees, since in most cases, the interviewer wants to know what is your thinking process, and asking about all the details involved in tree operations like insertions and deletions is a good way to measure your problem solving skills. As with the rest of the algorithms in this book, you do not need to memorize any code, you just need to understand how trees work, what is the time complexity for different operations, and what are their advantages and disadvantages, if you know that, you will be able to implement

a tree if the situation requires it.

3.2.4 AVL Tree

The main problem with BSTs without balancing is that in the worst case scenario they behave exactly as a linked list, and finding an element is a $O(n)$ operation. On the other hand, balanced trees maintain a running time of $O(\log n)$.

A tree is balanced if for any node the absolute difference of the height of its left sub-tree and its right sub-tree is less or equal than 1. So every time that a node is inserted or removed from the tree, a balancing process needs to take place to ensure that this property is maintained.

The AVL tree is a type of balanced tree, it gets its name from its inventors, Georgy Adelson-Velsky and Evgenii Landis [2]. It is also a binary search tree, with every node being greater or equal than all nodes in its left sub-tree, and smaller than any other node in its right sub-tree. For this kind of tree, every node needs to store its height, since the way to identify if a node is unbalanced is by checking the heights of its left and right sub-trees. For that reason, we add `height` and `balanceFactor` as attributes of the `Node` class. See code 3.20.

Listing 3.20: AVL: `Node` class

```
class Node {
 public:
  Node *left;
  Node *right;
  Node *parent;
  int value;
  int height;
  int balanceFactor;

  Node(Node *left = NULL, Node *right = NULL, Node *parent = NULL,
       int value = 0) {
    this->left = left;
    this->right = right;
    this->parent = parent;
    this->value = value;
    this->height = 0;
    this->balanceFactor = 0;
  }
};
```

The `balanceFactor` attribute will store the difference between the height of the left sub-tree and the height of the right sub-tree.

Since initially when we insert a node it has no children, both `height` and `balanceFactor` are set to 0.

The insertion process is almost the same as in the BST, but with the difference that we need to update the height of every node affected by the insertion, because of that, we will modify our `insertNode` function from our BST implementation as shown in code 3.21.

Listing 3.21: AVL: Insertion

```
void insertNode(int val, Node *parent, Node *curNode) {
  if (curNode == NULL) {
    curNode = new Node(NULL, NULL, parent, val);
    connectNode(curNode);
    return;
  }

  if (val <= curNode->value) {
    insertNode(val, curNode, curNode->left);
  } else {
    insertNode(val, curNode, curNode->right);
  }

  updateHeight(curNode); // update height and balanceFactor
  balanceNode(curNode);
}
```

The function `updateHeight` is an auxiliary function that computes the height and `balanceFactor` of the given node. See code 3.22.

Listing 3.22: AVL: Update Height

```
void updateHeight(Node *curNode) {
  int leftHeight = curNode->left == NULL ? 0 : curNode->left->height + 1;
  int rightHeight = curNode->right == NULL ? 0 : curNode->right->height + 1;
  curNode->height = max(leftHeight, rightHeight);
  curNode->balanceFactor = leftHeight - rightHeight;
}
```

By taking advantage from recursion, after calling the `insertNode` function, we can update the `height` and `balanceFactor` of the current Node, assuming that all nodes bellow are already balanced. This allows us to do the balancing process as we insert new nodes into the tree, instead of doing both things separately. Before jumping into the code of how a node is balanced, let's explain how it works.

Let X be a node with X_l and X_r as its left and right sub-trees respectively. We say that node X is unbalanced if its `balanceFac-`

tor is greater or equal than 2, or less or equal than -2. In order to re-balance a node we need to do something called **rotations**. There are four possible scenarios, which are described bellow.

1. $X.balanceFactor \geq 2$ and $X_l.balanceFactor > 0$, also called the *right-right rotation*. According to figure 3.11, to re-balance the tree we do the following:

   (a) X becomes the right child of X_l.

   (b) Z_2 becomes the left child of X.

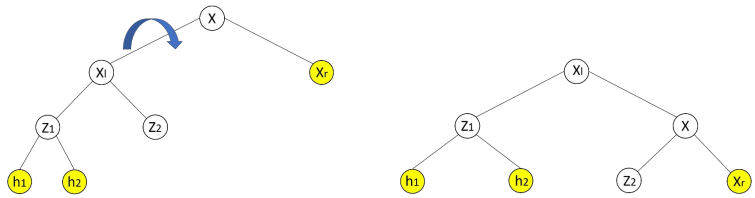


Figure 3.11: a) Node X is unbalanced with a $balanceFactor \geq 2$, and X_l with a $balanceFactor > 0$. b) Result after rotation. Node X is balanced and yellow nodes remain unchanged during the process.

2. $X.balanceFactor \leq -2$ and $X_r.balanceFactor < 0$, also called the *left-left rotation*. See figure 3.12. The balancing process consists of two steps:

   (a) X becomes the left child of X_r.

   (b) Z_1 becomes the right child of node X.

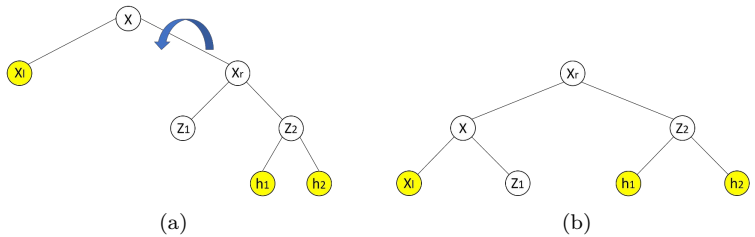


Figure 3.12: a) Node X is unbalanced with a $balanceFactor \leq -2$, and X_r with a $balanceFactor < 0$. b) After the rotation node X is balanced and yellow nodes remain unchanged.

3. Left - Right Rotation. $X.balanceFactor \geq 2$ and $X_l.balanceFactor \leq 0$. In this case, we need to do two rotations. The first is a left rotation over node X_l, which will cause that we end up in scenario 1 (right - right rotation). The second rotation is a right rotation over node X to re-balance the tree. Figure 3.13 shows how nodes are affected by the two rotations and how the tree gets balanced at the end of the process.

   For the first rotation we do:

   (a) Z_2 becomes the left child of X.

   (b) X_l becomes the left child of Z_2.

   (c) h_1 becomes the right child of X_l.

   and for the second rotation:

   (a) X becomes the right child of Z_2.

   (b) h_2 becomes the left child of X.

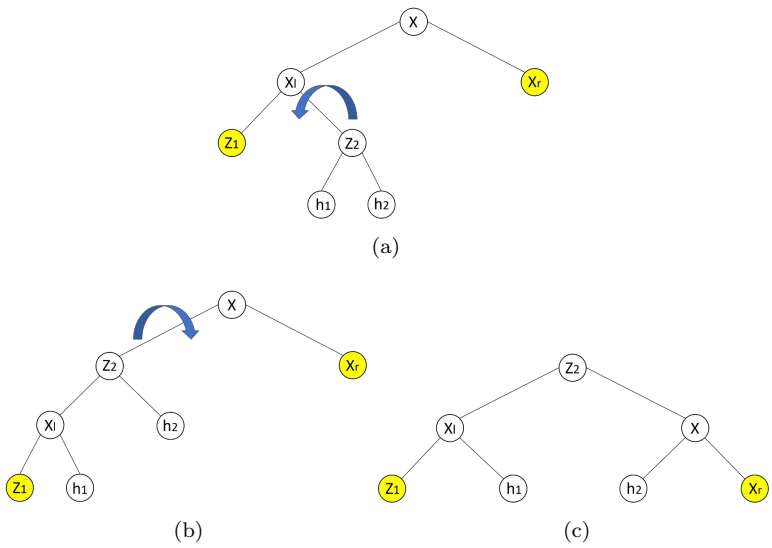


Figure 3.13: a) Node X is unbalanced with a $balanceFactor \geq 2$, and X_l has a $balanceFactor < 0$. b) After a left rotation over node X_l we end up with a right - right rotation. c) The entire tree is balanced after a right rotation over node X. Yellow nodes remain unchanged.

4. Right - Left Rotation. $X.balanceFactor \leq -2$ and $X_l.balanceFactor \geq 0$. As in the previous case, we need to do two rotations, first a right rotation over node X_r, which will cause that the tree takes the form of scenario 2 (left - left rotation), which involves a left rotation over node X, ending up with a balanced tree. The steps for the rotations are listed below and also are displayed in figure 3.14.

   For first rotation the steps are the following:

   (a) Z_1 becomes the right child of X.

   (b) X_r becomes the right child of Z_1.

   (c) h_2 becomes the left child of X_r.

   and for the rotation over node X we have:

   (a) X becomes the left child of Z_1.

   (b) h_1 becomes the right child of X.

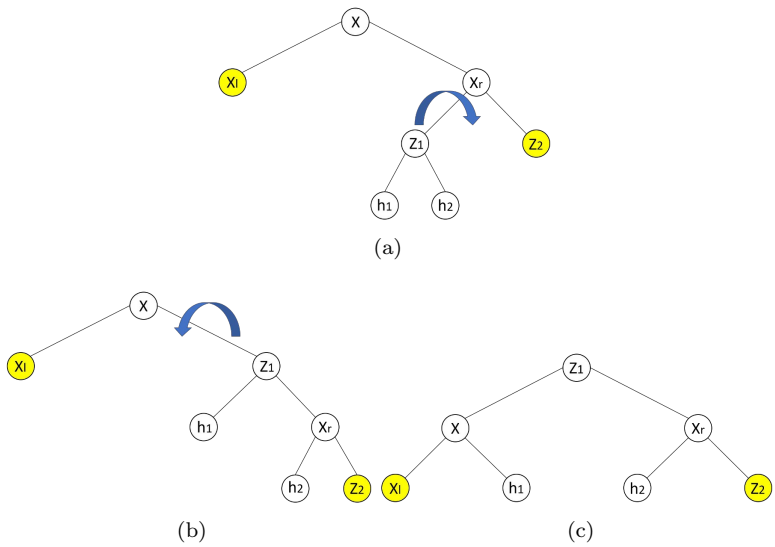


Figure 3.14: a) Node X is unbalanced with a $balanceFactor \leq -2$, and X_r has a $balanceFactor > 0$. b) After the first rotation the tree is in a left - left rotation scenario c) The tree gets balanced after a left rotation over node X. Yellow nodes remain unchanged.

Now that we have a better picture of the possible scenarios that we can face in the balancing process, let's analyze the code. The `balanceNode` function is implemented in code 3.23, and what it does is basically handle each one of the four scenarios that we just described.

Listing 3.23: AVL: Balance Node

```
void balanceNode(Node *curNode) {
  if (curNode->balanceFactor >= 2) {
    if (curNode->left->balanceFactor > 0) {
      // right - right rotation
      rightRotation(curNode);
    } else {
      // left - right rotation
      leftRotation(curNode->left);
      rightRotation(curNode);
    }
  } else if (curNode->balanceFactor <= -2) {
    if (curNode->right->balanceFactor > 0) {
      // right - left rotation
      rightRotation(curNode->right);
      leftRotation(curNode);
    } else {
      // left - left rotation
      leftRotation(curNode);
    }
  }
}
```

Depending on which scenario we are in, we do one or two rotations, to simplify the code we created two functions: `leftRotation` and `rightRotation`, both of them set the rules for re-balancing a node. See the codes in 3.24 and 3.25.

Listing 3.24: AVL: Left Rotation

```
void leftRotation(Node *node) {
  Node *rightNode = node->right;
  Node *temp = rightNode->left;

  rightNode->left = node;
  rightNode->parent = node->parent;
  node->right = temp;
  node->parent = rightNode;

  if (temp != NULL) {
    temp->parent = node;
  }

  linkParent(rightNode);
  updateHeight(node);
  updateHeight(rightNode);
}
```

Notice that `updateHeight(node)` should be called before `up-`

`dateHeight(rightNode)`, since `node` is now a child of `rightNode`, and to update the height of a node, the heights of its children must be already updated.

Listing 3.25: AVL: Right Rotation

```
void rightRotation(Node *node) {
  Node *leftNode = node->left;
  Node *temp = leftNode->right;

  leftNode->right = node;
  leftNode->parent = node->parent;
  node->left = temp;
  node->parent = leftNode;

  if (temp != NULL) {
    temp->parent = node;
  }

  linkParent(leftNode);
  updateHeight(node);
  updateHeight(leftNode);
}
```

If these two functions are confusing, use the figures that illustrates each one of the four scenarios as guidance. Sometimes we find easier to understand rotations by looking at examples.

Up to this point, we have covered the balancing process when elements are inserted in the tree, the only thing missing is balancing the tree when an element is removed. For this, we will take advantage of the recursive strategy implemented for the BST, but this time we need to verify for each node in the recursive path if it is unbalanced, and if it is, then proceed to call `balanceNode`. The function `removeNode` in 3.26 is similar to the one used in the BST, the only difference is that we need to call `updateHeight` and `balanceNode` for every node visited in the path. Here, we can see the benefits of having a recursive implementation, since that allows us to balance the tree at the same time a node is removed.

Listing 3.26: AVL: Node Removal

```
void removeNode(int val, Node *curNode) {
  if (curNode == NULL) {
    printf("Value %d not found\n", val);
    return;
  } else if (curNode->value == val) {
    Node *endNode = NULL;
    if (curNode->left != NULL) {
      endNode = getLargest(curNode->left, curNode);
    } else if (curNode->right != NULL) {
      endNode = getSmallest(curNode->right, curNode);
    } else {
```

```
      disconnectNode(curNode);
    }

    if (endNode != NULL) {
      // Replace the noDe to be removed, update balanceFactor and re-balance
      replaceNode(curNode, endNode);
      connectNode(endNode);
      updateHeight(endNode);
      balanceNode(endNode);
    }

    // Delete the node and return
    delete curNode;
    return;
  } else if (curNode->value >= val) {
    removeNode(val, curNode->left);
  } else {
    removeNode(val, curNode->right);
  }

  updateHeight(curNode);
  balanceNode(curNode);
}
```

Another change in our BST implementation is that we also need to call the functions `updateHeight` and `balanceNode` for each node visited in the functions `getSmallest` and `getLargest`. In few words, each node involved in the removal process needs to be balanced, starting from the last element up to the root. Codes 3.27 and 3.28 show the code for functions `getSmallest` and `getLargest` respectively.

Listing 3.27: AVL: Get Smallest Node

```
Node *getSmallest(Node *curNode, Node *pivot) {
  if (curNode->left != NULL) {
    Node *smallest = getSmallest(curNode->left, pivot);
    updateHeight(curNode);
    balanceNode(curNode);
    return smallest;
  } else if (curNode->parent != pivot) {
    curNode->parent->left = curNode->right;
  } else {
    curNode->parent->right = curNode->right;
  }

  if (curNode->right != NULL) {
    curNode->right->parent = curNode->parent;
  }

  return curNode;
}
```

Listing 3.28: AVL: Get Largest Node

```
Node *getLargest(Node *curNode, Node *pivot) {
```

```
  if (curNode->right != NULL) {
    Node *largest = getLargest(curNode->right, pivot);
    updateHeight(curNode);
    balanceNode(curNode);
    return largest;
  } else if (curNode->parent != pivot) {
    curNode->parent->right = curNode->left;
  } else {
    curNode->parent->left = curNode->left;
  }

  if (curNode->left != NULL) {
    curNode->left->parent = curNode->parent;
  }

  return curNode;
}
```

If we modify the function `printBST` in 3.19 and instead of just printing the node value, we print the node value and the balance factor closed by parentheses. The output of code 3.18 is:

```
Inserting elements into the tree.....
3 (0)
3 (-1) 8 (0)
8 (0) 3 (0) 9 (0)
8 (1) 3 (1) 2 (0) 9 (0)
8 (1) 2 (0) 1 (0) 3 (0) 9 (0)
3 (0) 2 (1) 1 (0) 8 (0) 5 (0) 9 (0)
3 (-1) 2 (1) 1 (0) 8 (1) 5 (-1) 6 (0) 9 (0)
3 (-1) 2 (1) 1 (0) 8 (1) 5 (0) 4 (0) 6 (0) 9 (0)
3 (-1) 2 (1) 1 (0) 6 (0) 5 (1) 4 (0) 8 (0) 7 (0) 9 (0)

Removing elements from the tree.....
6 (1) 2 (-1) 1 (0) 5 (1) 4 (0) 8 (0) 7 (0) 9 (0)
Value 0 not found
6 (1) 2 (-1) 1 (0) 5 (1) 4 (0) 8 (0) 7 (0) 9 (0)
6 (0) 2 (0) 1 (0) 5 (0) 8 (0) 7 (0) 9 (0)
6 (0) 2 (0) 1 (0) 5 (0) 7 (-1) 9 (0)
5 (0) 2 (1) 1 (0) 7 (-1) 9 (0)
5 (1) 2 (1) 1 (0) 9 (0)
2 (0) 1 (0) 5 (0)
1 (-1) 5 (0)
Value 2 not found
1 (-1) 5 (0)
1 (0)
```

At first sight, we notice that in some cases the root changes after inserting or removing a node, that tells us that rotations are doing something, which is good, but by looking at the balance factors of each node we see that none of them is greater than 2, or less than -2, which means that the tree remains balanced while nodes are being inserted or removed.

There are different kinds of balanced trees, all of them with the

propose of providing a fast way for searching elements. Their implementation, as you have seen, is not trivial and there are multiple cases that need to be considered. The goal of describing the functionality and code of AVL trees is not only to show the advantages of having a balanced tree structure, but also to give the reader a glimpse of the challenges that can appear at the moment of design and implementation, most of them related to pointers handling.

3.2.5 Segment Tree

Segment trees are a powerful tool that allow us to do fast searches over an interval. For example, Given an array X, find the minimum value over interval $X[a \dots b]$, or find the sum of all elements on $X[a \dots b]$.

A segment tree is a binary tree where leaf nodes store the original data, and internal nodes store information about segments of that data. The construction of the tree takes $O(n \log n)$, while searching takes $O(\log n)$. An update of an element runs in $O(\log n)$ without the need of re-building the whole tree, since only the segments where the updated element interacts will be modified.

For the implementation proposed here, we will use a full binary tree as the segment tree, a full binary tree is a binary tree with all its internal nodes having exactly two children. For this, first, we need to compute the number of levels necessary to store the data, then assign the original data into the leaf nodes, adding any necessary nodes to the tree to make it a full binary tree.

We will use a vector to represent the segment tree, with the root being at index 0. A node at index k will have its left child at index $2k + 1$, and its right child at index $2k + 2$. In this way, if the tree has l levels, the original data will be stored in the vector starting at index $2^{l-1} - 1$, and the total number of nodes in the tree is $2^l - 1$.

Let's see an example to describe the functionality of a segment tree. First, we define x as a vector containing the original data. For this example, we will use $x = [5, 3, 8, 1, 5, 7, 4, 0, 9, 2, 6]$. We will write a program that receives multiple queries asking for the maximum value in some interval, no update queries will be given. That said, the first thing to do is create the segment tree starting from the leaf nodes as explained before, and then moving up,

calculating the maximum value from the left and right sub-trees for all internal nodes, and keeping track of the segment that each node covers. Figure 3.15 shows the created segment tree for this example, there you can see the original data stored at leaf nodes and how the value of internal nodes is the maximum between the left child and right child. The yellow nodes represent the nodes that are added to fill the binary tree, here is important to mention that the value of those nodes should not affect the result in this case, since we are interested in finding the maximum value of an interval, we chose a value of zero, so they do not influence on the result. In the case that we were looking for the minimum value on an interval we would have to set a *"large"* value to those nodes, so they are never chosen over a node with the original data.

Based on the segment tree in 3.15, suppose we want to know what is the maximum value in $X[0 \dots 9]$. Starting from the root, we obtain at index 1 that the maximum value in $X[0 \dots 7]$ is 8, we do not need to move below that node. Meanwhile the maximum in $X[8 \dots 9]$ is 9, which is stored at index 11 of the segment tree. At the end we visited a total of five nodes to know the answer, which is better than doing a linear search over the original vector. We hope that this clarifies the advantages of using a segment tree.

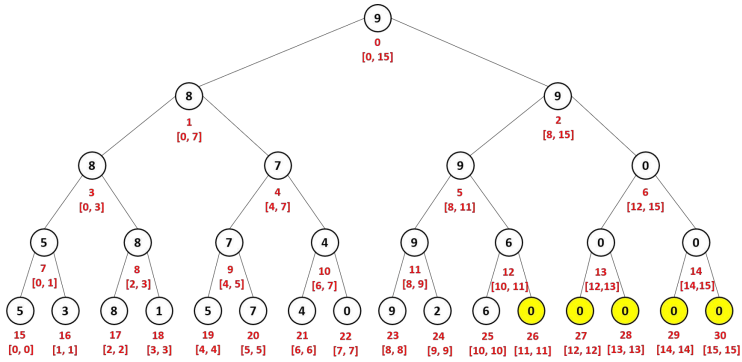


Figure 3.15: Example of a segment tree, the number inside the node represents the maximum value in the segment, in red we can see the index in the vector representation and the interval which that segment covers.

Now that we have a better picture of how segment trees work, let's write a small program. First, we need a class to store the

node's information, following the example above, we will store the maximum value over a segment, and the start and end point of the segment. See 3.29.

Listing 3.29: Segment Tree: Class Node

```
class Node {
  public:
  int maxValue;
  int a;
  int b;

  Node(int a = 0, int b = 0, int maxValue = 0) {
    this->a = a;
    this->b = b;
    this->maxValue = maxValue;
  }
};
```

The function `buildSegmentTree` in 3.30 receives a vector of integers larger or equal than zero that represents the original data, and creates a segment tree following the method described before and store it in the global variable `segmentTree`.

Listing 3.30: Creation of a Segment Tree

```
void buildSegmentTree(vector<int> x) {
  // Find the size of the tree
  int n = 1;
  while (n < x.size()) {
    n *= 2;
  }

  // Allocate memory for the segment tree
  segmentTree.resize(2 * n - 1);

  // Initialize leaf nodes
  for (int i = 0, j = n - 1; j < 2 * n - 1; i++, j++) {
    if (i < x.size()) {
      segmentTree[j] = Node(i, i, x[i]);
    } else {
      segmentTree[j] = Node(i, i, 0);
    }
  }

  // Compute internal nodes
  for (int i = 2 * n - 2; i >= 0; i -= 2) {
    int a = segmentTree[i - 1].a;
    int b = segmentTree[i].b;
    int maxValue = max(segmentTree[i - 1].maxValue, segmentTree[i].maxValue);
    segmentTree[i / 2 - 1] = Node(a, b, maxValue);
  }
}
```

The creation of the segment tree can be summarized in three steps:

1. Find the number of levels in the tree.

2. Assign the original data to leaf nodes and add the necessary nodes to make it a full binary tree, without affecting the result.

3. Compute internal nodes starting from leaf nodes up to the root.

Now that the segment tree is created, we can start with the queries asking for the maximum value of an interval. The function `findMaxValue` in 3.31 receives the start index a, and end index b of an interval, as well as the index k of a node in the segment tree, and it returns the maximum value in $x[a \dots b]$ starting from node k. The algorithm is the following:

1. Check if the query interval intersects with the node's interval, if it does not, then return 0 to avoid affecting the result.

2. Update the query interval, e.a. If query interval is [3,9] and node's interval is [0,7], update the query interval to [3,7].

3. If the query interval is equal to the node's interval, then return the maximum value of that segment, otherwise return the greatest value between the maximum of the left sub-tree and the maximum of the right sub-tree.

Listing 3.31: Search in a Segment Tree

```
int findMaxValue(int a, int b, int k) {
  // If given interval is outside node's interval then return
  if(a > segmentTree[k].b || b < segmentTree[k].a) {
    return 0;
  }

  // Compute new interval
  a = max(a, segmentTree[k].a);
  b = min(b, segmentTree[k].b);

  // If the node's interval covers the given interval, return node's maxValue
  if (a == segmentTree[k].a && b == segmentTree[k].b) {
    return segmentTree[k].maxValue;
  }

  // Otherwise return the maximum between the left sub-tree and the right
       sub-tree
  int maxLeft = findMaxValue(a, b, 2 * k + 1);
  int maxRight = findMaxValue(a, b, 2 * k + 2);
  return max(maxLeft, maxRight);
}
```

We will use the function `getMaxValue` to answer the queries, this function receives an interval and pass on the same interval to the `findMaxValue` function, setting $k = 0$ to start searching from the root. See code in 3.32.

Listing 3.32: Queries for a Segment Tree

```
int getMaxValue(int a, int b) {
  return findMaxValue(a, b, 0);
}
```

The following code tests the functions we wrote using the same example from figure 3.15. First, we create the segment tree by passing a vector x with our data to the `buildSegmentTree` function, then, we print the maximum value of some query intervals by calling the `getMaxValue` function.

```
typedef pair<int, int> query;

int main() {
  vector<int> x = {5, 3, 8, 1, 5, 7, 4, 0, 9, 2, 6};
  vector<query> queries = {query(3, 9),
    query(2, 3),
    query(4, 7),
    query(9, 10)};

  buildSegmentTree(x);

  for(int i = 0; i < queries.size(); i++) {
    printf("max value in [%d, %d] = %d\n",
      queries[i].first,
      queries[i].second,
      getMaxValue(queries[i].first, queries[i].second));
  }

  return 0;
}
```

The output for the program is the following:

```
max value in [3, 9] = 9
max value in [2, 3] = 8
max value in [4, 7] = 7
max value in [9, 10] = 6
```

3.2.6 Binary Indexed Tree (BIT)

Suppose we have an array composed of numbers $a_0, a_2, ..., a_{n-1}$ and we want to know what is the sum from the index i to index j, $0 \leq i \leq j < n$. The naive approach is go through the interval and sum element by element. If this query is performed once, then

the time cost is $O(n)$, but if we have m queries, the complexity time is $O(mn)$. This complexity can be reduced by using another array that stores the cumulative sum from the index 0 to the index i, this might be done in linear time, then, to resolve the query sum$\{a_i, a_{i+1}, \ldots, a_{j-1}, a_j\}$, with $i < j$, we can just simply calculate the difference between the cumulative sum of index j and the cumulative sum of index $i - 1$. The last operation can be computed in constant time $O(1)$. So far, everything seems good enough, but the last algorithm is considering a static array, in other words, the array is never modified, what happens if we want to update the array and make queries at the same time? Here is when BIT shows up, BIT allows us to solve the problem of the sum of an interval when this is not static.

The idea behind BIT consists in storing the accumulates from index 0 to index i using a binary representation of the indexes. A BIT is also known as Fenwick tree, since it was proposed by Peter Fenwick in 1994 [3].

Before to proceed with the explanation, let's see a bitwise trick which consists in isolating the last non-zero bit of a number (keep the last non-zero bit and put 0's in the other bits) by using the fact that computer represent a negative number as two's complement. See image 3.16.

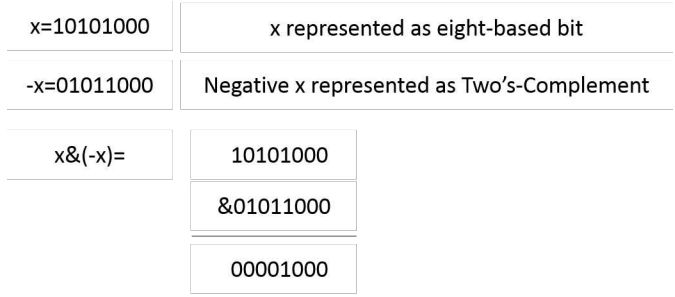


Figure 3.16: Bitwise operation to get the least significant bit

Let C_i be the accumulate sum from index 0 to some index i, and let T be the array that represents the BIT. It is important to mention that indexes of T starts from 1 and not from 0, meaning that T is an array with indexes from 1 to n. For example, suppose we want to obtain the value of $C_{10_{10}}$(10_{10} means the number 10

written in base 10), by using the array T this is accomplished with the following sum:

$$C_{10_{10}} = T_{1011_2} + T_{1010_2} + T_{1000_2}$$

The first thing to do is add 1 to the given index, since the indexes in T starts from 1. Then, we do a binary representation of that number, in this case 11_{10}, which is 1011_2, an then, using the bitwise trick explained before, we convert the last 1 from that number into a 0 to obtain the new index. We continue doing this until the index is zero.

$$1011_2 \rightarrow 1010_2 \rightarrow 1000_2 \rightarrow 0000_2$$

The procedure explained before was to compute a query. But if we want to make an update, we just need to do the process backwards from the given index i (not starting from 0) until we update the complete array T.

Figure 3.17 shows the tree structure of a BIT for 3-bit indexes. There we can see how for a node with index i, the index of its parent node is obtained by replacing with 0 the last 1 of the binary representation of i.

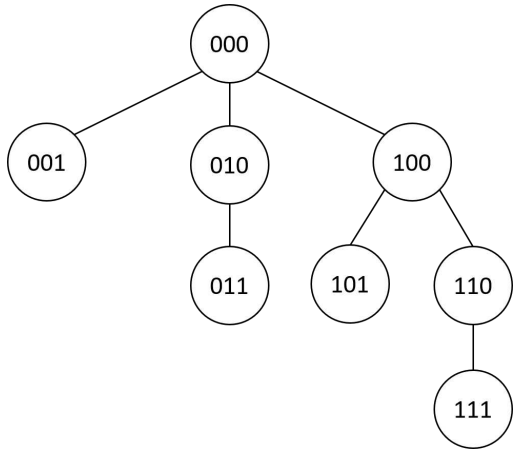


Figure 3.17: BIT for 3-bit indexes

The code in 3.33 shows a basic implementation of a BIT. It contains three functions. The function `updateBIT` add a value of

v to node $i + 1$ and to all its ancestors. The function `queryBIT` receives an integer i and returns the cumulative sum of all numbers in the original array from index 0 to index i. And the function `createBIT`, which creates the BIT from the original array.

The time complexity of the `queryBIT` and the `updateBIT` functions is $O(\log n)$. On the other hand, the time complexity of the `createBIT` is $O(n \log n)$, since we have to update the BIT for each element in the array.

Time Complexity:
creation: $O(n \log n)$
query: $O(\log n)$
update: $O(\log n)$
Input:
N: Number of elements in the array.
X: Array with integer numbers.
Output:
Queries using a BIT to obtain the sum $X[0] + \cdots + X[i]$, for some i.

Listing 3.33: BIT

```
#include <cstdio>
#define N 10
using namespace std;

int X[N] = {1, 4, 2, 4, 3, 0, 7, 5, 1, 6};
int T[N + 1];

void updateBIT(int, int);
int queryBIT(int);
void createBIT();

int main() {
  createBIT();

  // Print the cumulative sum from X[0] to X[6]
  printf("%d\n", queryBIT(6));

  // Print the cumulative sum from X[0] to X[9]
  printf("%d\n", queryBIT(9));

  // Add 2 to X[5] and print the sum from X[0] to X[9]
  updateBIT(5, 2);
  printf("%d\n", queryBIT(9));
  return 0;
}

void updateBIT(int i, int v) {
  i++;
  while (i <= N) {
    T[i] += v;
```

```
    i += (i & -i);
  }
}

int queryBIT(int i) {
  int res = 0;
  i++;
  while (i > 0) {
    res += T[i];
    i -= (i & -i);
  }
  return res;
}

void createBIT() {
  for (int i = 0; i < N; i++) {
    updateBIT(i, X[i]);
  }
}
```

Below is the output for the previous program. We can see how the cumulative sum incremented by 2 once we added 2 to the 5^{th} element of the array.

```
21
33
35
```

The main advantages of using a BIT are:

1. We can update without the need of expensive re-calculations. Updates run in $O(\log n)$.
2. Its memory complexity is $O(n)$.
3. Is easy to implement, it does not require a great amount of lines of code.

3.2.7 Trie

First described in 1959 by René de la Briandais [4]. A Trie, also called *Prefix Tree*, is a search tree commonly used to find a word in a dictionary. In a Trie every node represents a word or a prefix, and all its descendant nodes share the same prefix. The root node represents an empty string, its children represent words or prefixes of length 1, the children of these represent words or prefixes of length 2, and so on. e.g. Consider the following dictionary: { work, worker, worship, fire, fired, fly }. The corresponding Trie is showed in figure 3.18.

For figure 3.18, in case we are looking for the word "worker", we start the search from the root, and then move to the node with the 'w', that will leave out all those words that do not start with that letter. This will speed up the search. Then, we continue moving to lower levels in the tree until we get to the letter 'r', then we move to the letter 'k', discarding the word "worship". The search continues until we get to the last 'r' in the word 'worker'.

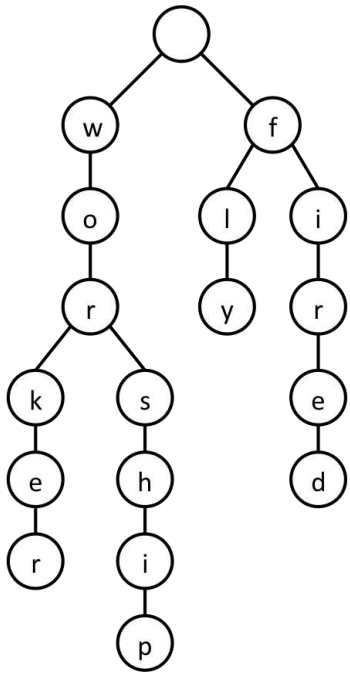


Figure 3.18: Example of a Trie

The time complexity to find a word in a Trie is $O(m)$, where m is the length of the word. Is important to notice that there is no need to store the entire word or prefix in each node, since it can be obtained as the Trie is traversed.

The main advantage of using a Trie is that as the search continues moving from one level to another, other words are being discarded, narrowing this way the search.

When coding a Trie we can store in each node any informa-

tion that we consider useful, for example, we can place a boolean variable that indicates if that node represents a word from the dictionary or a prefix, we can also store a counter that indicates the number of words found. The properties can be as many as we want, but one thing that it must contain is a reference to the rest of the letters. The class showed in 3.34 is an example of a node structure. Notice that the array `ref` is initialized with -1, indicating that there is no connection to another letter, or that initially the node has no children. As we continue adding words to the dictionary the array will contain references to other nodes.

Listing 3.34: Trie Class

```
1  class TrieNode {
2   public:
3    bool isWord;
4    vector<int> ref;
5
6    TrieNode(bool isWord = false) {
7      this->isWord = isWord;
8      ref = vector<int>(26, -1);
9    }
10 };
```

Suppose we store every node in a vector called `Trie`, and the array `ref` indicates the positions in that vector of the children nodes. The code in 3.35 inserts a word into the Trie. The function `addWord` receives the index of the current node, the word we are adding, and the position of the current letter. Then, it checks if the prefix $word[0 \dots pos]$ already exists (line 7), if it does not, we create a reference by adding a new node in `Trie` (lines 8-10). After that, we must check if `pos` is the last letter, if that is the case, then that node is a word. Finally, we move to the next letter and repeat the process.

Listing 3.35: Insert Word in Trie

```
1  void addWord(int nodeId, string word, int pos) {
2    if (pos == word.length()) {
3      return;
4    }
5
6    int k = Trie[nodeId].ref[word[pos] - 'A'];
7    if (k == -1) {
8      k = Trie.size();
9      Trie[nodeId].ref[word[pos] - 'A'] = k;
10     Trie.push_back(TrieNode());
11   }
12
13   Trie[k].isWord = (pos == word.length() - 1) ? true : false;
14   addWord(k, word, pos + 1);
15 }
```

To add the string `word` to the Trie, we only need to call `addWord(0, word, 0)`. It is important to mention that we need to add an initial `TrieNode` object in vector `Trie`, that way `Trie[0]` will represent the root.

3.3 Standard Template Library (STL) C++

STL is one of the most powerful tools that the C++ language has. Inside this library are implemented multiple algorithms and data structures that are very handy for competitive programming. Let's take a look at some of the most common data structures and their respective examples.

3.3.1 Unordered Set

This structure allows us to create an unordered set of unique elements, that means no matter the order just the uniqueness of the elements. Internally this structure is implemented as a hash table, which is a table that associates keys with values in constant time $O(1)$. It's important to say that this structure is available from the standard `C++11`. Suppose we want to count the numbers of different words in some text. We are going to use an unordered set in order to solve that problem. As the set doesn't let us store repeated elements, it will be sufficient to read the word and store it into the the set and at the end the set will keep only distinct elements, so we can get the size of the set and this will be the answer. Let's look at the code.

Time Complexity: $O(nl)$
l: The length of the longest word.
Input:
n: Number of words in the text.
str: A string representing each word of the text.
Output:
Number of different words in the text.

Listing 3.36: Number of different words

```
1 #include <iostream>
2 #include <string.h>
```

```cpp
#include <unordered_set>

using namespace std;

unordered_set<string> u_set;

int main() {
  int n;
  string str;

  // We read the number of words and the words
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> str;
    // We insert all the words we read because the repeated
    // words won't be taken into account
    u_set.insert(str);
  }

  cout << "The number of different words is: " << endl;
  cout << u_set.size() << endl;
  cout << "The words different are: " << endl;

  /* auto is part of C++ 11 and makes or life easier,
     auto substitutes this expression: set<string>::iterator it,
     (*it) means we want the value from iterator it, in our case
     the string.
   */
  for (auto it = u_set.begin(); it != u_set.end(); ++it) {
    cout << (*it) << endl;
  }

  return 0;
}
```

3.3.2 Ordered Set

This structure is almost the same as the unordered set, the only difference is that here the order matters, and actually that can be an advantage in some cases. Internally an ordered set is implemented as a binary search tree, so that guarantees $O(log)$ for the searches, insertions and eliminations.

Let's think in the following problem, suppose we have a list of discrete 2D-points and over the list we make operations like deletes, insertions and queries, the insertions will be simple, they give us a 2D point and we have to add it to our list, if the point is already there we ignore it otherwise we add it, then for the eliminations they give us a point and we need to remove it if exists otherwise do nothing. The queries will give us two points that defines a rectangle, these points represent the bottom-left corner and the top-right corner respectively, and the goal is to print the

number of points contained on that rectangle.

This problem is not trivial and could be an easy-intermediate problem in a ICPC Latin America Regional Contest, so let's start thinking what do we need to solve it. First of all we need a structure to keep the 2D-points list updated and also the structure should allow us perform the queries easy and fast. Thereby an ordered set fits perfectly to solve the problem. To do the insertions and eliminations we only need to use the methods already implemented in a set structure. Then to perform the queries we are going to use the fact that the set stores the elements in a ordered way and also we can use two handy methods that are already implemented in the STL, `lower_bound` and `upper_bound`. Both methods returns an iterator, `lower_bound` returns an iterator where the value is equal or greater than the element sent in the argument, while `upper_bound` returns an iterator where the value is strictly greater than the element sent in the argument. These methods will help us to narrow the search, since we only need to go trough over the points between the two pointers and check if they are inside the rectangle.

Time Complexity: $O(n \log n)$
Input:
n: Number of operations to be performed.
operation: Insert, delete or query operation.
Insertion case
x1, y1: Integer coordinates of the point that will be inserted.
Elimination case
x1 y1: Integer coordinates of the point that will be removed.
Query case
x1, y1, x2, y2: Integer coordinates of the points that defines the rectangle according to the problem.
Output:
In query case, print the number of points into the defined rectangle.

Listing 3.37: Number of 2D-points into a given rectangle

```
#include <iostream>
#include <set>
#include <algorithm>
#include <cmath>
using namespace std;
typedef pair<int, int> point;

```

```
set<point> s;

int main() {
  int x1, y1, x2, y2, ans, n, operation;
  set<point>::iterator p1, p2;

  // We read the number of operations we are going to perform
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> operation;
    switch (operation) {
      case 1: // Insertion case
        cin >> x1 >> y1;
        // make_pair method create a pair based on two parameters
        s.insert(make_pair(x1, y1));
        break;
      case 2: // Elimination case
        cin >> x1 >> y1;
        // We have to make sure that the point exists,
        if (s.find(make_pair(x1, y1)) != s.end()) {
          s.erase(make_pair(x1, y1));
        }
        break;
      case 3: // Query case
        cin >> x1 >> y1 >> x2 >> y2;
        ans = 0;
        // We get the iterator where the value is greater or equal than the
        // point sent
        p1 = s.lower_bound(make_pair(x1, y1));
        // We get the iterator where the value is greater than the point sent
        p2 = s.upper_bound(make_pair(x2, y2));
        /* All the points between these iterators can be inside the rectangle,
           but we have to make sure, because we took the x coordinate as first
           parameter to sort, and even when the current point x-coordinate is
           inside the bounds of the rectangle, it can occur that its
           y-coordinate is out of the range
        */
        for (set<point>::iterator it = p1; it != p2; ++it) {
          if ((*it).first >= x1 && (*it).first <= x2
          && (*it).second >= y1 &&(*it).second <= y2)
            ans++;
        }
        // Print out the numbers of points inside the rectangle
        cout << ans << endl;
        break;
    }
  }
  return 0;
}
```

3.3.3 Unordered Map

An unordered map is a handy structure implemented inside the STL library and is basically a hash table, that means we can associate a key with a value, where the key and value can be any abstract data type, and of course the uniqueness of the key is guaranteed. There are many problems that can be solved using this

structure, for instance, to find which is the most used word in some text, we can simply make a map with a `string` as key representing a word in the text, and a `int` as value representing the number of occurrences of the corresponding word. A more complex problem, suppose we want to know the number of 2D-points repeated in a input, for this problem we can use a map with a 2D-point object as key and a `int` as value. The code in 3.38 contains the class `Point` to be used as a key, as you can see we need to override the `==` operator, since the unordered map needs a way to distinguish one point from another, in this case both coordinates must be the same to consider two points as equal.

Listing 3.38: Class Point for Unordered Map

```
class Point {
  public:
  int x;
  int y;

  Point(int x = 0, int y = 0) {
    this->x = x;
    this->y = y;
  }

  bool operator == (const Point &otherPoint) const {
    return this->x == otherPoint.x && this->y == otherPoint.y;
  }
};
```

We need one more thing in order to use `Point` objects as keys, and that is a hash function. Hash functions assign an element to a bucket in memory, is understandable that the desired behavior is to assign each element in a different bucket to guarantee $O(1)$ all the time, but that is not always possible and collisions happen, and there are methods that are outside the scope of this book that handle those collisions. Back to our problem, to define our hash function, let's assume that all point's coordinates will be on the range [0, 99]. Then the hash function in 3.39 returns values $0 \cdots 99$ for all the points with $y = 0$, $100 \cdots 199$ for points with $y = 1$, and so on. This way we will assign a different value to each possible point.

Listing 3.39: Hash Function for Unordered Map

```
struct PointHasher {
  // Hash function for points with x in [0, 99] and y in [0, 99]
  size_t operator () (const Point &key) const {
    size_t hash = key.y * 100 + key.x;
    return hash;
  }
```

```
};
```

The program in 3.40 creates a unordered map that will store `Point` objects as keys, and the number of occurrences of a point as value. the code prints each point along with the number of times it appeared.

Listing 3.40: Unordered Map Example

```
int main() {
  unordered_map<Point, int, PointHasher> pointMap;
  unordered_map<Point, int, PointHasher>::iterator it;
  vector<Point> points = {
    Point(0, 0),
    Point(1, 2),
    Point(2, 1),
    Point(1, 1),
    Point(1, 2),
    Point(1, 1)};

  for (int i = 0; i < points.size(); i++) {
    pointMap[points[i]]++;
  }

  for (it = pointMap.begin(); it != pointMap.end(); it++) {
    printf("(%d %d): %d\n", (*it).first.x, (*it).first.y, (*it).second);
  }

  return 0;
}
```

The output can vary, since the elements are not stored in any particular order, here is the output we obtained by running the previous code.

```
(2 1): 1
(0 0): 1
(1 1): 2
(1 2): 2
```

The time complexity of a map depends on its hash function, already built-in hash functions works well, but in case you need to implement your own function, keep in mind that the goal of using maps and sets is to have a running time as close as possible to $O(1)$. That is their main advantage that separates them from other data structures.

3.3.4 Ordered Map

It's an upgrade of a ordered set, because is implemented internally as a binary search tree, what ensures a time complexity of $O(log)$

for insertions and eliminations, and we can link keys with values, the order is based on the keys.

If we use a primitive data type as key, for example an integer, the smallest key will be the first one on the map. Similarly if we use strings as keys, these will be sorted in increasing order accordingly to the ASCII values of their characters, but what happens if we want to define our own custom keys? Let's use the previous example of having a map with 2D-point as key and their occurrences as value. For an unordered-map we needed to override the `==` operator and also define a hash function, but for the case of ordered maps or just maps, we need a way to say if an object is greater or not than other object of the same class, otherwise they can't be sorted. This can be done by overriding the `<` operator. See code in 3.41.

Listing 3.41: Class Point for Ordered Map

```
class Point {
  public:
  int x;
  int y;

  Point(int x = 0, int y = 0) {
    this->x = x;
    this->y = y;
  }

  bool operator < (const Point &otherPoint) const {
    if (this->x == otherPoint.x) {
      return this->y < otherPoint.y;
    } else {
      return this->x < otherPoint.x;
    }
  }
};
```

When we override the `<` operator we set the rules for comparing two objects of the same class in order to determine which one is greater. For this example, when comparing two 2D-points, they will be sorted first by their x-coordinate, and if there is a tie, then they will be sorted by their y-coordinate.

From 3.41 we can notice that there is no need to override the `==` operator as we did for unordered maps, and that is not all, also we don't need to define a hash function, we just need a way to tell our program the rules to sort elements. If we run the code in 3.40, but replacing `unordered_map<Point, int, PointHasher>` with `map<Point, int>` we get the following result:

```
(0 0): 1
(1 1): 2
(1 2): 2
(2 1): 1
```

The output confirms the rules set, points are sorted by their x-coordinate, and in case of a tie, by their y-coordinate.

Operations on a map take $O(\log n)$, while on unordered maps take $O(1)$ on average, so at least that keeping the data sorted is important for the problem that you are trying to solve, perhaps is better to go with unsorted maps for your implementation, just keep in mind to be careful if you need to define your own hash function.

3.3.5 Stack

STL library has implemented a stack with its respective methods push, pop and empty. Other methods important are top and size. As we can see in section 3.1.1 the implementation of a stack from scratch perhaps is not that difficult, but it can take some time, specially if we deal with dynamic memory, and since it is a data structure frequently used, is better to use the built-in implementation from the STL library. Code 3.42 show a simple use case of a stack, that inserts, retrieve and removes elements.

Listing 3.42: STL: Stack Example

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
  stack<int> S;
  S.push(2);
  S.push(4);
  S.push(1);
  S.push(7);
  S.push(5);

  while (!S.empty()) {
    cout << S.top() << "\n";
    S.pop();
  }

  return 0;
}
```

The `push` method inserts an element into the stack, remember that the last element inserted will be the one at the top. The `empty` method returns `true` if the stack has no elements, otherwise

returns `false`. `top` returs the last element inserted into the stack, and finally `pop` deletes the element at the top. Those methods are the most commonly used, as long with `size` that returns the number elements contained in the stack. In you want to know more about the stack template from the STL library you can review the reference page http://www.cplusplus.com/reference/stack/stack/.

Below is the output of the program 3.42, as you can see the order on which the elements are taken out from the stack is the reverse of how they were inserted, that is expected, since the last element inserted is placed at the top.

```
5
7
1
4
2
```

3.3.6 Queue

STL library also has its own queue implemented with its respective methods push and pop as well. As queue template also has useful methods like front, back, size and empty. Code in 3.43 shows an example of using the `queue` library. To insert elements into the queue we use the `push` method, just like we did with the `stack` example on 3.42, but internally the last inserted element goes to the back of the queue. `empty` returns `true` if the queue has no elements, `false` otherwise. `pop` removes the first element of the queue, which corresponds to the first element inserted. The main difference with the stack template is in the methods for retrieval, here we use `front` instead of `top` to retrieve the first element of the queue, and as you can image there is also a `back` method that returns the last element on the queue, but there is no function to remove the last element.

Listing 3.43: STL: Queue Example

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4
5  int main() {
6    queue<int> Q;
7    Q.push(2);
8    Q.push(4);
9    Q.push(1);
10   Q.push(7);
```

```
11    Q.push(5);
12
13    while (!Q.empty()) {
14      cout << Q.front() << "\n";
15      Q.pop();
16    }
17
18    return 0;
19  }
```

Below is the output for this example, and there are no surprises here, the first element inserted is the first element printed, and the last inserted element was the last to be printed.

```
2
4
1
7
5
```

Even when this example seems simple, queues are particularly handy when we deal with graph traversals, as we will see on chapter 7, since they can be used to compute the path of minimum length to get from one node to another.

The previous code uses some of the most common methods for the `queue` library, but if you want to know more, check the reference in http://www.cplusplus.com/reference/queue/queue/, it describes with more detail each one of the methods available.

3.3.7 Priority queue

A priority queue is basically the same as a heap, internally is implemented like that. As we explained before, a heap is a binary tree where the parents nodes has a comparison relationship between themselves and his children. Therefore we can insert and delete in a time complexity of $O(log)$, as well as consult for the largest or smallest element in $O(1)$.

The program 3.44 shows an example of how use a `priority_queue`. As you can see, we need to import the same `queue` library used before, then we create a heap of integers by using `priority_queue<int>`. After that we insert the same elements that are showed in figure 3.6. The `top` method returns the root, which is the largest element, in this case 12. The `pop` method removes the root from the heap, that causes that the 2^{nd} largest element to become the root.

Listing 3.44: STL: Priority Queue Example

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
  priority_queue<int> heap;
  heap.push(1);
  heap.push(2);
  heap.push(0);
  heap.push(6);
  heap.push(5);
  heap.push(3);
  heap.push(4);
  heap.push(12);
  heap.push(10);
  heap.push(7);
  heap.push(6);
  heap.push(4);
  heap.push(9);
  heap.push(6);
  heap.push(8);

  while (!heap.empty()) {
    cout << heap.top() << "\n";
    heap.pop();
  }

  return 0;
}
```

The output for this program is showed below, and since we are printing the root then removing it, we will see the inserted numbers printed from the largest (12) to the smallest (0).

```
12 10 9 8 7 6 6 6 5 4 4 3 2 1 0
```

Heaps are not useful for searching arbitrary elements, instead they are useful for fast access to the maximum or minimum value of the data, that is why `top` and `pop` interacts with the root, but there are no method `find` like in vectors or lists.

3.4 Chapter Notes

Data structures are a fundamental part of computer science, and for competitive programming it is not the exception. In most cases the solution of a problem lies on finding the right data structure. If you find yourself writing a very confusing code, and you feel that is taking more time of what it should, is possible that you chose the wrong data structure.

Is a good practice to use libraries instead of writing your own data structures, the implementations in libraries, specially in the STL library are well tested and are reliable.

There a lot of information about data structures, too much, that sometimes that can be overwhelming. It can happen that after trying to solve a problem you find out that there is a data structure that does exactly what you were looking for. In this book we wanted to show the most common data structures, such as stack, queue, list, etc, but also some that are not that common, or that don't appear regularly in books, such as segment trees, BIT, and Tries. Some good references are *"Introduction to Algorithms"* [1], *"Algorithms"* [5], and Topcoder [6].

The problems in the exercises section and in appendix B are based on real problems that we have faced. The solutions are not complicated if you choose the right data structure from the beginning, but they can turn complicated if you don't.

if you think you need to more practice, the following links contains problems that focus on data structures.

- `https://www.hackerrank.com/domains/data-structures`
- `https://acm.timus.ru/problemset.aspx?space=1&tag=structure`
- `https://codeforces.com/problemset?tags=data+structures`

3.5 Exercises

1. An online retail store receives n orders sequentially, one after the other. Each one of the orders has a retail price. Every time that an order is placed, the boss of the company wants to know the mean of the retail price for the last k orders, in order to identify trends in the shopping behavior of the customers. As software engineer of this company you need to write a program that solves your boss's problem, maybe then you can finally have a raise.

   The input of your program consist on two integers n ($1 \leq n \leq 10^6$) and k ($1 \leq k \leq 10^3$). The following n lines contains the retail price of each order.

   For each order print the mean price of the last k orders, if there are less than k orders at the moment, prints the mean of all the orders so far.

2. Congratulations! You have been hired as software engineer of the most famous online book store. Your first task is to write an algorithm to recommend the best ranked books similar to one book that a customer bought. A book has the following information:

   - A title
   - A list of similar books.
   - A ranking based on customer reviews.

   The new feature that the company wants to launch is that if book A is similar to book B, and book is similar to book C, then book A is also similar to book C. Write a program that given a book A prints the best k ranked books similar to A.

   The input of your program consists of two integers, n ($2 \leq n \leq 10^6$), representing the total number of books in the inventory, and k ($1 \leq k \leq 10^3$), being $n \geq k$.

The next n lines describe the books in the following form:

`title rank` m S

Where `title` is the title of the book, `rank` is the ranking of the book, m is the number of books that are similar to the current book, and S is a list of book titles that are similar to the current book, titles are separated by a space. The last line of the input will consists on the title of book A.

The output consists on k lines representing the top k ranked books similar to book A. Each line must consists of the book title and ranking separated by a space. Print the books sorted in decreasing order based on their ranking. It is guaranteed that all book titles are different.

3. Word search puzzle is game where a dictionary of n words is given to you, along with a $r \times c$ matrix M containing only uppercase letters, the goal is to find all the words in the dictionary inside M, the words can be written forwards or backwards in any of the 8 directions (vertically, horizontally, diagonally). Write a program that given a dictionary and a puzzle, finds in the puzzle each word of the dictionary.

4

Sorting Algorithms

"Code is like humor. When you have to explain it, it's bad."

– Cory House

In practice is common to face with the necessity of sorting the data which we are working with to solve a specific problem. Sometimes is necessary due the memory and time constraints, or maybe because we want to improve the execution time of our code, in any case we are talking about a fundamental tool in computer science.

It is true that in real-life situations perhaps we ended up using a library containing some sorting methods, but understanding how algorithms work internally is fundamental, since at the end we need to justify why we choose a certain algorithm among others.

Sometimes happens that we are not familiar with some of the libraries at the time of a contest or interview, and we find ourselves in the position of implementing a solution from scratch. That's why simple algorithms like *Bubble Sort* and *Selection Sort* don't need to be discarded, they can be implemented fast, and if the number of elements to sort is small, they will work just fine.

In the case of technical interviews, is unlikely that an interviewer will ask you "write a code that sort n amount of numbers", but it is likely to be in a situation where we need to sort some data to solve the given problem, and in that case is better knowing

the different kinds of sorting algorithms, how they work, and their time complexity, since at the end you will need to justify and defend your solution.

In this chapter we will cover some of the most popular sorting algorithms. Depending on the circumstances there are algorithms that fit better, some of them are faster in execution time but are more difficult to implement, or sometimes they need more amount of memory. On the other hand, there are algorithms that perform poorly in execution time, but are very easy to implement. That's why is important to identify which algorithm is the best for the problem that needs to be solved.

4.1 Bubble Sort

Bubble sort is one of the most popular sorting algorithms, is easy to understand and easy to implement. Unfortunately is hard to use it in real-life applications.

The algorithm consists in the following. Having an array X with n elements, $x_0, x_1, \ldots, x_{n-1}$. Iterate from $i = 0$ to $i = n - 2$ comparing element x_i with element x_{i+1}, if case $x_i > x_{i+1}$ swap both values.

Consider the following array of numbers.

5	3	1	9	0	4	7	2	8	6

The iterations of the bubble sort method are shown in table 4.1.

Iteration 1:	**5**	**3**	1	9	0	4	7	2	8	6	Swap 5 and 3
Iteration 2:	3	**5**	**1**	9	0	4	7	2	8	6	Swap 5 and 1
Iteration 3:	3	1	**5**	**9**	0	4	7	2	8	6	Nothing happens
Iteration 4:	3	1	5	**9**	**0**	4	7	2	8	6	Swap 9 and 0
Iteration 5:	3	1	5	0	**9**	**4**	7	2	8	6	Swap 9 and 4
Iteration 6:	3	1	5	0	4	**9**	**7**	2	8	6	Swap 9 and 7
Iteration 7:	3	1	5	0	4	7	**9**	**2**	8	6	Swap 9 and 2
Iteration 8:	3	1	5	0	4	7	2	**9**	**8**	6	Swap 9 and 8
Iteration 9:	3	1	5	0	4	7	2	8	**9**	**6**	Swap 9 and 6

Table 4.1: Iterations of the *Bubble Sort.*

Resulting the following array:

3	1	5	0	4	7	2	8	6	9

In table 4.1 can be seen that greater values move to the end of the array, meanwhile the smaller ones move to the start of the array.

In order to sort the array, the process described above must be repeated a maximum of n times. This to ensure that all elements end in the correct position. The worst case scenario is when the array is initially in decreasing order, because in each iteration the smallest element moves only one position.

Program 4.1 read a number n ($1 \leq n \leq 100$), indicating the number of elements in the array X. The next n numbers represents the elements of X. The program print the array X with its elements sorted in increasing order.

Time Complexity: $O(n^2)$
Input:
n: Number of elements in the array.
X: Array to be sorted.
Output:
The array X in non-decreasing order.

Listing 4.1: Bubble Sort

```
#include <algorithm>
#include <cstdio>
#define N 101
using namespace std;

int X[N];
int n;
```

```
8
9   void bubbleSort();
10
11  int main() {
12    scanf("%d", &n);
13    for (int i = 0; i < n; i++) {
14      scanf("%d", &X[i]);
15    }
16
17    bubbleSort();
18
19    for (int i = 0; i < n; i++) {
20      printf("%d ", X[i]);
21    }
22
23    printf("\n");
24    return 0;
25  }
26
27  void bubbleSort() {
28    for (int i = 0; i < n; i++) {
29      for (int j = 0; j < n - 1; j++) {
30        if (X[j] > X[j + 1]) {
31          swap(X[j], X[j + 1]);
32        }
33      }
34    }
35  }
```

4.2 Selection Sort

Given an array of n elements, this algorithm find the smallest element in an array and place it in the first position, then finds the smallest number from the remaining elements and place it in the second position, and so on, until all elements are sorted.

In iteration i, the cost of finding the smallest element is $n - i$, because at this point we are sure that the first $i - 1$ elements are already sorted. Let's see an example. Consider the following array.

5	3	1	9	0	4	7	2	8	6

The iterations of the algorithm are described in table 4.2.

Iteration 1:	**5**	3	1	9	**0**	4	7	2	8	6	Swap 0 and 5
Iteration 2:	0	**3**	**1**	9	5	4	7	2	8	6	Swap 3 and 1
Iteration 3:	0	1	**3**	9	5	4	7	**2**	8	6	Swap 3 and 2
Iteration 4:	0	1	2	**9**	5	4	7	**3**	8	6	Swap 9 and 3
Iteration 5:	0	1	2	3	**5**	**4**	7	9	8	6	Swap 5 and 4
Iteration 6:	0	1	2	3	4	**5**	7	9	8	6	Keep 5 in the same place
Iteration 7:	0	1	2	3	4	5	**7**	9	8	**6**	Swap 7 and 6
Iteration 8:	0	1	2	3	4	5	6	**9**	8	**7**	Swap 9 and 7
Iteration 9:	0	1	2	3	4	5	6	7	8	9	Array sorted

Table 4.2: Iterations of the *Selection Sort*.

In table 4.2 the numbers in red are numbers that are already in correct positions, and the numbers in bold are the ones that need to be swapped.

In the first element we need to iterate trough all n elements, for the second one we need to iterate trough $n-1$ elements, and so on. So the number of iterations is given by $n + (n-1) + (n-2) + \cdots + 1 = \frac{n}{2}(n+1)$.

The code in 4.2 reads an integer n ($1 \leq n \leq 100$). n numbers follow, representing the elements of the array to be sorted. The program prints the array sorted in ascending order using the *Selection Sort* algorithm.

Time Complexity: $O(n^2)$
Input:
n: Number of elements in the array.
X: Array to be sorted.
Output:
The array X in non-decreasing order.

Listing 4.2: Selection Sort

```
#include <algorithm>
#include <cstdio>
#define N 101
using namespace std;

int X[N];
int n;

void selectionSort();

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &X[i]);
```

```
  }

  selectionSort();

  for (int i = 0; i < n; i++) {
    printf("%d ", X[i]);
  }
  printf("\n");

  return 0;
}

void selectionSort() {
  for (int i = 0; i < n - 1; i++) {
    int pos = i;
    for (int j = i + 1; j < n; j++) {
      if (X[j] < X[pos]) {
        pos = j;
      }
    }

    swap(X[i], X[pos]);
  }
}
```

4.3 Insertion Sort

To explain this method imagine we have an array with its elements sorted and we want to add a new element. To keep the array sorted we need to place the new element in the correct position. To achieve that, all the elements greater than the new element must be shifted to the right in order to make space to the new element. Consider the following array:

1	3	6	7	10

To add a new element with a value of 5, we must shift to the right the elements greater than 5, and place the new element in the correct position.

1	3	5	6	7	10

In the worst case the elements are given in descending order, making in each iteration to shift all the elements in the array. The program in 4.3 reads an integer n ($1 \leq n \leq 100$), indicating the number of elements in the array. Next n numbers represent the

elements in the array. The program uses *Insertion Sort* to print the given array in ascending order.

Time Complexity: $O(n^2)$
Input:
n: Number of elements in the array.
X: Array to be sorted.
Output:
The array X with its elements in non-decreasing order.

Listing 4.3: Insertion Sort

```
#include <cstdio>
#define N 101
using namespace std;

int X[N];
int n;

int main() {
  int j, num;

  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &num);
    j = i;
    while (j > 0 && num < X[j - 1]) {
      X[j] = X[j - 1];
      j--;
    }
    X[j] = num;
  }

  for (int i = 0; i < n; i++) {
    printf("%d ", X[i]);
  }
  printf("\n");

  return 0;
}
```

4.4 Quick Sort

This algorithm was discovered by Antony Richard Hoare [7] at the end of the 50's and begin of the 60's. The main idea behind this algorithm is to place an element called *pivot* in a position where the elements at its left are smaller and the elements at its right are bigger or equal.

Consider an array x_0, x_1, x_2, ... , x_{n-1}. If we take the last

Algorithm 1 *quicksort*(a, b)

$X_{pivot} = X_b$
$i \leftarrow a$
$j \leftarrow b - 1$
while $i \geq j$ **do**
  if $X_i < X_{pivot}$ **then**
    $i \leftarrow i + 1$
  else if $X_j \geq X_{pivot}$ **then**
    $j \leftarrow j - 1$
  else if $X_i > X_{pivot}$ and $X_j < X_{pivot}$ **then**
    $swap(X_i, X_j)$
    $i \leftarrow i + 1$
    $j \leftarrow j - 1$
  end if
end while
$swap(X_i, X_{pivot})$
$quicksort(a, i - 1)$
$quicksort(i + 1, b)$

element, x_{n-1}, as the pivot, and place an iterator i at position 0, and another iterator j at position $n - 2$. If x_i is smaller than the pivot, increase i in one. Also if x_j is greater or equal than the pivot, decrease j in one. In the case that x_i is greater or equal than the pivot and that x_j is smaller than the pivot, swap both elements and continue. The process stops when i is greater than j. Finally just swap x_i and the pivot. This will ensure that the pivot is in the correct position. Repeat the process with the sub-array in the left side of the pivot and with the sub-array in the right side of the pivot. At the end, the whole array will be sorted. Algorithm 1 shows the logic behind *Quick Sort* to sort an array X from position a to position b.

The worst case scenario is when all elements are sorted in non-ascending order, because all the elements will be located in the right side of the pivot in each iteration, for the first pivot there will be $n - 1$ elements at its right, for the next pivot there will be $n - 2$, and so on, in order to avoid this, many algorithms run a random sort algorithm before execute quicksort reaching $O(nlogn)$ most of the times.

The code in 4.4 implements *Quick Sort* to sort an array X of n elements ($1 \leq n < 20$). The input consists of number n, and the n numbers that form X. The output is X with its elements sorted in ascending order.

Time Complexity: $O(n^2)$,
It's important to say the the average time complexity of this algorithm is $O(nlogn)$, and that's why is widely used.
Input:
n: The number of elements in the array.
X: Array to be sorted.
Output:
The array sorted.

Listing 4.4: Quick Sort

```
#include <algorithm>
#include <cstdio>
using namespace std;

int X[20];
int n;

void quicksort(int, int);
int getPivot(int, int);

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &X[i]);
  }

  quicksort(0, n - 1);

  for (int i = 0; i < n; i++) {
    printf("%d ", X[i]);
  }
  printf("\n");

  return 0;
}
```

The function `quicksort` defined bellow, receives two integers that corresponds to the interval to be sorted. Using the pivot it call itself to sort the sub-interval at the left of the pivot and the sub-interval at the right od the pivot.

```
void quicksort(int a, int b) {
  if (a < b) {
    int p = getPivot(a, b);
    quicksort(a, p - 1);
    quicksort(p + 1, b);
  }
```

```
}
```

The key part in the *Quick Sort* algorithm consists on placing in the right position the pivot. The function `getPivot` place the pivot in the correct position in the interval $[a, b]$ according to the algorithm described before.

```
int getPivot(int a, int b) {
  int i = a;
  int j = b - 1;
  int p = b;
  while (i <= j) {
    if (X[i] < X[p]) {
      i++;
    } else if (X[j] >= X[p]) {
      j--;
    } else if (X[i] >= X[p] && X[j] < X[p]) {
      swap(X[i++], X[j--]);
    }
  }

  swap(X[i], X[p]);
  return i;
}
```

4.5 Counting Sort

This algorithm is useful when we are handling a big amount of data, but is recommended to use it only when the data can be expressed as non-negative integer numbers in a small interval. The main idea is to count occurrences in terms of the numerical value, for example, consider the following integers in the interval $[0, 9]$:

$$0, 0, 1, 3, 5, 3, 2, 1, 0, 1, 3, 7, 8, 2, 1, 3, 5, 6, 5, 3$$

For this case we have three 0's, four 1's, two 2's, five 3's, zero 4's, three 5's, one 6, one 7, one 8 and zero 9's. Then, we only need to store the occurrences in a vector C of size equal to the biggest element in the data, where C_k represents the number of occurrences of element k. The vector C for this example looks as follows:

	0	1	2	3	4	5	6	7	8	9
$C =$	3	4	2	5	0	3	1	1	1	0

The program in 4.5 receives a number n, indicating the amount of elements to be sorted. n numbers follow, each one in the

interval $[0, 9]$. The output is the numbers from the input sorted in ascending order.

Time Complexity: $O(n)$
Input:
n: Number of elements in the array.
num: Elements to be sorted.
Output:
The array sorted.

Listing 4.5: Counting Sort

```
#include <cstdio>
#define N 10
using namespace std;

int C[N];
int n;

int main() {
  int num;

  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &num);
    C[num]++;
  }

  for (int i = 0; i < N; i++) {
    for (int j = 0; j < C[i]; j++) {
      printf("%d ", i);
    }
  }
  printf("\n");

  return 0;
}
```

4.6 Merge Sort

Suppose we want to sort the elements of a vector X in non-decreasing order from positions a to b. This algorithm consists of four steps:

1. Divide the vector in two parts by finding the middle value X_m, where $m = (a + b)/2$.

2. Sort the elements in the left side.

3. Sort the elements in the right side.

4. Combine the elements in the left side with the elements in the right side in such a way that the resulting vector is sorted.

Step 4 is the most important. Figure 4.1 shows the merge process of array A and array B into an array C, where A and B are sorted in non-decreasing order. Basically the idea of the merging process consists on placing an iterator i (red) at the beginning of array A, and an iterator j (blue) at the beginning of array B. If $A_i < B_j$ the element A_i is inserted at the end of array C and i is moved to the next position. Otherwise if $A_i \geq B_j$, element B_j is inserted at the end of C and j is moved to the next position. The process continues until all the elements of either A or B are inserted into C.

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0]$

(a) Iteration 1

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0, 1]$

(b) Iteration 2

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0, 1, 2]$

(c) Iteration 3

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0, 1, 2, 5]$

(d) Iteration 4

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0, 1, 2, 5, 6]$

(e) Iteration 5

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0, 1, 2, 5, 6, 7]$

(f) Iteration 6

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0, 1, 2, 5, 6, 7, 9]$
(g) Iteration 7

$A = [0, 2, 5, 9, 10]$
$B = [1, 6, 7, 12, 16]$
$C = [0, 1, 2, 5, 6, 7, 9, 10]$
(h) Iteration 8

Figure 4.1: Iterations of the merge process of two arrays previously sorted

Once one of the iterators reach the end of the array, we just add to C the remaining elements of the other array. Now C contains all the elements of A and C in non-decreasing order.

$$C = [0, 1, 2, 5, 6, 7, 9, 10, 12, 16]$$

At the beginning there are n elements in the array, then it is split in two halves of $n/2$ elements each, and each half is divided again in two halves of $n/4$ elements, and so on. Then the execution time depends on how many times we divide the array. We know that we cannot divide the array if there is only one element, this is when $n/2^k = 1$. Solving for k, we have that $k = \log_2 n$, and each time we need to perform the merge process, so the time complexity of the *Merge Sort* is $O(n \log n)$.

The program 4.6 receives and integer n ($1 \leq n \leq 100$) representing the number of elements in the array X. The next n numbers are the elements of X. The output is the array X sorted using the *Merge Sort* algorithm.

Time Complexity: $O(n \log n)$
Input:
n: The number of elements in the array.
X: Array to be sorted.
Output:
The array sorted.

Listing 4.6: Merge Sort

```
#include <cstdio>
#define N 101
using namespace std;

int X[N], C[N];
int n;
```

```
8  void mergeSort(int, int);
9  void merge(int, int, int);
10
11 int main() {
12   scanf("%d", &n);
13
14   // Read the numbers to be sorted
15   for (int i = 0; i < n; i++) {
16     scanf("%d", &X[i]);
17   }
18
19   // Apply Merge Sort
20   mergeSort(0, n - 1);
21
22   // Print the sorted array
23   for (int i = 0; i < n; i++) {
24     printf("%d ", X[i]);
25   }
26   printf("\n");
27
28   return 0;
29 }
```

The function `mergeSort` receives an interval of the elements to sort, calculates the middle element, and recursively call itself again to sort both halves of the interval. Finally both halves are merged sorting all the elements in the interval.

```
1  void mergeSort(int i, int j) {
2    if (i != j) {
3      int m = (i + j) / 2;
4      mergeSort(i, m);
5      mergeSort(m + 1, j);
6      merge(i, m, j);
7    }
8  }
```

The merge process explained before takes place in the `merge` function, which receives the indexes i and j of the interval to sort, and the middle point m, and sort both halves of the array.

```
1  void merge(int i, int m, int j) {
2    // p and q are the indexes that will move across
3    // each half respectively.
4    int p = i;
5    int q = m + 1;
6    int r = i;
7
8    // Keep comparing the values of X[p] and X[q]
9    // until we reach the end of one of the halves
10   while (p <= m && q <= j) {
11     if (X[p] <= X[q]) {
12       C[r++] = X[p++];
13     } else {
14       C[r++] = X[q++];
15     }
16   }
17
```

```
18    // Add the remaining elements of the first half
19    while (p <= m) {
20      C[r++] = X[p++];
21    }
22
23    // Add the remaining elements of the second half
24    while (q <= j) {
25      C[r++] = X[q++];
26    }
27
28    // Update the original array
29    for (r = i; r <= j; r++) {
30      X[r] = C[r];
31    }
32  }
```

4.7 Heap Sort

The *Heap Sort* algorithm was invented by Joseph Williams in 1964. [8]. The main idea of this algorithm is to store the elements of a vector in a heap. A heap is a binary tree where the value of a node is greater or equal than the value of its children. If the value of a node is smaller than the value of one of its children, then the node is swapped with the child with largest value.

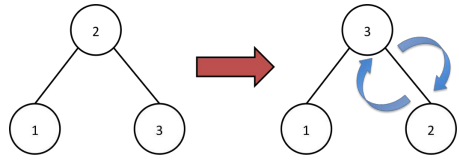


Figure 4.2: Swapping process in heap sort

Let's explain how to implement a binary tree in an array. Suppose we have a parent node in the position i then the left child would be in the position $2i + 1$ and the right child in the position $2i + 2$. For instance the root would be at index 0, its left child at index 1 and its right child at index 2, now the left child of the node located at index 1 would be at index 3 and the right child at index 4, for the node placed at index 2 the left child would be at index 5 and the right child at index 6, and so on. By doing this process we obtain the entire binary tree.

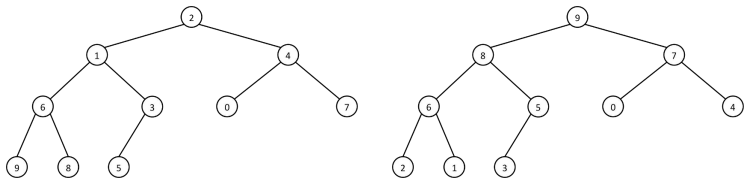
Consider the following sequence of numbers:

	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]	X[9]
$X =$	2	1	4	6	3	0	7	9	8	5

The heap can be constructed in different ways. One option is to store all elements in a tree like in 4.3a, and then starting from the last element to the first, swap each element the times it is necessary until it reaches a valid position, the result of doing this is showed in 4.3b.

Another option to build a heap consists on placing the elements in the correct position at the same time we are reading the numbers. That could led us to a different heap, but still a valid one.

We can use any method to build the heap, as long as we don't break the rule that a node must be greater or equal that its children. The maximum value will always be the root, no matter the method chosen to construct the heap.



(a) Vector X in a tree structure before swaps (not heap)

(b) Vector X in a heap structure after swaps

Figure 4.3: Building a heap sort with the input data. A node is greater or equal than its children.

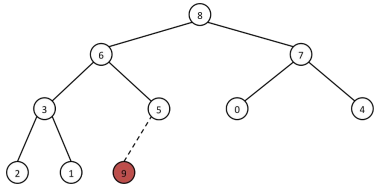
The array representation for the heap in 4.3b is the following:

	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]	X[9]
$X =$	9	8	7	6	5	0	4	2	1	3

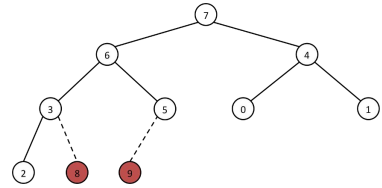
Table 4.3: Heap represented with an array

The next step is to swap the first element of X (root node) with the last element. Then remove the last element from the heap because it is already in the right position of the array, and move the new root to the correct position using swap operations. This will convert the tree into a heap again. Continue doing this until

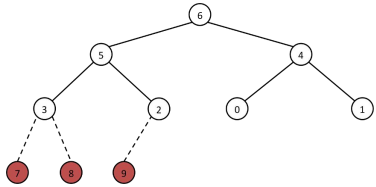
all elements in the heap have been removed. Figure 4.4 shows the different iterations of the *Heap Sort*.



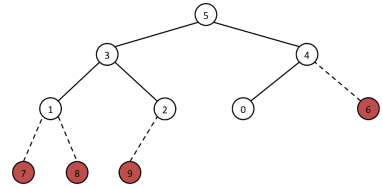
(a) Swap nodes 3 and 9, and rearrange the heap with the remaining nodes.



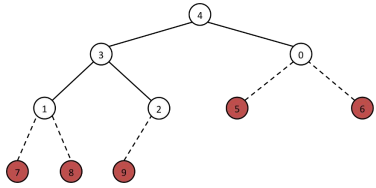
(b) Swap nodes 1 and 8, and rearrange the heap with the remaining nodes



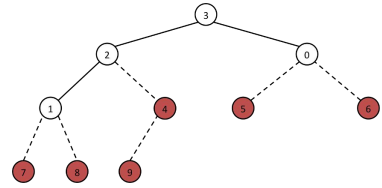
(c) Swap nodes 2 and 7, and rearrange the heap with the remaining nodes.



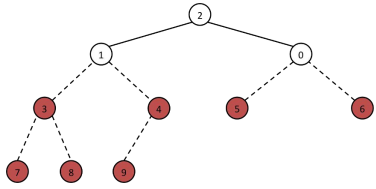
(d) Swap nodes 1 and 6, and rearrange the heap with the remaining nodes



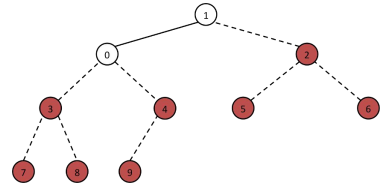
(e) Swap nodes 0 and 5, and rearrange the heap with the remaining nodes.



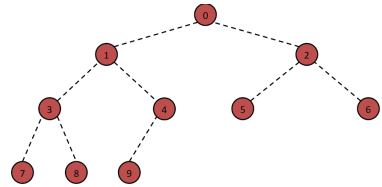
(f) Swap nodes 2 and 4, and rearrange the heap with the remaining nodes.



(g) Swap nodes 1 and 3, and rearrange the heap with the remaining nodes.



(h) Swap nodes 0 and 2, and rearrange the heap with the remaining nodes.



(i) Swap nodes 0 and 1, and rearrange the heap with the remaining nodes.

Figure 4.4: Iterations of heap sort

Because we are using a binary tree, the time complexity of moving an element to the correct position in the heap is $O(\log n)$, and because we need to do it for all n elements in the array, the time complexity of the *Heap Sort* is $O(n \log n)$. The code in 4.7 receives an array X of n elements, where $(1 \leq n \leq 100)$ and print the same array X, but with its elements sorted in ascending order.

Time Complexity: $O(n \log n)$
Input:
n: The number of elements in the array.
X: Array to be sorted.
Output:
The array sorted.

Listing 4.7: Heap Sort

```
#include <algorithm>
#include <cstdio>
#define N 101
using namespace std;

int X[N];
int n;

void heapSort();
void makeHeap();
void downHeap(int);

int main() {
  int m;

  // Read the number of elements and the array to be sorted
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &X[i]);
  }

  // Apply Heap Sort
  m = n;
  heapSort();

```

```
  // Print the sorted array
  for (int i = 0; i < m; i++) {
    printf("%d ", X[i]);
  }
  printf("\n");

  return 0;
}
```

The `heapSort` function, first converts the array X into a heap by calling the function `makeHeap`, then as explained before, the last element and the first element are swapped, that will cause that the last element to be in the correct position. The heap has to built again, but leaving aside the last element. The process is repeated until the array is completely sorted.

```
void heapSort() {
  makeHeap(); // Convert the original array into a heap
  while (n > 1) {
    swap(X[0], X[n - 1]); // Swap the last element and the root
    n--;                  // Ignore the last element from the heap
    downHeap(0);          // Place the current root in the correct position
  }
}
```

Given an array X the `makeHeap` function converts X into a heap. See the example in figure 4.3.

```
void makeHeap() {
  for (int i = n / 2 - 1; i >= 0; i--) {
    downHeap(i);
  }
}
```

The function `downHeap` is essential for the correct operation of the program, it receives an integer k, and place the element X_k in the correct position of the heap by doing the necessary swap operations described in 4.2.

```
void downHeap(int k) {
  int w = 2 * k + 1;
  while (w < n) {
    if (w + 1 < n && X[w + 1] > X[w]) {
      w++;
    }

    if (X[k] >= X[w]) {
      return;
    }

    swap(X[k], X[w]);
    k = w;
    w = 2 * k + 1;
  }
```

```
}
```

4.8 Sorting With the `algorithm` Library

Now that we know some of the most popular sorting algorithms, we will see how to make our life easier when we have to sort a set of comparable elements.

Most programming languages have a library that help us sort. In `C++` there is a library called `algorithm` that contains the `sort` method that guarantees a complexity time of $O(n \log n)$. The interesting thing here is that this library allows us to customize our own sorting rules.

In order to use the `algorithm` library we only have to add it and use the `sort` method. Let's see some code examples and take a look to the comments inside the code.

Listing 4.8: Simplest sort function

```
// This example shows the simplest form to use sort function
#include <stdio.h>
#include <algorithm> // necessary to use the sort method
#define N 10

using namespace std;

void print(int *A, int length) {
  for (int i = 0; i < length; i++) {
    printf("%d ", A[i]);
  }
}

int main() {
  int arr[N] = {3, 5, 7, 8, 2, 1, 0, 9, 10, 11};

  printf("Unordered array:\n");
  print(arr, N);

  /* sort function uses two parameters, they are the initial
     and final position of the elements that will be sorted */
  sort(arr, arr + N);

  printf("\n");
  printf("Sorted array:\n");
  print(arr, N);

  return 0;
}
```

4.8.1 Overloading operator <

So far we have only ordered numbers in a non-decreasing way and based on one parameter, but what if we want to sort in a non-increasing way or taking two parameters into account, to do that we can modify the behavior of the `sort` function by overloading the operator <. Let's review how to implement it with the following two examples.

The first example in 4.9 sort an array in non-increasing order by overloading the < operator.

Listing 4.9: Sorting in a non-increasing way

```
// This example shows how to sort an array in a non-increasing way
#include <stdio.h>
#include <algorithm>
#define N 10
using namespace std;

/* We need a structure to overload
   the operator < */
struct _int {
  int x;
};

// Function to print the array elements
void print(_int *A, int length) {
  for (int i = 0; i < length; i++) {
    printf("%d ", A[i].x);
  }
}

/* Here we are overloading < operator.
   What we are doing is simply converting
   < operator to > operator.
   In this way we can sort the elements
   in non-increasing form.*/
bool operator<(_int A, _int B) { return (A.x > B.x); }

int main() {
  _int arr[N] = {2, 5, 7, 7, 2, 1, 0, 9, 10, 11};
  printf("Unordered array:\n");
  print(arr, N);

  /* sort function uses two parameters, they are the initial and final
       position
   * of the elements that will be sorted */
  sort(arr, arr + N);

  printf("\n");
  printf("Sorted array in non-increasing way:\n");
  print(arr, N);
  return 0;
}
```

In the second example in 4.10, we sort an array of points, first in increasing order according to their x-coordinate, and if two points have the same x-coordinate, then they are sorted in increasing order by their y-coordinate.

Listing 4.10: Sorting pairs in non-increasing form

```
// This example shows how to sort a list of pairs in non-decreasing way
#include <stdio.h>
#include <algorithm>
#define N 10
using namespace std;

/* We need a structure _pair
   to overload the operator < */
struct _pair {
  int x;
  int y;
};

// Function to print the pairs
void print(_pair *A, int length) {
  for (int i = 0; i < length; i++) {
    printf("%d %d\n", A[i].x, A[i].y);
  }
}

/* As the x parameter is the first criteria to sort,
   we check first the 'x' member, then if A and B have
   the same 'x' we compare 'y' parameter, making 'y'
   second sorting criteria. */
bool operator<(_pair A, _pair B) {
  return (A.x < B.x || (A.x == B.x && A.y < B.y));
}

int main() {
  // Pair list
  _pair arr[N] = {{2, 3}, {5, 7}, {5, 2}, {1, 0}, {1, -1},
                  {-2, 2}, {-2, 0}, {-2, 0}, {0, 0}, {0, 1}};
  printf("Unordered array of pairs:\n");
  print(arr, N);

  /* sort function uses two parameters, they are the initial and final
       position
   * of the elements that will be sorted */
  sort(arr, arr + N);

  printf("\n");
  printf(
      "Pairs sorted in non-decreasing order by x coordinate as first criteria
           "
      "and y coordinate as second criteria:\n");
  print(arr, N);
  return 0;
}
```

4.8.2 Adding a function to sort

Sometimes it is better to add a function instead of overloading an operator, this is used when you need to sort a complex structure or in a special way. For example suppose we have to sort characters, digits [0-9] and letters [a-z][A-Z], in such way that the letters have greater priority than numbers. To do that we need to implement a function that allows us to alter the ASCII natural order, in which the numbers come first than the letters. Let's look at the code to see how it works.

Listing 4.11: Sorting the letters first than the digits

```
// This example shows how to implement a custom function to sort
#include <stdio.h>
#include <algorithm>
#define N 10
using namespace std;

// Function to print the array elements
void print(char *A, int length) {
  for (int i = 0; i < length; i++) {
    printf("%c ", A[i]);
  }
}

bool sortFirstLetters(char A, char B) {
  if (A >= '0' && A <= '9') {
    if (B >= '0' && B <= '9') {
      // A and B are numbers
      return (A < B);
    } else {
      // A is a number and B is a letter
      return false;
    }
  }

  if (B >= '0' && B <= '9') {
    // A is a letter and B is a number
    return true;
  }

  // A and B are letters
  return (A < B);
}

int main() {
  // Character array
  char arr[] = {'7', '5', '3', '1', 'a', 'n', 'z', 'A', 'N', 'Z'};
  printf("Array as defined: \n");
  print(arr, N);
  printf("\n");

  // Here we use the normal sort
  sort(arr, arr + N);
  printf("Sorted array according to ASCII table: \n");
  print(arr, N);
  printf("\n");
```

```
46
47    // Here we use custom function in order to sort by our criteria
48    sort(arr arr + N, sortFirstLetters);
49    printf("Sorted array according to letters first than digits criteria: \n");
50    print(arr, N);
51    return 0;
52  }
```

4.9 Chapter Notes

Fastest sorting algorithms run in $O(n \log n)$ time, but they are sometimes a little bit harder to implement. On the other hand, algorithms that run in $O(n^2)$ time tend to be easier to implement, but their performance is poor when dealing with a large amount of data. For contests, when the number of elements to sort is greater than 1000, we recommend not to use a $O(n^2)$ algorithm. In fact is a good practice to use the `sort` function of the `algorithm` library whenever is possible.

Is important to keep in mind those methods that are not based on comparisons, like the Counting Sort. They are faster for some situations and are easy to implement.

The book of *Introduction to Algorithms* [1] contains a deeper explanation and analysis of the Heap Sort and Quick Sort algorithms, and provides a great review of some techniques used to sort in linear time like the Counting Sort. Sedgewick [5] makes an analysis of different sorting methods, mentioning their advantages and disadvantages, and include code for most of them written in C/C++. Knuth [9] describes with great detail the performance of different sorting algorithms, always providing a mathematical perspective.

A brief explanation of sorting algorithms can be found in topcoder tutorials [10]. And some problems involving sorting and searching can be found in the following online judges:

- `https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=98`
- `http://acm.timus.ru/problemset.aspx?space=1&tag=beginners`
- `https://leetcode.com/tag/sort/`

Appendix C contains a set of problems with non-trivial solutions that involve sorting.

4.10 Exercises

1. Certain country is divided into regions, and each region has certain number of citizens. They are about to choose their leader, and the only condition for a person to be the leader is that he or she must obtain control of the majority of the regions. A person gain control of a region if the majority of the citizens of that region vote for that person. Write a program that given the number of regions and the numbers of citizens in each region, find the minimum number of votes a person needs to become the leader. The number of regions is no more than 100000.

2. The conclave is the ritual where all cardinals of the Catholic church choose the new pope. The pope is elected among the cardinals and all of them vote for the one they want to be the new pope. The pope is the one that gets at least 2/3 of the votes. For simplicity suppose that the cardinals are numbered from 1 to n. Write a program that reads an integer n ($1 \leq n \leq 1000$) indicating the number of cardinals, followed by n numbers representing the votes of each cardinal. The output is the number associated to the new pope.

3. Write a program that given n points, find the k closest points to the origin. Assume that the values of n and k can be very large, in the order of 10^7, where $k \leq n$. All coordinates are integer values. The input consists of a line containing the values of n and k, followed by n lines, each one defining the coordinates x, y of a point. The output are k lines displaying the coordinates of the closest points to the origin in ascending order according to its Euclidean distance to the origin. Find a solution that runs faster than $O(n \log n)$. See C.3.

5

Divide and Conquer

"It always seems impossible until it's done."
– Nelson Mandela

The *Divide and Conquer* technique is one of the most important tools in algorithms, consists on taking a problem and divide it in smaller sub-problems, and then those sub-problems can be divided in more sub-problems, and so on. This is useful when the original problem is hard to solve or solving it involves a high computational cost, but the sub-problems on which it is divided can be solved in an easier way or with less computational cost, and the solution or those sub-problems can be used to solve the original problem.

There is no rule that tell us when a problem must be solved using *Divide and Conquer*, it is more about to be aware on which cases can be used, because not all problems can be divided in sub-problems, and always keeping in mind that the sub-problems must be less-costly than the original problem, because in that case it would be better to solve the original problem itself instead of dividing it in more complex sub-problems.

In this section we will see problems that are solved using the *Divide and Conquer* technique. Some of them are popular and easy to implement like *Binary Search*, but other involves a more complex solution. In either case, the goal is to give a general perspective of the cases where it is a good option to use *Divide and Conquer*.

5.1 Binary Search

Binary search finds an element in a sorted array. The idea of the algorithm is to look the middle element, and see if it is smaller or larger than the element we are trying to find, if it is smaller, then keep the right half of the array and repeat the process, otherwise, keep the left part of the array and do the same.

The algorithm consists on dividing the array in two halves, until a single element is left, that means that in the first iteration there are n elements, in the next one $n/2$, then $n/4$, and so on, until $n/2^k = 1$, where k is the number of times we divide the array. Solving for k we have that $k = \log n$. So the time complexity for the *Binary Search* is $O(\log n)$. The code in 5.1 implements a binary search to find a number `key` in an array X in the interval $[a, b]$.

Time Complexity: $O(\log n)$
Input:
x: Previously sorted array.
a: Left index
b: Right index
key: The number to be found
Output:
The position where the element *"key"* was found. Otherwise returns -1

Listing 5.1: Binary Search

```
int binarySearch(int X[], int a, int b, int key) {
  int c;

  while (a <= b) {
    c = (a + b) / 2;

    if (key == X[c]) {
      return c;
    } else if (key < X[c]) {
      b = c - 1;
    } else {
      a = c + 1;
    }
  }

  return -1;
}
```

5.2 Binary Exponentiation

Raising a number A to some power B doesn't seem like a big problem, we just need to multiply it B times, but what if $B = 10000000$, it would take a while to compute the result. The trick here is to obtain the binary representation of the power B. For example, if we want to find the value of 7^5, we can do it in the traditional way

$$7^5 = 7 \times 7 \times 7 \times 7 \times 7 = 16807$$

The total number of operations is 5. On the other hand, if we represent 5 as a binary number, then we have that only two multiplications are needed.

$$7^5 = 7^4 \times 7^1 = 16807$$

What about 7^{13}?

$$7^{13} = 7^8 \times 7^4 \times 7^1$$

Only three multiplications are needed. Since the value of A^B can be quite large, sometimes it is asked to return the result modulo M, where M can fit in an integer variable. For that case is good to keep in mind one of the properties of modular arithmetic which states that

$$(a \times b) \bmod m = ((a \bmod m) \times (b \bmod m)) \bmod m. \qquad (5.1)$$

According to 5.1 we only need to store the values of the products that are being made. The program 5.2 receives the values of A, B and C and return the value of $A^B \bmod M$ in the way explained before. In variable S we store the result, A, B and M will be used as mentioned before. Then as long as B is greater than 0, we check the parity of B, if is odd means that in its binary representation there is a 1 and we need to multiply by the current power of A and apply modulo M to the result, the powers of A will be A, A^2, A^4, A^8, and so on, we divide B by 2 because we are using the binary representation of B.

Time Complexity: $O(\log B)$
Input:
Three values: A, B, M
Output:
The value of $A^B MODM$

Listing 5.2: Big Mod (A^B mod M)

```
int bigMod(int A, int B, int M) {
  int S = 1;
  A = A % M;

  while (B > 0) {
    if (B % 2 != 0) {
      S = (S * A) % M;
    }

    A = (A * A) % M;
    B /= 2;
  }

  return S;
}
```

5.3 Closest Pair of Points

Algorithm that use a *Divide and Conquer* strategy to find the distance between the closest pair of points. The algorithm works in the following way.

- Sort the points by their x coordinate.
- Divide an imaginary vertical line that divides the domain in two parts.
- Find the closest pair of points in the left side.
- Find the closest pair of points in the right side.
- Verify if the closest pair of points are in different sides.

To sort the points we used the `sort` function of the `algorithm` library, which is part of the *STL*, for that, we just need to overload the operator `<`, and add the necessary rules, for this case we first compare the x coordinate, and if there is a tie compare the y coordinate.

In the source code of this algorithm showed in 5.3, in each iteration of the function `closestPair` we divide the current domain in two equal parts until a pair of points or a single point is left. If a single point is left, the function returns a big number, indicating that the closest pair is not on that side. On the other hand, if a pair of points is found, the function returns the distance between them. Then we check pair of points formed by points

from different parts to find if there is a closer pair of points. The final result is the distance between the closest pair points.

Dividing the domain by two, as we have seen before has a $O(\log n)$ time complexity, and searching if the closest pair of points is formed by one point from the left side and another from the right side has a complexity of $O(n)$, that because the points are previously sorted, making the algorithm to run in $(n \log n)$ time.

Time Complexity: $O(n \log n)$
Input:

An integer n indicating the number of (x, y) coordinates. Then n lines follow, each describing a (x, y) coordinate.
Output:

The distance between the closes pair of points. If that distance is bigger than MAX, then it will print *"INFINITY"*.

Listing 5.3: Closest Pair of Points

```
#include <cstdio>
#include <cmath>
#include <algorithm>
#define N 10001
#define MAX 9999.99999
using namespace std;

class Point {
 public:
  double x;
  double y;

  Point(double x = 0.0, double y = 0.0) {
    this->x = x;
    this->y = y;
  }

  bool operator<(const Point &b) const {
    if (this->x < b.x) {
      return true;
    } else if (this->x > b.x) {
      return false;
    } else {
      if (this->y < b.y) {
        return true;
      } else if (this->y > b.y) {
        return false;
      }
    }
    return false;
  }
};

Point point[N];

double closestPair(long, long);
```

```
37 double distance(Point, Point);
38
39 int main() {
40   long i, n;
41   double d;
42
43   scanf("%ld", &n);
44   for (i = 0; i < n; i++) {
45     scanf("%lf %lf", &point[i].x, &point[i].y);
46   }
47
48   sort(point, point + n);
49   d = closestPair(0, n - 1);
50
51   if (d > MAX) {
52     printf("INFINITY\n");
53   } else {
54     printf("%.4lf\n", d);
55   }
56
57   return 0;
58 }
```

The function `closestPair` receives two integers a and b, and return the distance of the closest pair of points considering the points with indexes in the interval $[a, b]$. If there is only one point, the distance is *infinite*, if there are two points it returns the Euclidean distance between them, and if there are more than two points it separates the points in two halves and repeat the process recursively for each half, finally it searches for a closer pair of points taking one point from each half.

```
1  double closestPair(long a, long b) {
2    long i, j, k;
3    double d1, d2, d;
4    double xp;
5
6    if (a == b) {
7      return MAX + 1.0;
8    }
9
10   else if (b - a == 1) {
11     return distance(point[b], point[a]);
12   } else {
13     d1 = closestPair(a, (a + b) / 2);
14     d2 = closestPair((a + b) / 2 + 1, b);
15     d = min(d1, d2);
16
17     j = (a + b) / 2;
18     xp = point[j].x;
19
20     do {
21       k = (a + b) / 2 + 1;
22       while (xp - point[k].x < d && k <= b) {
23         d1 = Distance(point[k], point[j]);
24         d = min(d, d1);
25         k++;
26       }
```

```
27
28        j--;
29      } while (xp - point[j].x < d && j >= a);
30
31      return d;
32    }
33  }
```

The `distance` function receives two points and returns the Euclidean distance between them.

```
double distance(Point p1, Point p2) {
  double d =
      sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
  if (d > MAX) {
    d = MAX + 1.0;
  }
  return d;
}
```

5.4 Polynomial Multiplication (FFT)

A polynomial of degree bound n has the following form:

$$a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$

This can be expressed as an array of coefficients a, where

$$a = [a_{n-1}, a_{n-2}, \ldots, a_1, a_0]$$

In the same way, a polynomial b of degree bound m, can be expressed as

$$b = [b_{m-1}, b_{m-2}, \ldots, b_1, b_0]$$

The sum and difference of two polynomials is done in linear time, but the multiplication is done in $O(nm)$ time, and the resulting polynomial will have a degree bound of $n + m$. If both polynomials have the same degree n, the resulting polynomial would have a degree bound of $2n$. This multiplication can be expensive if n is large. A Fast Fourier Transform (FFT) will allow us to do this multiplication in $O(n \log n)$ time.

First we need to generate n points, $w_n^0, w_n^1, \cdots, w_n^{n-1}$, where n is a power of 2. Such points have the form $e^{2\pi ik/n}$ for $k = 0, 1, \ldots, n-1$. To interpret this formula, we use the definition of the exponential of a complex number:

$$e^{iu} = cos(u) + isin(u)$$

The cancellation lemma tell us that

$$w_{dn}^{dk} = w_n^k$$

Also we know that

$$w_n^0 = 1$$
$$w_n^{n/2} = -1$$

The Halving lemma says that

$$\left(w_n^{k+n/2}\right)^2 = \left(w_n^k\right)^2$$

Since $w_n^{n/2} = -1$, then $w_n^{k+n/2} = -w_n^k$

The DFT

Recall we want to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree bound n at $w_n^0, w_n^1, \cdots, w_n^{n-1}$. Let us define the results y_k, for $k = 0, 1, \ldots, n-1$ by

$$\begin{aligned} y_k &= A(w_n^k) \\ &= \sum_{j=0}^{n-1} a_j w_n^{kj} \end{aligned} \tag{5.2}$$

The vector $y = [y_0, y1, \cdots, y_{n-1}]$ is the **discrete Fourier transform (DFT)** of the coefficient vector $a = [a_{n-1}, a_{n-2}, \ldots, a_1, a_0]$. We also write $y = DFT_n(a)$.

The FFT

By using the **fast Fourier transform (FFT)**, we can compute the $DFT_n(a)$ in time $O(n \log n)$. We assume that n is an exact power of 2.

The FFT employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of $A(x)$ to define two new polynomials of degree bound $n/2$

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

then

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{5.3}$$

so that the problem of evaluating $A(x)$ at $w_n^0, w_n^1, \cdots, w_n^{n-1}$ reduces to

1. Evaluating the degree-bound $n/2$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ at the points points

$$(w_n^0)^2, (w_n^1)^2, \cdots, (w_n^{n-1})^2 \tag{5.4}$$

2. combining the results according to equation (5.3).

By the halving lemma, the list of values (5.4) consists not of n distinct values but only $n/2$ complex roots of unity, which each root occurring exactly twice.

Algorithm 2 $FFT(a)$

```
n = a.length
if n == 1 then
  return a
end if
w_n = e^{2πi/n}
w = 1
a^[0] = (a_0, a_2, ··· , a_{n-2})
a^[1] = (a_1, a_3, ··· , a_{n-1})
y^[0] = FFT(a^[0])
y^[1] = FFT(a^[1])
for k = 0 to n/2 − 1 do
  y_k = y_k^[0] + w y_k^[1]
  y_{k+n/2} = y_k^[0] − w y_k^[1]
  w = w w_n
end for
return y
```

Now that we complete the polynomial multiplication we must convert from point-value form back to coefficient form. To accomplish this we must compute the inverse FFT (FFT^{-1}), where:

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k w_n^{-kj} \tag{5.5}$$

By comparing equations 5.2 and 5.5, we see that by modifying the FFT algorithm to switch the roles of a and y, replace w_n by w_n^{-1}, and divide each element of the result by n. Then we define the multiplication of two polynomials of length n, where n is a power of 2, as:

$$a \times b = FFT^{-1}(FFT(a) \times FFT(b)) \tag{5.6}$$

Appendix D.1 contains the source code of a polynomial multiplication using the FFT.

5.5 Range Minimum Query (RMQ)

Given an array X of n elements and two positions or indices of the array, the RMQ finds the minimum element in X between those two positions. The algorithm is commonly used to handle

multiple queries, since using a linear search for every query represents a high cost, the *RMQ* builds a table M in $O(n \log n)$ time, and using that table each query can be answered in $O(1)$ time.

Each row of M represents a starting position in the array, and each column represents a power of two, in such a way that $M_{i,j}$ contains the index of the minimum element between elements $X[i], X[i+1], \ldots, X[i+2^j-1]$.

The first thing to do is initialize the table M by filling its first column, which is column zero, where $M_{i,0} = i$, for every $i = 0, \ldots n-1$. The next step is to fill column one, then column two, and so on. Each element in the table is obtained by using 5.7, which stores in $M_{i,j}$ the index of the minimum element between two sub-arrays, one formed by elements $X[i], \ldots, X[i+2^{j-1}-1]$, and the other by $X[i+2^{j-1}], \ldots, X[i+2^j-1]$.

$$M_{i,j} = \arg\min \left(X[M_{i,j-1}], X[M_{i+2^{j-1},j-1}] \right) \tag{5.7}$$

Table 5.1 shows the table M for the array $X = [2, 4, 3, 1, 6, 7, 8, 9, 1, 7]$, that as we can see has dimensions of $n \times \log n$. Another important thing to mention is that as we start filling the columns, more elements are left with an empty value, since the intervals they represent are out of range, in fact the k^{th} column will contain only $n - 2^k + 1$ non-empty values. Meaning that filling the whole table takes $O(n \log n)$ time.

0	0	3	3
1	2	3	3
2	3	3	3
3	3	3	-
4	4	4	-
5	5	8	-
6	6	8	-
7	8	-	-
8	8	-	-
9	-	-	-

Table 5.1: Table built using the *RMQ* algorithm

To answer a query we just need to check if the range can be covered with one single power of two step, if it is, we just need

to get the value directly from the table, otherwise, if the range cannot be covered by a power of two step, we can divide it in two intervals and return the minimum of the two intervals. For example, for the previous array, if we want to know what is the minimum value in X between positions 2 and 6, we face with the problem that we cannot obtain it directly from the table since that range is not covered in the table, but we can return the minimum value between $M_{2,2}$ (which covers positions 2,3,4,5), and $M_{3,2}$ (which covers positions 3,4,5,6), those intervals overlap, but that doesn't affect the result. The code in 5.4 fills table M given an array X of n elements. On the other hand, code in 5.5 prints the index of the minimum element in X between position i and position j.

Time Complexity: $O(n \log n)$
Input:
`n`. The number of elements in the array.
`X`. Array of n elements.
Output:
Constructs the table M used to find the minimum value in X between two positions.

Listing 5.4: RMQ (Fill the Table)

```
// initialize M for the intervals with length 1
for (i = 0; i < n; i++) {
  M[i][0] = i;
}

// compute values from smaller to bigger intervals
for (j = 1; 1 << j <= n; j++) {
  for (i = 0; i + (1 << j) - 1 < n; i++) {
    if (X[M[i][j - 1]] < X[M[i + (1 << (j - 1))][j - 1]]) {
      M[i][j] = M[i][j - 1];
    } else {
      M[i][j] = M[i + (1 << (j - 1))][j - 1];
    }
  }
}
```

Time Complexity: $O(1)$
Input:
`X`. Array of n elements
`i,j`. Two numbers where $i \leq j < n$.
Output:
The index of the minimum value in the sub-array $X_i, \ldots, X_j$.

Listing 5.5: RMQ (Answer a Query)

```
ans = 0;
k = (long)floor(log(double(j - i + 1)) / log(2.0));
if (X[M[i][k]] <= X[M[j - (1 << k) + 1][k]]) {
  ans = M[i][k];
} else {
  ans = M[j - (1 << k) + 1][k];
}
printf("%d\n", ans);
```

There is a more detailed explanation and implementation of the RMQ algorithm in the *topcoder* forum [11], where they also mention some applications of the algorithm, specially to solve the *Lowest Common Ancestor* problem.

5.6 Chapter Notes

The goal of the *Divide and Conquer* technique is to divide a problem in smaller and easier sub-problems, the sub-problems can't be harder to solve than the original problem.

Lee, Tseng, Chang, and Tsai [12] explain some problems solved by the Divide and Conquer technique, among them, it is the *Closest Pair of Points* problem, basically they give the steps that we follow to code the solution presented in this chapter. Cormen, Leiserson, Rivest, and Stein [1] give a great analysis of the *Fast Fourier Transform* applied to polynomial multiplication, and also includes an introduction to Divide and Conquer, and give some rules about the time performances in certain cases.

In appendix D you can find problems solved using the *Divide and Conquer* technique, including the implementation of the polynomial multiplication algorithm using the FFT described in this chapter.

5.7 Exercises

1. Given an array X of n elements, $(2 \leq n \leq 10000)$, and q queries, $(1 \leq q \leq 100000)$, where each of those queries contains two numbers a and b, $(a < b < n)$. Describe an efficient algorithm that calculates the sum $X_a + X_{a+1} + \cdots + X_b$ for each of the queries.

2. Consider a board of size $n \times m$ which hides a prize in one of its cells. At the beginning all cells are covered and is impossible see what is inside. The game consists on the following: In each turn the contestant choose a cell, and the content of that cell is reveled. If it contains the prize the game is over and the contestant can take the prize. If the prize is not there, then the host of the game tells the contestant if the prize is on the "left" or "right" of the selected cell, also tells if the prize is "up" or "down". The problem is that the contestant only have k opportunities to find the prize. How can we determine if the contestant can win the prize given the dimensions of the board and the opportunities the contestant has?

3. Given an array X of n integers, $(2 \leq n \leq 100000)$,where $X_i \leq X_{i+1}$. In addition to that, q queries are given, $(1 \leq q \leq 100000)$. Each query contains two numbers a and b, and the goal is to find the number of occurrences of the most frequent number among $X_a, \ldots, X_b$. Write a program that solves that problem in an efficient way.

6

Dynamic Programming

"It does not matter how slowly you go as long as you do not stop."

– Confucius

Dynamic Programming (DP) is one of the most enjoyable areas in the field of algorithms, because it involves a lot of thinking and develops the creativity. Is not rare to face a problem that requires to think on a solution for hours, days, weeks, o more, and all to end with an implementation of just twenty lines of code or less.

DP can be defined as a tool that uses previously calculated values to obtain a new value. In other words, it uses what is already know in time t to answer a question in time $t+1$. Most of the DP problems have these two following properties:

1. A recursive function. A way to express a new value using previously obtained values.
2. Memory usage. Is common the use of arrays and multidimensional arrays to store the information needed to compute a new value.

A simple example of a DP problem is to obtain the n^{th} Fibonacci number . Remember that the Fibonacci sequence F start with $F_0 = 1$ and $F_1 = 1$, and F_k is the sum of the two previous elements in the sequence, then we have that

$$F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

So, given a number n the objective is to obtain the value of F_n. The greatest challenge when trying to find a DP solution for a problem is to find the recursive function, sometimes is easy to see it, but sometimes it isn't, for this case is quite easy and it is

$$F_n = F_{n-1} + F_{n-2}. \tag{6.1}$$

The next thing is to use memory to store the values that are needed to calculate a new value. In this step the programmer has different options. One option is to use two variables and keep updating them trough all the process until the n^{th} Fibonacci number is obtained. Another option is to use an array of size $n + 1$, where the value on position k in the array corresponds to the k^{th} Fibonacci number.

There is no easy way to learn how to use DP, it more like a habit that must be acquired trough practice and solving new problems, and gradually it becomes easier to identify when is a good idea to implement a DP solution. The goal for this chapter then is to develop that habit, and improve the ability of the reader to think outside the box and be able to identify when a DP solution is needed.

6.1 Longest Increasing Sub-sequence (LIS)

Given a sequence X of n integers, the objective is to find the longest sub-sequence, $X_{k_1}, X_{k_2}, \ldots, X_{k_m}$, such that $k_i > k_{i-1}$ and $X_{k_i} > X_{k_{i-1}}$. For example, for the following sequence:

$$3, 8, 2, 7, 3, 9, 12, 4, 1, 6, 10,$$

the longest increasing sub-sequence would be:

$$2, 3, 4, 6, 10.$$

The idea of the algorithm is to keep an array L where L_i represents the length of a *LIS* with X_i as its final element. First start L_i with 1, then for every j from 0 to $i - 1$, check if $X_j < X_i$ and $L_j + 1 > L_i$, if that happens then make $L_i = L_j + 1$. What we are doing here is to check if we can add the element X_i to the *LIS* that ends in X_j, if we can, then check if that *LIS* is longer than

the one we already have, and keep the longest one. The length of the *LIS* of the whole sequence will be the maximum value in L.

For the sequence above, the array L will look like this.

X =	3	8	2	7	3	9	12	4	1	6	10
L =	1	2	1	2	2	3	4	3	1	4	5

Table 6.1: The value of L_i represents the length of the *LIS* ending with X_i.

Since for every element we have to go trough for all its previous elements. The number of operations is $n(n-1)/2$. So the time complexity for this algorithm is $O(n^2)$.

To keep track which elements are part of the *LIS*, every time that $X_j < X_i$ and $L_j + 1 > L_i$ we say that element j precedes element i. In that way we only need the last element of the *LIS* and then move backwards until reach the first element to obtain the whole sequence. The code in 6.1 implement the *LIS* algorithm for an array of n elements.

Time Complexity: $O(n^2)$
Input:
x: Vector of integers
n: Number of elements in x
Output:
The length of the LIS and the elements in the LIS.

Listing 6.1: Longest Increasing Sub-sequence

```
1  void LIS() {
2    int i, j;
3    int max, pos;
4
5    memset(Prev, -1, sizeof(Prev));
6    L[0] = 1;
7    max = L[0];
8    pos = 0;
9
10   for (i = 1; i < n; i++) {
11     L[i] = 1;
12     for (j = 0; j < i; j++) {
13       if (X[j] < X[i] && (L[j] + 1) > L[i]) {
14         L[i] = L[j] + 1;
15         Prev[i] = j;
16       }
17     }
18
```

```
    if (L[i] > max) {
      max = L[i];
      pos = i;
    }
  }

  printf("LIS length: %d\n", max);
  printLIS(pos);
}

void printLIS(int pos) {
  if (Prev[pos] != -1) {
    printLIS(Prev[pos]);
  }
  printf("%d\n", X[pos]);
}
```

6.1.1 Longest Increasing Subsequence with Binary Search

From the previous *LIS* algorithm, we can notice that the second cycle makes the things to run quite slow, because to obtain the value of L_i we need to walk trough all the previous elements. This makes the previous code useless when n is large.

Given that we are dealing with an increasing sub-sequence, we can use a binary search to speed things up, and make this step in logarithmic time. The idea is to keep an array of positions P, where P_i represents the position of the last element of a *LIS* with length $i+1$. If there are more than one element, keep the smallest element. Consider the following array:

$$X = [3, 8, 2, 7, 3, 9]$$

For each number in the array we do the following:

1. $X_0 = 3$. Insert the first element.

$$P = [0]$$
$$X_P = [3]$$

2. $X_1 = 8$. Is greater than the last element in X_P, so add it

$$P = [0, 1]$$
$$X_P = [3, 8]$$

3. $X_2 = 2$. Is not greater than the last element, so find the first element that is greater using binary search. In this case is the 3, and because X_2 is smaller, then replace it.

$$P = [2, 1]$$
$$X_P = [2, 8]$$

4. $X_3 = 7$. The first element that is greater is 8, then replace it.

$$P = [2, 3]$$
$$X_P = [2, 7]$$

5. $X_4 = 3$. The first element that is greater is 7, then replace it.

$$P = [2, 4]$$
$$X_P = [2, 3]$$

6. $X_5 = 9$. Is greater than the last element, then add it.

$$P = [2, 4, 5]$$
$$X_P = [2, 3, 9]$$

The array X_P represents the elements of X in the positions indicated by P. The array $P = [2, 4, 5]$ tells us that there is *LIS* of length 1 (2) that ends with element X_2. There is *LIS* of length 2 $(2, 3)$ that ends with element X_4. And there is a *LIS* of length 3 $(2, 3, 9)$ with X_5 as its last element. The program in 6.2 implements a *LIS* with a binary search over an array X of n elements.

Time Complexity: $O(n \log n)$
Input:
x: Vector of integers
n: Number of elements in x
Output:
The length of the LIS and the elements in the LIS.

Listing 6.2: Longest Increasing Sub-sequence $O(n \log n)$

```
1 void LIS() {
2   int a, b, c;
3   int tail;
```

```
4
5   memset(Prev, -1, sizeof(Prev));
6   P[0] = 0;
7   tail = 0;
8
9   for (int i = 1; i < n; i++) {
10    if (X[i] > X[P[tail]]) {
11      Prev[i] = P[tail];
12      P[++tail] = i;
13      continue;
14    }
15
16    for (a = 0, b = tail; a < b;) {
17      c = (a + b) / 2;
18      if (X[P[c]] < X[i]) {
19        a = c + 1;
20      } else {
21        b = c;
22      }
23    }
24
25    if (X[i] < X[P[a]]) {
26      if (a > 0) {
27        Prev[i] = P[a - 1];
28      }
29      P[a] = i;
30    }
31  }
32
33  printf("LIS length: %d\n", tail + 1);
34  printLIS(P[tail]);
35 }
36
37 void printLIS(int pos) {
38   if (Prev[pos] != -1) {
39     printLIS(Prev[pos]);
40   }
41   printf("%d\n", X[pos]);
42 }
```

6.2 Longest Common Sub-sequence (LCS)

A classic dynamic programming problem that consists on finding the length of the longest common sub-sequence of two sequences X and Y of size n and m respectively. A sub-sequence is a sequence that can be obtained from another sequence by removing some of its elements and preserving the order of the remaining elements. The LCS of two sequences is a sub-sequence that is common to both the sequences and has a maximal length, e. g. Consider the following two strings:

$$X = mexico$$
$$Y = america,$$

their *LCS* is *meic*.

The algorithm to find the *LCS* of two strings consists on having a matrix C of $n \times m$, where $C_{i,j}$ represents the length of the *LCS* using the first i letters from X and the first j letters from Y. So the result will be stored in position $C_{n,m}$

For the case where X_i is equal to Y_j that means that adding letter X_i and Y_j increments the length of the *LCS* by one, $C_{i,j} = C_{i-1,j-1} + 1$.

And for the case where X_i and Y_j are different, we only need to keep the greatest value in C so far, $C_{i,j} = \max(C_{i-1,j}, C_{i,j-1})$.

The matrix C for the example above would look like this.

		a	m	e	r	i	c	a
	0	0	0	0	0	0	0	0
m	0	0	1	1	1	1	1	1
e	0	0	1	2	2	2	2	2
x	0	0	1	2	2	2	2	2
i	0	0	1	2	2	3	3	3
c	0	0	1	2	2	3	4	4
o	0	0	1	2	2	3	4	4

Figure 6.1: The value of C_{ij} represents the length of the *LCS* considering the first i characters of the word "mexico"and the first j characters of the word "america".

Like in the *LIS* algorithm, to know the elements of the *LCS* we just need to keep track of the location where each value comes from. For this case it can come from the element in the left, top, and top-left. The code in 6.3 returns the length of the *LCS* of two strings X and Y with n and m characters respectively.

Time Complexity: $O(nm)$
Input:
The variables X, Y, n and m are declared as global.
Output:
The length of the LCS

Listing 6.3: Longest Common Sub-sequence (LCS)

```
int LCS() {
  memset(C, 0, sizeof(C));

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      if (X[i - 1] == Y[j - 1]) {
        C[i][j] = C[i - 1][j - 1] + 1;
      } else {
        C[i][j] = max(C[i - 1][j], C[i][j - 1])
      }
    }
  }

  return C[n][m];
}
```

In case we want to print the *LCS* and not just its length, we can use the same matrix C defined in the code 6.3 to find such sub-sequence. The code in 6.4 just print one *LCS*, but there can be more than one *LCS* with the same length.

Listing 6.4: Printing of the LCS

```
void printLCS(int i, int j) {
  if (i == 0 || j == 0) {
    return;
  }

  if (C[i][j] == C[i - 1][j - 1] + 1) {
    printLCS(i - 1, j - 1);
    printf("%d ", X[i - 1]);
  } else if (C[i][j] == C[i - 1][j]) {
    printLCS(i - 1, j);
  } else {
    printLCS(i, j - 1);
  }
}
```

6.3 Levenshtein Distance (Edit Distance)

Named after Vladimir Levenshtein is a metric for measuring the difference between two sequences. Given two strings of characters *str1* and *str2*, Levenshtein distance is the minimum number of steps needed to transform *str1* into *str2* using three operations.

- **Insertion.** Insert one character in *str1*.
- **Deletion.** Remove one character from *str1*.
- **Replace.** Change one character of *str1* with another.

Suppose we want to change the string "LOVE" to "ALONE". Two operations would be needed.

1. Insert A in pisition 0.
2. Replace V witn N.

The algorithm is similar to the one used to find the *Longest Common Sub-sequence*. It is also based on a matrix C, where $C_{i,j}$ represents the minimum number of steps to transform the first i characters of *str1* into the first j characters of *str2*. That means that the result will be stored in $C_{n,m}$.

The value of $C_{i,j}$ is given by

$$C_{i,j} = \min\left(C_{i-1,j-1} + k, \min\left(C_{i,j-1} + 1, C_{i-1,j} + 1\right)\right),$$

where k is 1 if $str1_i \neq str2_j$, otherwise is 0.

Is also important to initialize the matrix C in the following way

$$\begin{aligned} C_{0,0} &= 0 \\ C_{0,i} &= i,\ i = 1, \ldots m \\ C_{i,0} &= i,\ i = 1, \ldots n \end{aligned}$$

If $str1_i = str2_j$ it doesn't represent a cost and $C_{i,j}$ would be equal to $C_{i-1,j-1}$. In case that $str1_i \neq str2_j$ it means that we may

need to replace the i^{th} character of $str1$ with $str2_j$ which would we give us a cost of $C_{i-1,j-1}+1$. In any case we still need to check if it is more convenient to remove character str_i that will have a cost of $C_{i-1,j}+1$, or insert character $str2_j$ with a cost of $C_{i,j-1}+1$.

To print the operations we just need to keep track of where the value in $C_{i,j}$ comes from. Starting from the element $C_{n,m}$ we need to move backwards, if $C_{i,j} = C_{i-1,j-1} + 1$ then is a *replace* operation, if $C_{i,j} = C_{i-1,j} + 1$ is a *remove* operation, and if $C_{i,j} = C_{i,j-1} + 1$ is an *insert* operation. The code in 6.5 reads two strings str_1 and str_2, and prints the minimum number of operations to turn str_1 into str_2, and the operations in the order that they must be executed.

Time Complexity: $O(nm)$, where n and m are the length of $str1$ and $str2$ respectively

Input:

Strings $str1$ and $str2$.

Output:

The Levenshtein distance of strings $str1$ and $str2$ and the steps to transform $str1$ into $str2$

Listing 6.5: Edit Distance

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#define N 100
using namespace std;

char str1[N], str2[N];
int C[N][N];
int n, m, len;

int editDistance();
void printEditDistance(int, int);

int main() {
  scanf("%s", str1);
  scanf("%s", str2);

  len = 0;
  n = strlen(str1);
  m = strlen(str2);

  printf("%d\n", editDistance());
  printEditDistance(n, m);
  return 0;
}
```

Following the method described above the function `editDistance` computes the minimum number of operations to transform `str1` into `str2`. The value of `C[i][j]` represents the minimum number of steps to transform the sub-string formed by the first i characters of `str1` into the sub-string formed by the fist j characters of `str2`.

```
int editDistance() {
  int k;

  C[0][0] = 0;
  for (int i = 1; i <= n; i++) {
    C[i][0] = i;
  }

  for (int i = 1; i <= m; i++) {
    C[0][i] = i;
  }

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      k = (str1[i - 1] == str2[j - 1]) ? 0 : 1;
      C[i][j] = min(min(C[i - 1][j - 1] + k, C[i - 1][j] + 1), C[i][j - 1] +
          1);
    }
  }

  return C[n][m];
}
```

The function `printEditDistance` prints the operations needed to transform `str1` into `str2` using the values obtained in `editDistance`. For each location (i, j) in the matrix is possible to know which operation was made by checking the value of `C[i][j]` and its neighbors.

```
void printEditDistance(int i, int j) {
  int pos;

  if (i == 0 && j == 0) {
    return;
  }

  if (j > 0 && C[i][j - 1] + 1 == C[i][j]) {
    printEditDistance(i, j - 1);
    len--;
    pos = i - len;
    printf("Insert %d,%c\n", pos, str2[j - 1]);
  } else if (i > 0 && j > 0 && C[i - 1][j - 1] + 1 == C[i][j]) {
    printEditDistance(i - 1, j - 1);
    pos = i - len;
    printf("Replace %d,%c\n", pos, str2[j - 1]);
  } else if (i > 0 && C[i - 1][j] + 1 == C[i][j]) {
    printEditDistance(i - 1, j);
    pos = i - len;
    printf("Delete %d\n", pos);
```

```
    len++;
  } else if (i > 0 && j > 0)
    printEditDistance(i - 1, j - 1);
}
```

6.4 Knapsack Problem

Consider the following problem.

Is Christmas Eve in Mexico and everyone is out celebrating this special day, you are not the exception and you are in you grandparent's house eating tamales. Meanwhile a thief gets into your house with the intention to steal different objects. Each object has a specific value and weight; the thief's knapsack only can carry certain weight. The thief wants to maximize the value of the objects he steal, in other words the thief prefers one object with value of 100 than 99 objects with value of 1. What would be the total maximum value the thief can steal that night? Could you code and algorithm to solve the thief's dilemma?

One approach to this problem using DP consists on using a matrix C, where $C_{i,j}$ represents the maximum value the thief can get considering the first i objects and using a knapsack of capacity j. In that way the result will be stored in $C_{n,m}$, where n is the number of objects and m is the weight capacity of the knapsack.

Each element in the matrix is obtained using the following equation.

$$C_{i,j} = \begin{cases} C_{i-1,j} & W_i > j, \\ \max\left(C_{i-1,j}, C_{i-1,j-W_i} + V_i\right) & W_i \leq j, \end{cases}$$

where W_i and V_i represents the weight and value of object i respectively.

The algorithm stops until the matrix C is filled, making the time complexity of the algorithm equal to the dimensions of C, which is $O(nm)$. The algorithm in 6.6 uses a vector V to store the object values and a vector W to store the object weights and returns the maximum value that can be carried in a knapsack of capacity m.

Time Complexity: $O(nm)$, where n is the number of objects and m is the weight the knapsack can carry
Input:
W: The weight of the objects
P: The value of objects
n: The number of objects
m: The capacity of the knapsack
Output:
The maximum value the thief can steal

Listing 6.6: Knapsack Problem

```
int knapsack() {
  memset(C, 0, sizeof(C));
  for (long i = 1; i <= n; i++) {
    for (long j = 1; j <= m; j++) {
      if (W[i] > j) {
        C[i][j] = C[i - 1][j];
      } else {
        C[i][j] = max(C[i - 1][j], C[i - 1][j - W[i]] + V[i]);
      }
    }
  }

  return C[n][m];
}
```

6.5 Maximum Sum in a Sequence

Given a sequence of numbers, which can be positive or negative, find a sub-sequence of consecutive elements, which its sum is maximal. For example, consider the following sequence of 10 elements.

$$-2, \mathbf{3,-2,4,4}, -8, -5, 8, -7, 1$$

The maximum sum that can be obtained is 9, which corresponds to the sub-sequence: 3,-2,4,4. Any other sub-sequence will have a smaller sum.

This problem can be solved in linear time by just keeping an cumulative sum of all elements, and when that sum is smaller than zero then reset it to zero. Just be careful with the case where all elements are negatives. In that case the answer is the greatest value. The program in 6.7 reads n numbers and returns the maximum sum of consecutive elements.

Time Complexity: $O(n)$
Input:

n: The number of elements in the sequence. Then n integers are given.

Output:

The maximum sum that can be obtained from a sub-sequence of consecutive elements.

Listing 6.7: Maximum Sum

```
#include <algorithm>
#include <cstdio>
#include <iostream>
#define oo 1000000
using namespace std;

int main() {
  long n, num, s, maxValue;

  scanf("%ld", &n);

  maxValue = -oo;
  s = 0;
  for (long i = 0; i < n; i++) {
    scanf("%ld", &num);
    if (s + num > 0) {
      if (num > s + num) {
        s = num;
      } else {
        s += num;
      }
      maxValue = max(maxValue, s);
    } else {
      maxValue = max(maxValue, s + num);
      s = 0;
    }
  }

  printf("The maximum sum is %ld.\n", maxValue);
  return 0;
}
```

6.6 Rectangle of Maximum Sum

Given a $n \times n$ matrix of integers, the goal of this algorithm is to find the sub-matrix whose sum of its elements is maximum.

The solution proposed here runs in $O(n^3)$, but the idea is the same for the *Maximum Sum* problem, to keep an cumulative sum and reset it when it is smaller than zero. Do it for every column for every sub-matrix.

Consider the following matrix.

4	-1	3	-8	2
6	5	-4	7	3
0	-8	1	1	3
9	-5	-7	-4	0
1	-1	-4	4	3

Figure 6.2: The matrix can contain positive and negative numbers

For the matrix in 6.2 the maximum sum in a sub-matrix is 20 that corresponds to the sub-matrix formed by the first column. The solution showed in 6.8 receives a square matrix of size n as input, and prints the maximum sum that can be found inside a sub-matrix.

Time Complexity: $O(n^3)$
Input:
n: The size of the matrix
x: The matrix of integers
Output:
The sum of the elements inside the rectangle of maximum sum

Listing 6.8: Rectangle of Maximum Sum

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#define N 101
#define oo 32767
using namespace std;

int x[N][N];
int u[N];

int main() {
  int n, maxValue;

  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      scanf("%d", &x[i][j]);
      x[i][j] += x[i - 1][j];
    }
  }

  maxValue = -oo;
  for (int i = 0; i < n; i++) {
```

```
    for (int j = i + 1; j <= n; j++) {
      for (int k = 1; k <= n; k++) {
        u[k] = x[j][k] - x[i][k];
        if (u[k - 1] > 0) {
          u[k] += u[k - 1];
        }
        maxValue = max(maxValue, u[k]);
      }
    }
  }

  printf("%d\n", maxValue);
  return 0;
}
```

6.7 Optimal Matrix Multiplication

Given a sequence of n matrices, where the number of rows of matrix i is equal to the number of columns of matrix $i-1$. Our task is to choose the location of open and closed parenthesis in order to minimize the number of multiplications. For example, consider the matrices A, B and C with sizes $5 \times 10, 10 \times 20$ and 20×35 respectively.

If we choose the arrangement $A \times (B \times C)$, the number of multiplications is 8750. On the other hand, if we choose $(A \times B) \times C$ the number of multiplications is 4500, making this a better solution.

We can write the sizes of the matrices in a single vector A, where the size of matrix i has size $A_{i-1} \times A_i$, then the number of operations needed to multiply matrix i with matrix $i+1$ is given by $A_{i-1} \times A_i \times A_{i+1}$.

The algorithm goes like this. Suppose there is a $n \times n$ matrix X, where $X_{i,j}$ represents the minimum number of operations needed to multiply the matrices in the interval $[i, j]$. The idea is to update matrix X in each iteration of the algorithm according to this formula.

$$X_{i,j} = \min\left(X_{i,j}, X_{i,k} + X_{k+1,j} + A_{i-1} \times A_k \times A_j\right) \tag{6.2}$$

The total number of iterations needed is $n-1$. In the first iteration, segments of length 2 will be updated, in the second iteration segments of length 3 will be updated and so on.

$X_{1,2}, X_{2,3}, \ldots, X_{n-1,n}$	Values updated in the iteration 1
$X_{1,3}, X_{2,4}, \ldots, X_{n-2,n}$	Values updated in the iteration 2
$\vdots$	$\vdots$
$X_{1,n}$	Values updated in the iteration n-1

Table 6.2: Iterations of the Optimal Matrix Multiplication problem

The code in 6.9 reads an array A of n elements, $(2 \leq n < 20)$, representing the dimensions of the matrices as explained before, and prints the minimum number of multiplications needed and a representation of how the multiplications should be made.

Time Complexity: $O(n^3)$
Input:
n: The number of matrices
A: The size of the matrices
Output:
The minimum number of multiplications needed and the sequence of matrices with the parenthesis located in the optimal position.

Listing 6.9: Optimal Matrix Multiplication

```
#include <cstdio>
#include <cstring>
#include <iostream>
#define N 20
#define oo 1000000
using namespace std;

int X[N][N], S[N][N];
int A[N];

int matrixMultiplication(int);
void printSequence(int, int);

int main() {
  int n;

  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &A[i], &A[i + 1]);
  }

  printf("%d\n", matrixMultiplication(n));
  printSequence(1, n);
  printf("\n");

  return 0;
}

int matrixMultiplication(int n) {
```

```
  int j, val;

  memset(X, 0, sizeof(X));
  for (int l = 2; l <= n; l++) {
    for (int i = 1; i <= n - l + 1; i++) {
      j = i + l - 1;
      X[i][j] = oo;
      for (int k = i; k <= j; k++) {
        val = X[i][k] + X[k + 1][j] + A[i - 1] * A[k] * A[j];
        if (val < X[i][j]) {
          X[i][j] = val;
          S[i][j] = k;
        }
      }
    }
  }
  return X[1][n];
}

void printSequence(int i, int j) {
  if (i == j) {
    printf("A%d", i);
  } else {
    printf("(");
    printSequence(i, S[i][j]);
    printf(" x ");
    printSequence(S[i][j] + 1, j);
    printf(")");
  }
}
```

6.8 Coin Change Problem

Given a bottle with an infinite amount of coins of different denominations, in how many ways can you pay a certain amount of money, using just the coins of that bottle?

Suppose there are three kinds of coins of 1, 2, 5 cents and we have an infinite amount of them, and we want to know in how many ways we can pay 7 cents. It results that there are 6 ways to pay it.

- Seven 1 cent coins.
- Five 1 cent coins and one 2 cents coin.
- Three 1 cent coins and two 2 cents coins.
- One 1 cents coin and three 2 cents coins.
- Two 1 cent coins and one 5 cents coin.
- One 2 cents coin and one 5 cents coin.

Consider the following problem.

Mexico's currency consists of \$100, \$50, \$20, \$10, and \$5 bills and \$2, \$1, 50c, 20c, 10c and 5c coins. Program 6.10 determines for any given amount, in how many ways that amount may be completed. The input consists of a real number no greater than \$300.00. Such amount will be valid, that is, it will be a multiple of 5c. The Output is a single line consisting of the number of ways in which that amount may be completed.

The solution for this problem is similar to the one for the *Knapsack problem*, but this time the number of ways in which i Mexican pesos can be completed by adding coin k to the currency is given by:

$$X_i = \begin{cases} 1 & i = 0 \\ X_i & i < k \\ X_i + X_{i-k} & i \geq k \end{cases}$$

notice that in the beginning X must be initialized with zeros.

Listing 6.10: Coing Change Problem

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#define N 30001
#define M 11
using namespace std;

long long C[M] = {10000, 5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5};
long long X[N];

int main() {
  long long k, money;
  double num;

  // We can pay 0 dollars in one way
  X[0] = 1;
  for (long long i = 0; i < M; i++) {
    k = C[i];
    for (long long j = k; j < N; j++) {
      X[j] += X[j - k];
    }
  }

  while (scanf("%lf", &num) == 1) {
    money = (long long)(num * 100.00000001);
    if (money == 0) {
      break;
    }
    printf("%lld\n", X[money]);
```

```
30    }
31
32    return 0;
33  }
```

6.9 Chapter Notes

The difficult part in dynamic programming problem is first to realize that it is in fact a dynamic programming problem, and second, to find the recursive formula, because sometimes there is this feeling that a problem has a dynamic programming solution, but is hard to see it. Well the only way to solve this problem is to practice, practice, and practice.

There is a section dedicated to dynamic programming in the book *"Introduction to Algorithms"* [1], there we can find a analytic description of some famous problems. Also we recommend to visit the forums of the different online judges, there we can find information and tricks that don't appear in any book.

Some online judges have their problems divided by category, bellow there are a couple of links containing only dynamic programming problems.

- `http://acm.timus.ru/problemset.aspx?space=1&tag=dynprog`
- `https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=114`

6.10 Exercises

1. How many domino tiles of 2×1 fit in a $2 \times n$ table for some given n?

2. A "bad luck" number is a number that contains a "1" followed by a "3". e.a. 14137, or 3133. Given a number n, write a program that find how many "bad luck" numbers of n digits exist?

3. We have an ice cream shop where we serve three kind of flavors: chocolate, vanilla and strawberry. Our ice cream cones are special, they can be n scoops tall with the constraint that there cannot be two equal flavors together, and that vanilla must be always between chocolate and strawberry. In how many ways can we make a n scoops ice cream cone?

4. Consider a matrix A of $n \times m$ cells, in how many ways can we get to cell $A_{n,m}$ from cell $A_{1,1}$ if we only can move one cell at a time to the right or up?

5. Marie has n coins in her purse, coins can have different values, but she can have more than one coin of the same value. She wants to give **all** her coins to her two children, but in a way that the difference of what they receive be minimal, in that way any of them will be sad. Write a program that find how many money receive each child.

7

Graph Theory

"The secret of getting ahead is getting started."

– Mark Twain

Graph Theory is one of the areas with more applications, from image processing to social networks. A graph is no more than nodes or vertices that can be connected by edges, but what they represent is what makes them so important. For example, in a social network every individual can be seen as a node, and if two individuals are friends, then we can connect the corresponding nodes with an edge. Another example can be the cities in a country, each city can be represented as a node, and the roads connecting two cities represents an edge. Figure 7.1 shows a graph representing a map, where the countries are the nodes and edges indicate that two countries have a border in common.

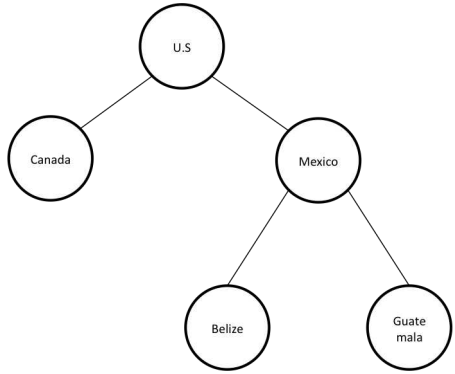


Figure 7.1: smallGraph representing a map, where countries are represented as nodes and edges indicate that two countries share a border.

During this chapter we will see different kind of graphs and learn different algorithms for specific problems, that is why is important to define some concepts about graphs first.

1. **Bidirected Graph.** In this graphs an edge connecting node a with node b, also connects node b with node a. So we can go from a to b and from b to a.

2. **Directed Graph.** Here the edges have a direction, if an edge connects node a with node b, then we can go from a to b, but not the other way around. The edges on this kind of graphs are represented with an arrow indicating the direction.

3. **Weighted Graph.** For this kind of graphs the edges have a weight or cost associated to it. For example, traveling from New York to Boston has some toll cost, well, this can be seen as two nodes (New York and Boston) connected by and edge (road) with a certain cost (toll).

In some cases there can be combinations of different graphs, like a directed and weighted graph, with edges having a direction and a cost. **Along this chapter we will refer to the number of nodes in a graph with letter n, and the number of edges with letter m.** Following we list other concepts that are important to know about.

- **Path.** It's a series of edges that take us from an initial node to a destination node.
- **Cycle.** It's a path where the destination node is the same as the initial node.
- **Degree.** The degree of a node is the number of edges incident to that node.
- **Eulerian Path.** A path that travels across all the edges in the graph only once.
- **Eulerian Cycle.** A cycle that go through all the edges in the graph only once.
- **Hamiltonian Path.** A path that pass through all the nodes in the graph only once.
- **Hamiltonian Cycle.** A cycle that visits all the nodes in the graph only once.
- **Complete Graph.** A graph where all nodes are connected directly. A complete graph contains $n(n-1)/2$ edges exactly.
- **Connected Graph.** In this graphs there is always a path between any pair of nodes. In other words, it is always possible to reach one node from another node in the graph.
- **Disconnected Graph.** A graph that is not connected. Meaning that there is at least one pair of nodes (a, b) such that there is no path between node a and node b.
- **Cut.** A cut is a set of edges that if removed, the vertices are separated in two disjoint sets.
- **Minimum Cut.** Is the cut whose sum of the edge weights is minimal.
- **Directed Acyclic Graph (DAG).** It's a directed graph without cycles.

7.1 Graph Representation

There are different methods to represent a graph in a program, two of the most common are *adjacency matrix* and *adjacency list*, which will be explained in this section using graph in figure 7.2.

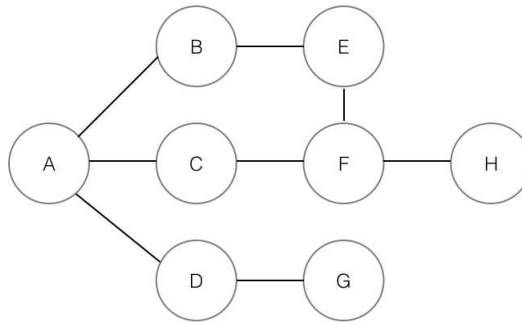


Figure 7.2: Connected graph with 8 nodes and 8 edges.

7.1.1 Adjacency Matrix

Is a boolean matrix A where each row and each column represent a node, and the value stored in the cells indicates if there is an edge between a pair of nodes. If node i and node j are connected by and edge then $A_{i,j} = 1$, otherwise $A_{i,j} = 0$. For bidirectional graphs the adjacency matrix is symmetric. Below is the adjacency matrix for the graph in figure 7.2.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	0	1	1	1	0	0	0	0
B	1	0	0	0	1	0	0	0
C	1	0	0	0	0	1	0	0
D	1	0	0	0	0	0	1	0
E	0	1	0	0	0	1	0	0
F	0	0	1	0	1	0	0	1
G	0	0	0	1	0	0	0	0
H	0	0	0	0	0	1	0	0

Table 7.1: Adjacency Matrix for graph in 7.2

The main advantage of using an adjacency matrix is that finding out if two nodes are connected by an edge is a $O(1)$ operation, because we just need check the value of one specific cell.

When an adjacency matrix contains few non-zero elements we said that it is a sparse matrix. Sparsity in a matrix is a situation

that we want to avoid, since we require a high amount of memory ($O(n^2)$) to represent few edges, since most of the cells are 0's. For this case using an adjacency list would be a better choice.

7.1.2 Adjacency List

Another way to represent a graph consists on having a list for each node in the graph. This list will contain the nodes to which one specific node is connected with. The adjacency list for graph 7.2 is the following:

A:	B, C, D
B:	A, E
C:	A, F
D:	A, G
E:	B, F
F:	C, E, H
G:	D
H:	F

Table 7.2: Adjacency List for graph in 7.2

An adjacency list can be implemented in different ways, it can be an array of linked lists, or a vector of vectors, as long as it uses dynamic memory to store the information. In the book you will commonly see it as a vector of lists.

Adjacency lists are ideal when the number of nodes is greater compared with the number of edges in the graph, avoiding the problem of sparsity in the adjacency matrix.

To find out if node i and node k are connected, we must traverse the whole list of node i to check if node j is contained in it. This is a $O(m)$ operation, where m is the size of the list of node i.

7.2 Graph Traversal

Two of the most common methods for traversing a graph are the *Depth First Search (DFS)* and the *Breadth First Search (BFS)*. The first one uses a stack in its implementation, and the other one uses a queue. Both algorithms need a starting vertex and on each iteration the element a the front/top is removed

and the vertices adjacent to it that have not been added before are inserted into the data structure. The process continues until all the vertices have been explored and the stack/queue is empty.

The difference between these two methods is that DFS explores one branch of the graph until it cannot advance anymore and then returns and try to explore another branch. On the other hand, BFS explores the graph by levels, first the initial vertex, then the vertices at a distance of 1 from the initial vertex, then the vertices at a distance of 2 from the initial vertex, and so on. Let's define the stack for the *DFS* as S and the queue for the *BFS* as Q, and using the graph in 7.2 with vertex A as the initial vertex, the graph traversal using both methods is presented in table 7.3. For this case the nodes are inserted into the data structure used in lexicographical order.

DFS (stack)	**BFS (queue)**
$S = [A]$	$Q = [A]$
$S = [D, C, B]$	$Q = [B, C, D]$
$S = [G, C, B]$	$Q = [C, D, E]$
$S = [C, B]$	$Q = [D, E, F]$
$S = [F, B]$	$Q = [E, F, G]$
$S = [H, E, B]$	$Q = [F, G]$
$S = [E, B]$	$Q = [G, H]$
$S = [B]$	$Q = [H]$
$S = []$	$Q = []$

Table 7.3: Graph Traversal for *DFS* and *BFS*

The way in which the traversal is made is different in both algorithms, the nodes in red are the ones that are been explored. For the *DFS* the order on which the vertexes are visited is *A,D,G,C,F,H,E,B*. On the other hand, for the *BFS* the order is *A,B,C,D,E,F,G,H*. So depending on the problem it will be more convenient to use one method then the other, but both of them has the same time complexity.

7.2.1 DFS

The program in 7.1 implements a *DFS* starting from node 0 on a graph G with nodes numbered from 0 to $n-1$, and prints the nodes

as they are visited. The time complexity of the implementation is $O(n^2)$, but if we use an adjacency list instead of an adjacency matrix it runs in $O(n + m)$ time.

Time Complexity: $O(n^2)$
Input:

Two numbers n $(1 \leq n \leq 100)$,and m $(1 \leq m \leq n(n-1)/2)$, indicating the number of vertices and edges respectively. Then follows m lines, each with two numbers a and b, indicating that there is an edge that connects vertex a with vertex b, and in the other way around.

Output:

The nodes that were found by the *DFS* in the order they were visited.

Listing 7.1: DFS

```
#include <algorithm>
#include <cstdio>
#include <stack>
#define N 101
using namespace std;

stack<int> S;
int G[N][N]; // Adjacency matrix
int V[N];    // Visited nodes
int n, m;    // Number of nodes and edges

void DFS(int);

int main() {
  int a, b;

  scanf("%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    scanf("%d %d", &a, &b);
    G[a][b] = G[b][a] = 1;
  }

  DFS(0); // Start a DFS from node 0
  return 0;
}

void DFS(int a) {
  int v;

  V[a] = 1;
  S.push(a);
  while (!S.empty()) {
    v = S.top(); // Get the elemet in the top
    S.pop();     // Remove the top element

    printf("%d\n", v);

    for (int i = 0; i < n; i++) {
      if (G[v][i] == 1 && V[i] == 0) {
```

```
40          V[i] = 1;
41          S.push(i); // Add ajacent node to the top
42        }
43      }
44    }
45  }
```

7.2.2 BFS

The code in 7.2 implements a *BFS*, and as we can notice it is almost equal to the code in 7.1, but instead of using the library `stack` we use the library `queue`, and instead of using a stack S we use a queue Q. This will cause that every time we execute `Q.push(a)` the node a will be added at the end to the queue. The input of the program is a graph $G = \{V, E\}$ and prints the nodes as they are visited using node 0 as the initial node.

The time complexity is the same as the *DFS*, using an adjacency matrix we obtain more simplicity but also we get a time complexity of $O(n^2)$, which is worst than the $O(n + m)$ we get by using an adjacency list instead.

Time Complexity: $O(n^2)$
Input:
Two numbers n $(1 \leq n \leq 100)$, and m $(1 \leq m \leq n(n-1)/2)$, indicating the number of vertices and edges respectively. Then follows m lines, each with two numbers a and b, indicating that there is an edge that connects vertex a with vertex b, and in the other way around.
Output:
The nodes that were found by the *BFS* in the order they were visited.

Listing 7.2: BFS

```
1  #include <algorithm>
2  #include <cstdio>
3  #include <queue>
4  #define N 101
5  using namespace std;
6
7  queue<int> Q;
8  int G[N][N]; // Adjacency matrix
9  int V[N];    // Visited nodes
10 int n, m;    // Number of nodes and edges
11
12 void BFS(int);
13
```

```
int main() {
  int a, b;

  scanf("%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    scanf("%d %d", &a, &b);
    G[a][b] = G[b][a] = 1;
  }

  BFS(0); // Start a BFS from node 0
  return 0;
}

void BFS(int a) {
  int v;

  V[a] = 1;
  Q.push(a);
  while (!Q.empty()) {
    v = Q.front(); // Get the elemet in the top
    Q.pop();       // Remove the top element

    printf("%d\n", v);

    for (int i = 0; i < n; i++) {
      if (G[v][i] == 1 && V[i] == 0) {
        V[i] = 1;
        Q.push(i); // Add ajacent node to the bottom
      }
    }
  }
}
```

7.2.3 Topological Sort

Topological Sort is an application of DFS. Consider a series of tasks that must be accomplished in certain order, for example Suppose there are four tasks: A, B, C, D. and A must be accomplished before tasks C and D, and task D must be accomplished before task B. You need to find a proper order to finish all tasks without breaking any rule. One possible solution is $ADBC$. Meaning that first we finish task A, then move to task D, then to task B, and finally task C. That solution doesn't break any constraint. Other solutions are: $ACDB$, and $ADCB$.

These rules can be seen as a directed graph, in case that task A comes before of task B, there is an edge that goes from node A to node B. With the graph representation it is possible to use any of the traversal methods seen so far, but for the case of the Topological Sort a *DFS* is needed, and the solution is obtained by adding into a stack the visited nodes until all their respective

adjacent nodes have been explored.

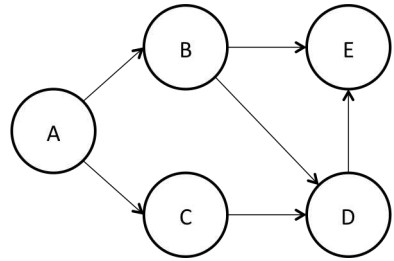


Figure 7.3: Directed graph indicating the order in which a set of tasks must be executed. Task A must be executed before task B and task C. Task D must be executed after tasks B and C, and task E must be executed after tasks B and D.

In the graph 7.3 a possible *DFS* traversal can be $ACDEB$, and if each node is added into a stack once all their adjacent nodes have been explored that stack will look like this: $ABCDE$, which is a solution for the Topological Sort problem.

The time complexity to find a topological sort in a directed graph is the same for the *DFS*. The program in 7.3 shows a recursive implementation of *DFS* and stores in stack S a valid topological sort for a given graph G.

Time Complexity: $O(n^2)$
Input:
n: Amount of nodes in the graph. $(1 \leq n \leq 100)$.
G: Adjacency matrix of the graph.
Output:
S: Stack with a valid *Topological Sort*.

Listing 7.3: Topological Sort

```
#include <stack>
#define N 101
using namespace std;

stack<int> S; // Topological sort
int V[N];     // Visited nodes
int G[N][N]; // Adjacency matrix
int n;        // Number of nodes

```

```
10  void Topological_sort() {
11    for (int i = 0; i < n; i++) {
12      if (V[i] == 0) {
13        DFS(i);
14      }
15    }
16  }
17
18  void DFS(int k) {
19    V[k] = 1;
20    for (int i = 0; i < n; i++) {
21      if (G[k][i] == 1 && V[i] == 0) {
22        DFS(i);
23      }
24    }
25    S.push(k);
26  }
```

7.3 Disjoint Sets

For some problems we need to join two sets into a single one and find if a certain element is part of a certain set. If the number of instructions is large, an ordinary graph traversal such as DFS or BFS would be too expensive, so we need something faster. One of the most popular algorithms is the Union-Find algorithm that will be described ahead. Union-Find is considered a data structure as well. Basically the structure itself is the solution for the problem that poses if two nodes are in the same connected component. It is a quite simple and elegant structure.

Many of the discoveries in the disjoint-set data structures are due to Robert E. Tarjan [13], whom in 1975 found the upper bound of the time complexity for the operations on any disjoint set data structure satisfying certain conditions.

7.3.1 Union-Find

As its name indicates, this algorithm consists on two main steps.

1. **Union.** Join two sets into a single one.
2. **Find.** Finds the set to which a given element belongs to.

At the beginning every element is in a different set. A set can be seen as a tree initially with zero height and with just one node, which is the root. To connect element a with element b, we first

must find the set of each element, here enters the *Find* operation. Finding the set is equal to finding the root of the tree, if the root is the same for the two elements, then they are already part of the same set. If the root is different then we do the *Union* operation, which consists of merging both trees in a single one. For this we ask for the height of each root and make the root with smaller height child of the root with larger height. For the case where both roots have the same height any of them can be child of the other, just remember to increase the height of the root selected as parent.

Consider a graph with 5 nodes and no connections, like the one showed in figure 7.4. Each one of these nodes have a reference to its parent node, which at the beginning is the node itself, and all of them have zero height.

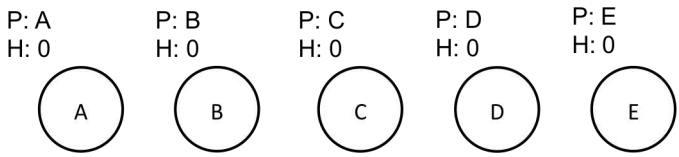


Figure 7.4: At first the parent of each node is the node itself, and all nodes have zero height.

If we add a connection between node A and node D, since both of them have zero height any of them can be the root, for this case A will be the root and it will have a height of 1. If then we connect node D and node E, since the root of D is A, and A has a larger height than node E, then A will be the parent of E. See figure 7.5.

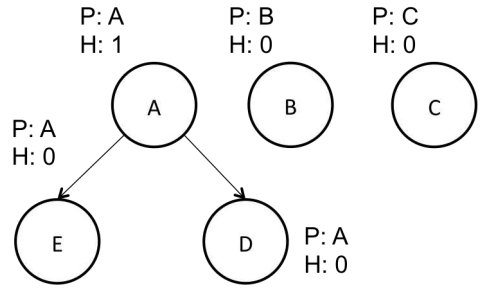


Figure 7.5: Graph resulting from connecting nodes $A - D$ and $D - E$ using the Union-Find algorithm. Node D and E have node A as their parent, and node A has a height of 1. The directed edge represents a parent-son relationship between nodes.

One application of the The *Union-Find* is to find the number of Eulerian cycles in a graph which is obtained by 7.1.

$$\#EulerianCycles = 2^k - 1, \tag{7.1}$$

where k is the number of times a connection of two elements of the same set is created. The algorithm in 7.4 represents a general implementation of the *Union-Find* method, which can be adapted to different applications.

Time Complexity:

Union: $O(1)$

Find: $O(\log n)$

Input:

For every node x added into the graph call the method `makeSet(x)`.

For every connection between two nodes a and b call method `unionSet(a,b)`.

Output:

Depends on the problem, usually the *Union-Find* algorithm is used when there are a large amount of queries.

Listing 7.4: Union-Find

```
1 #define N 1000
2
3 int p[N], rank[N];
4
5 void makeSet(int x) {
```

```
  p[x] = x;
  rank[x] = 0;
}

void link(int x, int y) {
  if (rank[x] > rank[y]) {
    p[y] = x;
  } else {
    p[x] = y;
    if (rank[x] == rank[y]) {
      rank[y] = rank[y] + 1;
    }
  }
}

int findSet(int x) {
  if (x != p[x]) {
    p[x] = findSet(p[x]);
  }

  return p[x];
}

void unionSet(int x, int y) { link(findSet(x), findSet(y)); }
```

The function `link` in 7.4 receives two root nodes, x and y. The height of node k is stored in `rank[k]`. If `rank[x]` is greater than `rank[y]` then x becomes the parent of y. On the other hand, if `rank[y]` is greater than `rank[x]` then y becomes the parent of x. If both root nodes have the same height any of both nodes can be the parent, so for this case we decided to set y as parent of x, that causes that the value of `rank[y]` increases in one.

7.4 Shortest Paths

In this section we will cover some of the most popular algorithms to find the shortest path in a graph. These algorithms work for weighted graphs, meaning that edges have a certain weight or cost. A path is a set of edges leading from vertex A to vertex B, and the cost of the path is the sum of the weights of all edges that form that path. Then, the shortest path between two nodes is the path with minimum cost. Consider the graph in 7.6. Here the cost of the shortest path that leads from node 0 to node 5 is 15, and the path is $0 - 3 - 2 - 5$.

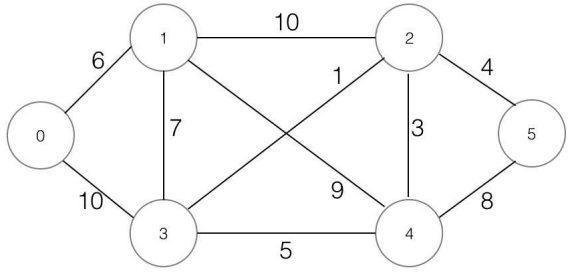


Figure 7.6: In a weighted graph edges have a cost associated to it. The cost of a path is the sum of the cost of all edges conforming that path.

7.4.1 Dijkstra's Algorithm

Published by Edsger W. Dijkstra in 1959 [14]. This algorithm finds the shortest paths between an initial node to all other nodes. Because of its greedy behavior, it only works for positive weights.

Given a graph G with n nodes numbered from 0 to $n-1$, an initial node a, and with $G_{ij} = \infty$ if there is no connection between node i and node j. The first step is to initialize the distance vector D and visited vector V as follows:

$$V_i = 0, D_i = \begin{cases} \infty & i \neq a \\ 0 & i = a \end{cases}$$

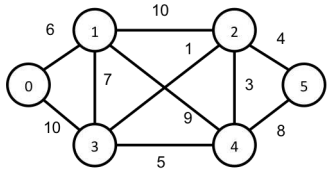
for every i from 0 to $n-1$.

When $V_i = 1$ we can be sure that the value stored in D_i contains the minimum cost to go from node a to node i. The algorithm finds the value of k, where D_k is minimum, and $V_k = 0$. Once it is found is marked as visited ($V_k = 1$) and the value of D is updated using the following formula:

$$D_j = \min\left(D_j, D_k + G_{kj}\right), \text{ where } V_j = 0 \tag{7.2}$$

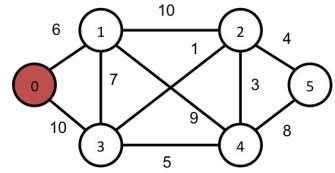
The process is repeated until all nodes are visited or until a value of k can't be found (disconnected graph). At the end the value of D_i stores the minimum cost to go from node a to node i. Figure 7.7 shows every iteration of Dijkstra's algorithm for the

graph in 7.6 with node 0 as the initial node.



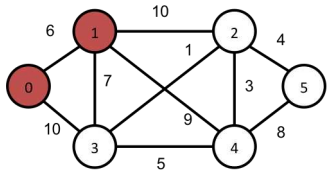
	0	1	2	3	4	5
D =	0	INF	INF	INF	INF	INF
V =	0	0	0	0	0	0

(a)



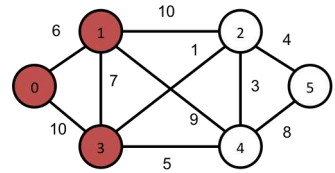
	0	1	2	3	4	5
D =	0	6	INF	10	INF	INF
V =	1	0	0	0	0	0

(b)



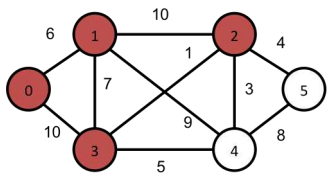
	0	1	2	3	4	5
D =	0	6	16	10	15	INF
V =	1	1	0	0	0	0

(c)



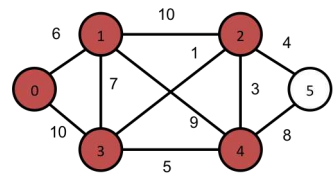
	0	1	2	3	4	5
D =	0	6	11	10	15	INF
V =	1	1	0	1	0	0

(d)



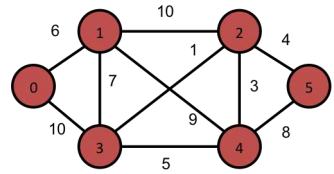
	0	1	2	3	4	5
D =	0	6	11	10	14	15
V =	1	1	1	1	0	0

(e)



	0	1	2	3	4	5
D =	0	6	11	10	14	15
V =	1	1	1	1	1	0

(f)



	0	1	2	3	4	5
D =	0	6	11	10	14	15
V =	1	1	1	1	1	1

(g)

Figure 7.7: Iterations of Dijkstra's algorithm for graph 7.6 with node 0 as initial node.

Finding the minimum number in D take $O(n)$ using a *Linear Search*, doing that n times makes the algorithm to run in $O(n^2)$. The code in 7.5 implements a generic *Dijkstra* algorithm for a weighted graph W with a given initial node.

Time Complexity: $O(n^2)$
Input:
n. The number of nodes in the graph. $(1 \leq n \leq 100)$.
W. The matrix of weights.
a. The initial node.
Output:
D. the minimum cost to reach some node i from node a is stored in D_i.

Listing 7.5: Dijkstra

```
#include <algorithm>
#define N 101
#define oo 10000000
using namespace std;

int D[N];    // Array of distances
int V[N];    // Array of visited nodes
int W[N][N]; // Adjacency matrix
int n;       // Number of nodes

void dijkstra(int a) {
  int pos;

  for (int i = 0; i < n; i++) {
    D[i] = oo;
    V[i] = 0;
  }

  D[a] = 0;
```

```
  for (int i = 0; i < n; i++) {
    pos = minVertex();
    if (pos == -1) {
      break;
    }

    V[pos] = 1;
    for (int j = 0; j < n; j++) {
      if (V[j] == 0) {
        D[j] = min(D[j], D[pos] + W[pos][j]);
      }
    }
  }
}

int minVertex() {
  int minVal, pos;

  minVal = oo;
  pos = -1;
  for (int i = 0; i < n; i++) {
    if (V[i] == 0 && D[i] < minVal) {
      minVal = D[i];
      pos = i;
    }
  }
  return pos;
}
```

7.4.2 Bellman-Ford

The Bellman-Ford algorithm was published in separate works by Richard Bellman [15], and Lester Ford Jr. [16]. The algorithm finds the cost of the shortest paths from a source vertex to all other vertices. At the contrary of Dijkstra's Algorithm, this algorithm is capable to handle negative weights and identify if a negative cycle exists.

Using a vector of distances D where D_i represents the minimum cost to travel from the starting node to node i. The algorithm goes trough every edge in the graph, if there is an edge that connects node a with node b and has a cost w, the vector D is updated as follows:

$$D_b = min(D_b, D_a + w) \tag{7.3}$$

The process is repeated n times for each edge to ensure that the value of D_i contains the minimum cost to go from the initial node to node i. The equation 7.3 is called *edge relaxation*.

To verify if there is a negative cycle in the graph an extra relaxation of the edges is made, and if a value of D changes then there is a negative cycle. That happens because in every iteration the algorithm finds a better path by looping in that negative cycle causing the vector D to change every time.

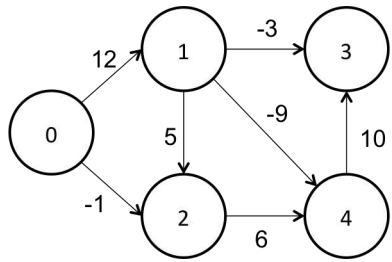


Figure 7.8: Directed graph with negative weights.

Using the graph defined in 7.8, to find the minimum cost to travel from node 0 to the rest of the nodes. The distance vector D is initialized as following:

$$D_i = \begin{cases} \infty & i \neq 0 \\ 0 & i = 0 \end{cases}$$

The next step is to perform the edge relaxation, which can be in any order, for example:

$$\begin{array}{ll} 1 \to 4 & D_4 = \min(\infty, \infty - 9) = \infty - 9 = \infty \\ 0 \to 1 & D_1 = \min(\infty, 0 + 12) = 12 \\ 1 \to 3 & D_3 = \min(\infty, 12 - 3) = 9 \\ 2 \to 4 & D_4 = \min(\infty, \infty + 6) = \infty \\ 0 \to 2 & D_2 = \min(\infty, 0 - 1) = -1 \\ 4 \to 3 & D_3 = \min(9, \infty + 10) = 9 \\ 1 \to 2 & D_2 = \min(-1, 12 + 5) = -1 \end{array}$$

The resulting vector of distances after the edge relaxation is

0	12	-1	9	∞

To assure that vector D contains the minimum cost to travel from node 0 to the rest of the nodes, we need to do the edge

relaxation at least n times. For this example, with one more edge relaxation is enough and it the vector will look like this:

0	12	−1	9	3

Notice that we must be careful to set the value of ∞ at the moment to code the Bellman-Ford algorithm, a small value can return a wrong answer.

The edge relaxation runs in $O(m)$ time, and it is performed n times, which makes the time complexity of the algorithm to be $O(nm)$. The Bellman-Ford algorithm is showed in 7.6, and it uses an array E to store the edges in the graph. The program returns 1 if there is a negative cycle, otherwise return 0.

Time Complexity: $O(nm)$
n. Number of nodes. $(1 \leq n \leq 100)$
m. Number of edges.
Input:
E. Array of edges.
src. The initial node.
Output:
D. the minimum cost to reach some node i from node `src` is stored in D_i. If there is a negative cycle the function `bellmanFord` returns 1.

Listing 7.6: Bellman-Ford

```
#define N 101
#define oo 10000000

class Edge {
 public:
  int u;
  int v;
  int w;

  Edge(int u = 0, int v = 0, int w = 0) {
    this->u = u;
    this->v = v;
    this->w = w;
  }
};

Edge E[N * N]; // Array of edges
int D[N];      // Array of distances
int n, m;      // Number of nodes and edges

```

```
21 int bellmanFord(int src) {
22   int u, v, w;
23
24   for (int i = 0; i < n; i++) {
25     D[i] = oo;
26   }
27
28   D[src] = 0;
29   for (int i = 0; i < n - 1; i++) {
30     for (int j = 0; j < m; j++) {
31       u = E[j].u;
32       v = E[j].v;
33       w = E[j].w;
34       D[v] = min(D[v], D[u] + w);
35     }
36   }
37
38   for (int j = 0; j < m; j++) {
39     u = E[j].u;
40     v = E[j].v;
41     w = E[j].w;
42     if (D[v] > D[u] + w) {
43       return 1; // Negative cycle!!
44     }
45   }
46
47   return 0; // No negative cycles
48 }
```

7.4.3 Floyd-Warshall

Described by Robert Floyd [17], who based it on the work of Stephen Warshall [18]. The Floyd-Warshall algorithm is used to find the cost of the shortest paths between all pair of vertices, which are stored in the same weighted adjacency matrix W. This algorithm works for positive and negative weights. It is important to notice that the time complexity of this algorithm makes it very hard to use it for real applications, and is recommended to use it in graphs with few vertices.

The idea behind the Floyd-Warshall's algorithm is to constantly update the minimum cost to go from some node i to another node j passing trough node k.
One advantage of this algorithm is that is easy to implement, just three nested loops and the matrix of weights is what it needs. See the code 7.7.

Time Complexity: $O(n^3)$
Input:
n. The number of nodes.

W. Matrix of weights.

Output:

W. The minimum cost to go from some node i to another node j is stored in W_{ij}.

Listing 7.7: Floyd-Warshall

```
void floydWarshall() {
  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        W[i][j] = min(W[i][j], W[i][k] + W[k][j]);
      }
    }
  }
}
```

Figure 7.9 shows how the matrix of weights W change for every value of k in Floyd-Warshall's algorithm using the graph in 7.8.

0	12	-1	∞	∞
∞	0	5	-3	-9
∞	∞	0	∞	6
∞	∞	∞	0	∞
∞	∞	∞	10	0

(a) Original matrix of weights

0	12	-1	∞	∞
∞	0	5	-3	-9
∞	∞	0	∞	6
∞	∞	∞	0	∞
∞	∞	∞	10	0

(b) $k = 0$. There is no edge from any node to node 0, so the matrix remains the same.

0	12	-1	9	3
∞	0	5	-3	-9
∞	∞	0	∞	6
∞	∞	∞	0	∞
∞	∞	∞	10	0

(c) $k = 1$. Nodes 3 and 4 can be reached from node 0 passing trough node 1.

0	12	-1	9	3
∞	0	5	-3	-9
∞	∞	0	∞	6
∞	∞	∞	0	∞
∞	∞	∞	10	0

(d) $k = 2$. Crossing trough node 2 doesn't led us to a better solution, so the matrix remains without change.

0	12	-1	9	3
∞	0	5	-3	-9
∞	∞	0	∞	6
∞	∞	∞	0	∞
∞	∞	∞	10	0

(e) $k = 3$. Node 3 cannot be an intermediate node because there is no edge coming out from it. The matrix remains the same.

0	12	-1	9	3
∞	0	5	-3	-9
∞	∞	0	16	6
∞	∞	∞	0	∞
∞	∞	∞	10	0

(f) $k = 4$. Going from node 2 to node 3 has a cost of 16 if we pass trough node 4.

Figure 7.9: Matrix of weights for different values of k in the Floyd-Warshall algorithm. The first row is equal to the resulting vector of distances in the Bellman-Ford algorithm.

7.4.4 A*

First described by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968 [19]. The A-star is an heuristic algorithm that is used to find the shortest path to go from some node A to some other node B. It is particular useful when memory or time constraints make hard to use the other algorithms seen so far.

The idea of the algorithm is to explore in every iteration the node with less cost associated to it. The cost of a given node k is define by:

$$f(k) = g(k) + h(k), \tag{7.4}$$

where $g(k)$ is the accumulated cost so far until reaching node k, and $h(k)$ is an **optimistic** guess of the cost needed to reach the destination node from node k, meaning that if $d(k)$ is the minimum cost to travel from node k to the destination node, then we must assure that $h(k) \leq d(k)$.

We must be very careful when choosing the heuristic function h, a wrong function will lead us to an incorrect result. This function depends on the problem to solve. For example, in the case of grid with obstacles where the goal is to reach some specific cell, starting from another cell just moving up, down, left and right. One possible heuristic function is to return the Manhattan distance between the current cell and the destination cell.

The code in 7.8 reads two numbers n ad m, that represents the number of nodes and edges in the graph respectively. m lines follow, each one containing three numbers a, b, and k, indicating that there is an edge with cost k connecting nodes a and b. Finally two more numbers are given indicating the initial and destination nodes respectively. The program prints the minimum cost to travel from the initial node to the destination node using the A* algorithm.

The class `Node` contains three attributes, `v` that refers to the node index, `g` that indicates the cost of reaching node `v` from the initial node, which represents the current cost. The value of `f` is an optimistic estimate of what will cost us to reach the destination. Since we use a `priority_queue` to get the node with less value of `f` we need to overload any of the operators $< >$.

Listing 7.8: A-star

```
#include <iostream>
#include <list>
#include <queue>
#include <vector>
#define oo 1000000
using namespace std;

class Node {
 public:
  int v;
  int g;
  int f;
  Node() {}
  Node(int v, int g, int f) {
    this->v = v;
    this->g = g;
    this->f = f;
  }

  bool operator<(const Node &node) const { return this->f > node.f; }
  bool operator>(const Node &node) const { return this->f < node.f; }
};

int nVertex, nEdges;
vector<list<Node> > G;
vector<int> h;
priority_queue<Node> L;

void aStar(int, int);

int main() {
  int a, b, k;
  int src, dst;

  cin >> nVertex >> nEdges;
  G.resize(nVertex);
  h.resize(nVertex);
```

```
38
39    for (int i = 0; i < nEdges; i++) {
40      cin >> a >> b >> k;
41      if (h[a] == 0) {
42        h[a] = k;
43      } else {
44        h[a] = min(h[a], k);
45      }
46
47      G[a].push_back(Node(b, k, 0));
48    }
49
50    cin >> src >> dst;
51    aStar(src, dst);
52
53    return 0;
54  }
```

The function `aStar` implements the $A*$ algorithm given the source and destination nodes. The nodes are added to a priority queue, and in each iteration the element at the top is selected, which represent the node with less cost. The heuristic for some node k used in this exercise is the minimum cost between all edges connected to k.

```
1   void aStar(int src, int dst) {
2     int u, v, g, f;
3     Node nodeA, nodeB;
4
5     u = src;
6     h[dst] = 0;
7     L.push(Node(src, 0, 0));
8
9     nodeA = L.top();
10    L.pop();
11    do {
12      cout << "Node : " << u << " f = " << nodeA.f << endl;
13      for (auto it = G[u].begin(); it != G[u].end(); it++) {
14        nodeB = *it;
15        v = nodeB.v;
16        g = nodeA.g + nodeB.g;
17        f = g + h[v];
18        L.push(Node(v, g, f));
19      }
20
21      nodeA = L.top();
22      L.pop();
23      u = nodeA.v;
24    } while (u != dst);
25
26    cout << nodeA.f << endl;
27  }
```

7.5 Strongly Connected Components

A strongly connected component (SCC) in a graph is a subgraph where all its nodes are connected directly or indirectly, thus we can reach any node from any other node. For figure 7.10 there are four SCC's, one formed by nodes $\{0, 1, 2, 3, 5\}$, other formed by $\{4, 6\}$, and two more formed by a single node $\{7\}$, and $\{8\}$.

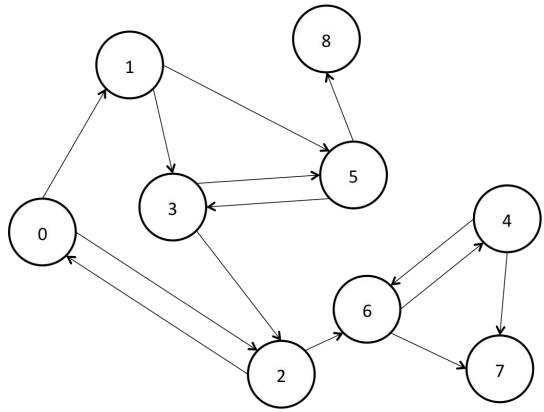


Figure 7.10: Directed graph with cycles.

If all SCC's are seen as individual nodes, we can convert a cyclic graph into a DAG (directed acyclic graph). In this section we will review two algorithms that find the strongly connected components in a graph, Tarjan's algorithm and Kosaraju's algorithm, both based in *DFS*.

7.5.1 Tarjan's Algorithm

Invented by Robert Tarjan in 1972 [20] and based on a *DFS*. Tarjan's algorithm finds the strongly connected components and cycles in a **directed graph**. The complexity of the algorithm is the same for the *DFS* $O(n + m)$. If we use an adjacency matrix as in the code in 7.9, the time complexity is $O(n^2)$.

As nodes are visited in the *DFS*, they are added to a stack L and enumerated, this way each node v will have an index or identifier S_v. Each node v also needs to keep track of the node

with the smallest index that is reachable from v, including v itself, we will call it low_v. Once all adjacent nodes of v have been explored, and if low_v is equal to S_v, then we can be sure that there is a cycle, and all nodes added to L after v are part of the same SCC. Remove all the elements of the SCC from L and continue with the search.

The time complexity of the algorithm is the same as the *DFS*, but needs more memory to store the values of S_i and low_i for each $i = 0, \ldots, n-1$. The algorithm is implemented in 7.9

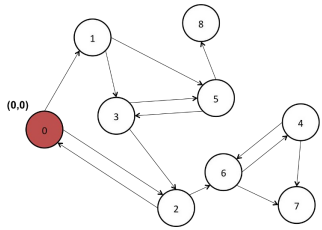
Time Complexity: $O(n^2)$
Input:
n. The number of nodes. $(1 \leq n \leq 100)$.
G. The adjacency matrix.
Output:
Prints the strongly connected components of G

Listing 7.9: Tarjan Algorithm

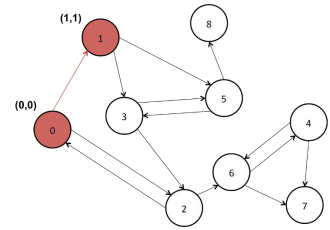
```
#include <algorithm>
#define N 101
using namespace std;

int G[N][N];
int S[N], LOW[N], L[N], R[N];
int n, nVertex, nComponents;

void tarjan() {
  for (int i = 0; i < n; i++) {
    if (S[i] == -1) {
      nVertex = 0;
      nComponents = 0;
      DFS(i);
    }
  }
}

void DFS(int v) {
  int k;

  S[v] = nVertex++;
  LOW[v] = S[v];
  L[nComponents++] = v;
  R[v] = 1;

  for (int i = 0; i < n; i++) {
    if (G[v][i] == 1) {
      if (S[i] == -1) {
        DFS(i);
        LOW[v] = min(LOW[v], LOW[i]);
      } else if (R[i] == 1)
        LOW[v] = min(LOW[v], S[i]);
```

```
34      }
35    }
36
37    if (S[v] == LOW[v]) {
38      printf("SCC:\n");
39      do {
40        k = L[nComponents - 1];
41        R[k] = 0;
42        nComponents--;
43        printf("%d\n", k);
44      } while (k != v && nComponents > 0);
45    }
46  }
```

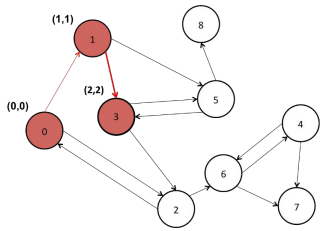
Figure 7.11 shows how the values of S_v and low_v change trough the *DFS* for the graph in 7.10.



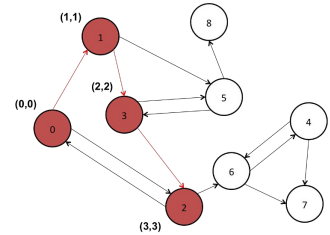
(a) $L = \{0\}$. Start the *DFS* from node 0.



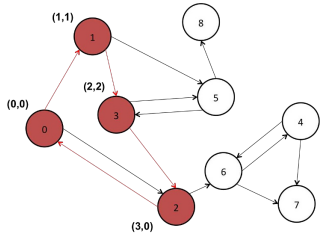
(b) $L = \{0, 1\}$. Visit node 1.



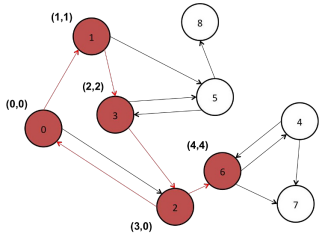
(c) $L = \{0, 1, 3\}$. Visit node 3.



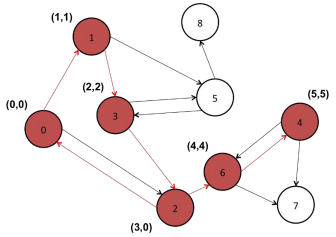
(d) $L = \{0, 1, 3, 2\}$. Visit node 2.



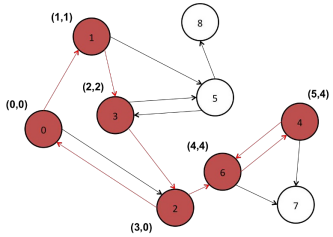
(e) $L = \{0, 1, 3, 2\}$. $low_2 = 0$ because node 2 is connected to the node with index 0.



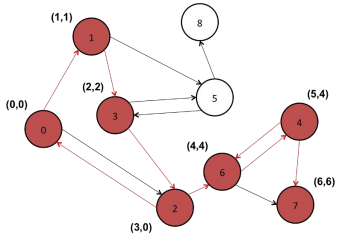
(f) $L = \{0, 1, 3, 2, 6\}$. Visit node 6.



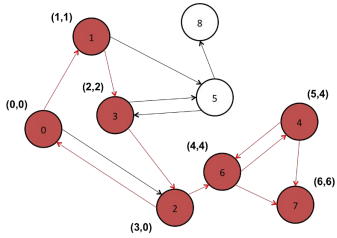
(g) $L = \{0, 1, 3, 2, 6, 4\}$. Visit node 4.



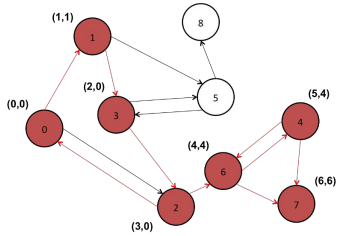
(h) $L = \{0, 1, 3, 2, 6, 4\}$. $low_4 = 4$ because is connected to a node with a smaller index.



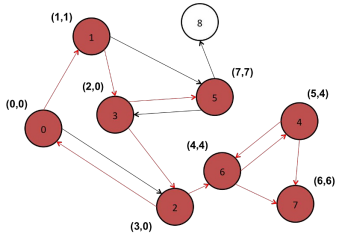
(i) $L = \{0, 1, 3, 2, 6, 4, 7\}$. Visit node 7. Because node 7 has been full explored and $S_7 = low_7$ then node 7 is a SCC.



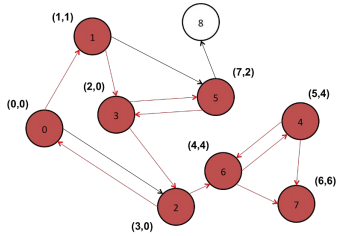
(j) $L = \{0, 1, 3, 2, 6, 4\}$. Node 7 has been removed from L, and low_6 remains equal because node 7 has a larger index. $S_6 = low_6$ meaning that nodes 6 and 4 are part of a SCC.



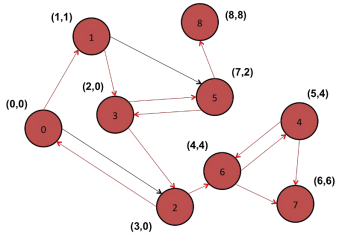
(k) $L = \{0, 1, 3, 2\}$. $low_3 = 0$, because it comes from node 2, and low_2 is smaller than low_3.



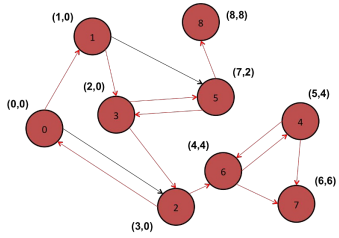
(l) $L = \{0, 1, 3, 2, 5\}$. Visit node 5.



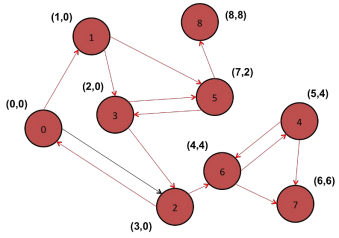
(m) $L = \{0, 1, 3, 2, 5\}$. $low_5 = 2$, because is connected to node 3, which has $S_3 = 2$.



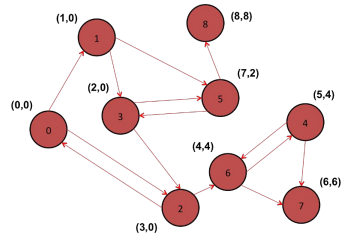
(n) $L = \{0, 1, 3, 2, 5, 8\}$. Visit node 8. Because node 8 doesn't have any adjacent nodes and $S_8 = low_8$, it forms another SCC.



(o) $L = \{0, 1, 3, 2, 5\}$. $low_1 = 0$, because it comes from node 3 and $low_3 = 0$.



(p) $L = \{0, 1, 3, 2, 5\}$. low_1 remains without change because S_5 is larger.



(q) $L = \{0, 1, 3, 2, 5\}$. low_0 remains without change because S_2 is larger. $S_0 = low_0$ then nodes $0, 1, 3, 2, 5$ are part of a SCC. then all of them are removed from L, leaving the stack empty.

Figure 7.11: Tarjan's algorithm. The values of S_v and low_v are changing in every step of the *DFS* for graph in 7.10. If node v, after all its adjacent nodes has been explored, happens that $S_v = low_v$, then v an all nodes inserted to the stack L after v form a SCC.

7.5.2 Kosaraju's Algorithm

In 1983 Aho, Hopcroft, and Ullman [21] published the algorithm and credit it to S. R. Kosaraju. The algorithm first runs a *DFS* to get a topological sort. The second step is to compute the transpose of the adjacency matrix. Finally, following the order obtained in the first DFS and using the transpose matrix, the graph is traversed and the number of times a *DFS* is started is the number of SCC's in the graph, and the nodes visited in the same *DFS* are part of the the same SCC.

Running a *DFS* using the order obtained in a topological sort will cause to traverse the graph from root nodes to leaf nodes. But using the reversed graph (with the transponse matrix), will cause to traverse the graph from leaf nodes to root nodes.

If node b is reachable from node a in the original graph, then a will be reachable from node b in the reversed graph.

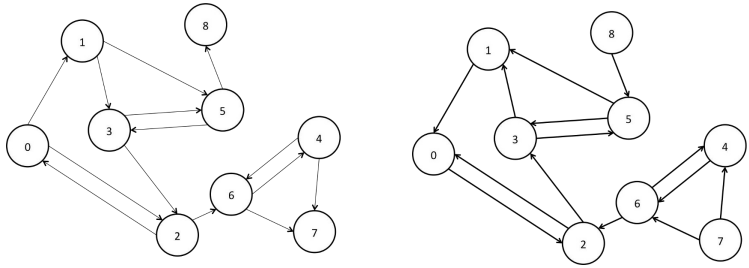


Figure 7.12: 7.12a shows the original graph. 7.12b shows the reversed graph with the arrows to point in the opossite direction.

For the graph in 7.12a the topological sort would be $0, 1, 3, 5, 8, 2, 6, 4, 7$, and if we go through that order and start a *DFS* for each node that has not been visited using the reversed graph represented in 7.12b. We obtain that the first *DFS* visits the nodes $0, 2, 3, 1, 5$, and occurs that they form a SCC in the original graph. Following the topological order, the next *DFS* only visits node 8, the next one visits nodes 6 and 4, and the last one visits node 7. All of them SCC's in the original graph.

Compared to Tarjan's algorithm, Kosaraju's algorithm has worst performance, because it need two *DFS*, meanwhile Tarjan only needs one, but on the other hand, Kosaraju doesn't need to store additional information beside the topological sort. Program 7.10 implements the Kosaraju's algorithm to identify the number of *SCC's* in a graph G.

Time Complexity: $O(n^2)$
Input:
`n`. Number of vertices.
`G`. The adjacency matrix.
Output:
`L`. Represents the topological sort.
`nComponents`. Stores the number of SCC's

Listing 7.10: Kosaraju's Algorithm

```
1 void kosaraju() {
2   int k;
3
4   memset(S, 0, sizeof(S));
5   nVertex = n - 1;
6   nComponents = 1;
```

```
7
8     for (int i = 0; i < n; i++) { // Apply the first DFS
9       if (S[i] == 0) {
10        DFS(i, true); // Obtain the post-order traversal
11      }
12    }
13
14    // Obtain the transponse of the adjacency matrix
15    for (int i = 0; i < n; i++) {
16      for (int j = i; j < n; j++) {
17        swap(G[i][j], G[j][i]);
18      }
19    }
20
21    memset(S, 0, sizeof(S));
22    nComponents = 0;
23    for (int i = 0; i < n; i++) {
24      k = L[i]; // Visit the nodes using the order obtained in the first DFS
25      if (S[k] == 0) {
26        nComponents++; // Increment the number of SCC
27        DFS(k, false);
28      }
29    }
30  }
31
32  void DFS(int v, bool flag) {
33    // Label the vertex with the number
34    // of the strongly connected component
35    // it belongs.
36    S[v] = nComponents;
37
38    for (int i = 0; i < n; i++) {
39      if (G[v][i] == 1 && S[i] == 0) {
40        DFS(i, flag);
41      }
42    }
43
44    if (flag) {        // if flag is true
45      L[nVertex--] = v; // Add v to the stack
46    }
47  }
```

Articulation Points

An articulation point in a graph is a vertex that if removed, the graph becomes disconnected. See figure 7.13.

A variant of Tarjan's algorithm can be used to find articulation points. Basically a node v is an articulation point if:

1. v is the root and has more than one children.
2. if $low_i \geq S_v$, where i is the adjacent node of v which the recursive function of the *DFS* returns from. This means that there is no connection from node i to previously indexed nodes, so removing v will disconnect the graph.

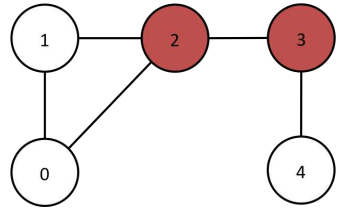


Figure 7.13: Nodes 2 and 3 are articulation points, because if they are removed the graph becomes disconnected. Node 1 is not an articulation point, because node 0 is directed connected to node 2.

The code 7.11 increments the variable `nLinks` in one every time an articulation point is found. The vector V is used to avoid counting duplicated articulation points.

Listing 7.11: Articulation Points

```
void DFS(int v, int prev) {
  S[v] = nVertex++;
  low[v] = S[v];

  for (int i = 0; i < n; i++) {
    if (graph[v][i] == 1) {
      // if is a tree vertex (unvisited)
      if (x[i] == -1) {
        DFS(i, v);
        low[v] = min(low[v], low[i]);

        if (S[v] == 0) {
          if (S[i] >= 2 && V[v] == 0) {
            V[v] = 1;
            nLinks++; // is an articulation point
          }
        } else if (low[i] >= S[v] && V[v] == 0) {
          V[v] = 1;
          nLinks++; // is an articulation point
        }
      } else if (i != prev)
        low[v] = min(low[v], S[i]);
    }
  }
}
```

Bridge Detection

A bridge is an edge that if removed the graph becomes disconnected. See figure 7.14. As with the articulation points, a variant of Tarjan's algorithm can be used to find bridges. For this case we say there is a bridge connecting nodes i and v if $low_i > S_v$, where i

is the node which the recursive function returns from. That means that node i cannot reach any of the nodes indexed before him, so if we remove the edge $v - i$, node i will not be able to reach any of the previously indexed nodes.

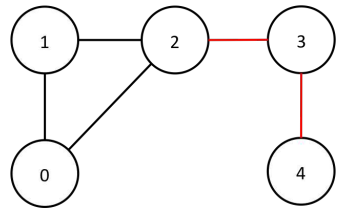


Figure 7.14: The edge connecting nodes 2 and 3, and the edge connecting nodes 3 and 4 are bridges. None of the other edges if removed disconnects the graph.

The program 7.12 increments by one the variable `nBridges` when a bridge is found given a graph G.

Listing 7.12: Bridge Detection

```
void DFS(int v, int prev) {
  S[v] = nVertex++;
  low[v] = S[v];

  for (int i = 0; i < n; i++) {
    if (G[v][i] == 1) {
      if (S[i] == -1) {
        DFS(i, v);
        if (low[i] > S[v]) {
          nBridges++; // There is a bridge between i and v
        }
        low[v] = min(low[v], low[i]);
      } else if (i != prev) { // Is a back edge?
        low[v] = min(low[v], S[i]);
      }
    }
  }
}
```

7.6 Minimum Spanning Tree

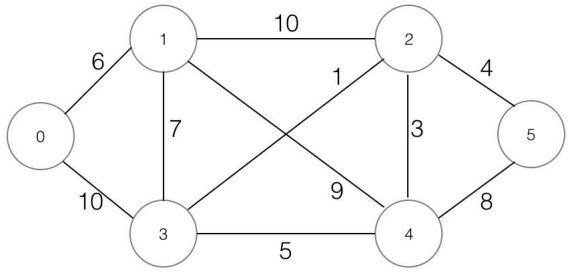
Consider a weighted graph with n nodes and m edges that is connected and is not directed. The minimum spanning tree (MST) of that graph is a sub-graph with the following characteristics.

1. Contains $n - 1$ edges

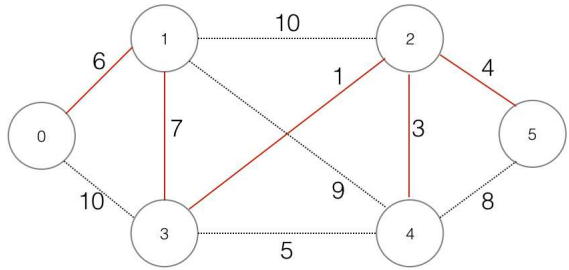
2. The graph remains connected

3. the cost of the tree is minimum

The points listed above describe a sub-graph with no cycles, where all the nodes are connected, and which cost is minimal. The cost of a MST is the sum of the weights of all its edges. Image 7.15 shows an example of a MST in a graph. Is important to notice that a graph can have more than one MST.



(a) Connected graph



(b) MST of the graph

Figure 7.15: Image 7.15a shows a connected graph. Image 7.15b shows in red solid lines the $n-1$ edges that are part of the MST with a cost of 21.

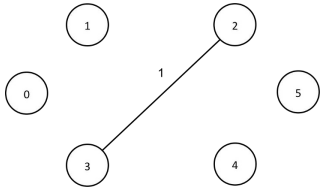
In this section we will analyze two algorithms that allow us to find the *MST*. One is the *Kruskal's algorithm* and the other is the *Prim algorithm*. Each one has its own advantages and for some problems is better to use one over the other, but at the end the

problem is the same, to find a set of edges that are part of the MST.

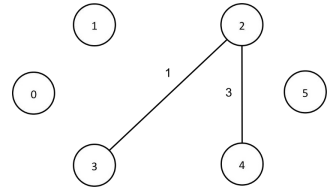
7.6.1 Kruskal's Algorithm

Reported by Joseph Kruskal in 1956 [22]. Kruskal's algorithm first needs to sort the edges in the graph in ascending order according to their weights. Then iterate from the beginning of the sorted array of edges and check if the current edge creates a cycle, if it does, discard it and move to the next edge, otherwise add that edge to the MST.

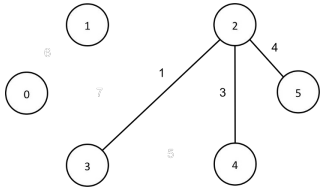
One way to check if an edge creates a cycle is to use the Union-Find algorithm, just take the nodes connected by the edge and if those nodes are part of the same set then that edge will create a cycle. If they are part of different sets, then the edge will not create a cycle, so add it to the MST and merge both sets. Figure 7.16 shows how Kruskal's algorithm works to find the MST of the graph in 7.15a.



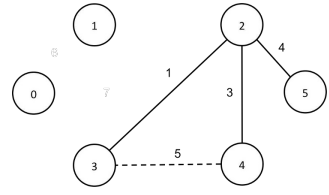
(a) The edge with minimum cost is the one connecting nodes 2 and 3, add it to the MST.



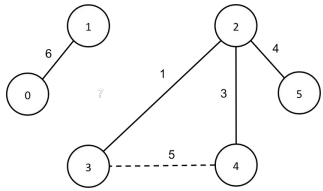
(b) The next edge is $2-4$, it doesn't form a cycle so is added to the MST.



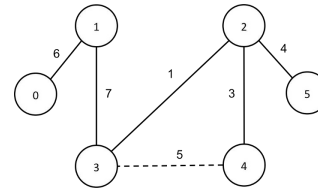
(c) The next edge is $2-5$ and is added to the MST.



(d) The next edge is the one connecting nodes 3 and 4, but if it is added it will create a cycle, so we skip this edge.



(e) Add edge $0-1$ to the MST.



(f) The edge $1-3$ is added to the MST. We have added all 5 edges, any other edge will generate a cycle, so we can stop looking.

Figure 7.16: Kruskal's algorithm needs that the edges to be previously sorted by their cost. Once $n-1$ edges are added to the MST the process can be stopped.

The performance of Kruskal's algorithm will depend on the algorithm used to sort the edges, and the algorithm used to identify cycles. For the sorting part an algorithm that runs in $O(m \log m)$ can be used, like *Merge Sort* or *Heap Sort*. To detect cycles is recommended to use the *Union-Find* algorithm, which runs in $O(\log n)$ and is easy to implement.

The code showed in 7.13 implements a class `Edge` to easily represent an edge and be able to create an array of edges. This class contains three values u, v and w indicating that there is an edge connecting nodes u and v with a cost of w. The overloaded operator is needed to sort the array of edges. The program uses the *Union-Find* method described in 7.4.

Time Complexity: $O(m \log m + m \log n)$
$O(m \log m)$ to sort all the edges
$O(m \log n)$ for the union find of the sorted edges
Input:
edge. A vector with all the edges in the graph.
Output:
The cost of the MST.

Listing 7.13: Kruskal's Algorithm

```
#include <algorithm>
#include <vector>
using namespace std;

class Edge {
 public:
```

```
  int u;
  int v;
  int w;

  Edge(int u = 0, int v = 0, int w = 0) {
    this->u = u;
    this->v = v;
    this->w = w;
  }

  bool operator<(const Edge &b) const { return this->w < b.w; }
};

vector<Edge> edge;
int nVertex;

int kruskal() {
  int n, a, b, w;
  int d = 0;

  sort(edge.begin(), edge.end());

  n = nVertex;
  for (int i = 0; n > 1; i++) {
    a = edge[i].u;
    b = edge[i].v;
    w = edge[i].w;

    if (findSet(a) != findSet(b)) {
      d += w;
      unionSet(a, b);
      n--;
    }
  }

  return d;
}
```

7.6.2 Prim's Algorithm

Discovered by Prim [23], but invented earlier by V. Jarník in 1930. Prim's algorithm is similar to Dijkstra's algorithm. The adjacency matrix is filled with ∞ in those locations where there is no edge. The initial node doesn't matter in Prim, any node can be used as the initial node. In every iteration as in Dijkstra's the minimum value of vector D must found across all non-visited nodes, and then proceed to update the vector D. While in Dijkstra's the vector D is updated with the cost to move from the initial node to other non-visited nodes, in Prim the vector is updated using only the weight of the edges. The value of element D_J is given by the equation 7.5.

$$D_j = \min\left(D_j, W_{kj}\right), \tag{7.5}$$

where k represents the position of the minimum value of D

across the non-visited vertices. Table 7.4 shows the vector D in every iteration of Prim algorithm for the graph in 7.15a and node 0 as the initial node.

it. 1	0	∞	∞	∞	∞	∞
it. 2	**0**	6	∞	10	∞	∞
it. 3	**0**	**6**	10	7	9	∞
it. 4	**0**	**6**	1	**7**	5	∞
it. 5	**0**	**6**	**1**	**7**	3	4
it. 6	**0**	**6**	**1**	**7**	**3**	4

Table 7.4: Vector D for the Prim algorithm. Red numbers represents the minimum value chosen in an specific iteration. Bold numbers represent already visited nodes.

The cost of the MST for the graph 7.15a is the sum of all red numbers in table 7.4, $0+6+7+1+3+4 = 21$. If we want to know which edges are part of the MST, then we must have a record of the nodes connected to the ones represented in D, something similar to how a path is obtained in *Dijkstra*. The program 7.14 consists on a function `prim` that returns the cost of the *MST* of a graph given its weighted matrix W by using *Prim's* algorithm starting from node 0.

Time Complexity: $O(n^2)$
Input:
n. AThe number of nodes.
W. A weighted and connected graph.
Output:
The cost of the MST.

Listing 7.14: Prim Algorithm

```
int prim() {
  int k, minVal;
  int d = 0;

  for (int i = 1; i < n; i++) {
    D[i] = W[0][i];
    V[i] = 0;
  }

  V[0] = 1;
  for (int i = 1; i < n; i++) {
    minVal = oo;
    k = -1;
```

```
14
15      for (int j = 1; j < n; j++) {
16        if (V[j] == 0 && D[j] < minVal) {
17          minVal = D[j];
18          k = j;
19        }
20      }
21
22      if (k == -1) {
23        printf("Error: No connection found");
24        break;
25      }
26
27      d += D[k];
28      V[k] = 1;
29      for (int j = 1; j < n; j++) {
30        if (V[j] == 0 && W[k][j] < D[j]) {
31          D[j] = W[k][j];
32        }
33      }
34    }
35
36    return d;
37  }
```

7.7 Maximum Bipartite Matching

A bipartite graph can be defined as two sets of vertices A, and B, where all the edges connect a node from set A with a node of set B. Elements of the same set are not connected directly.

A matching is a set of edges, where no two edges share a common vertex. Then a maximum matching is a matching that contains the largest number of edges.

Consider the case of an university that has n professors labeled as $0, 1, \cdots n-1$, and m courses numbered as $0, 1 \cdots, m-1$. By politics of the university, no professor can teach more than one course, if a professor cannot be found to teach a course, then that course cannot be opened that semester. The goal is to maximize the number of courses that can be imparted in the semester.

This task can be seen as a graph, where courses and professors are the nodes, and an edge from node i to node j means that professor i can teach course j. See image 7.17.

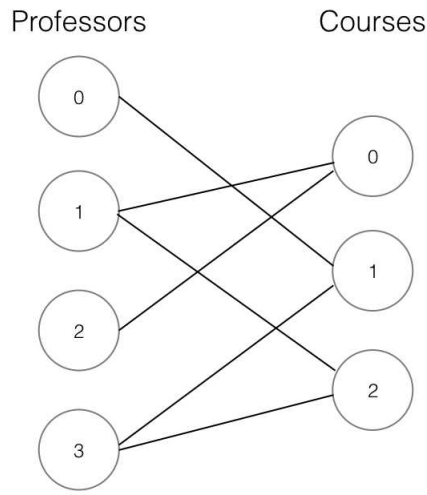


Figure 7.17: Bipartite graph with four professors and three courses.

For the bipartite graph in 7.17, one possible solution is that professor 0 teaches course 1, professor 1 teaches course 2, and professor 2 teaches course 0, with that arrangement all the courses can be opened for the semester. Professor 3 can have a sabatical year.

One possible solution for this problem is to try to assign a professor p to some course c that doesn't has a professor assigned, or if it already has a professor assigned, then try move the current professor to another course, in order that professor p teaches course c. That idea is basically a *DFS*, which have a time complexity of $O(V + E)$, for this case $V = n + m$, and $E \leq n * m$. The search is repeated m times, meaning that that the time complexity for the algorithm is $O(mn + m^2 + nm^2) = O(nm^2)$. The implementation for this solution is showed in program 7.15.

Time Complexity: $O(nm^2)$
Input:
n. Number of professors. $1 \leq n \leq 100$.
m. Number of courses. $1 \leq m \leq 100$.
graph. The bipartite graph, where $graph_{ij} = true$ if professor i can teach course j.
Output:

cnt. The maximum matching.

Listing 7.15: Maximum Bipartite Matching

```
#include <cstring>
#define M 128
#define N 128

bool graph[M][N];
bool seen[N];
int matchL[M], matchR[N];
int n, m;
bool bpm(int);

int main() {
  // Read input and populate graph[][] // Set m, n
  memset(matchL, -1, sizeof(matchL));
  memset(matchR, -1, sizeof(matchR));

  int cnt = 0;
  for (int i = 0; i < m; i++) {
    memset(seen, 0, sizeof(seen));
    if (bpm(i)) {
      cnt++;
    }
  }

  // cnt contains the number of professors assigned to a course
  // matchL[i] contains the course of professor i or -1
  // matchR[j] contains the professor in course j or -1
  return 0;
}

bool bpm(int u) {
  for (int v = 0; v < n; v++) {
    if (graph[u][v]) {
      if (seen[v]) {
        continue;
      }

      seen[v] = true;
      if (matchR[v] < 0 || bpm(matchR[v])) {
        matchL[u] = v;
        matchR[v] = u;
        return true;
      }
    }
  }
  return false;
}
```

7.8 Flow Network

A flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. A flow must satisfy that

the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow.

A network is a graph $G = (V, E)$, together with a non-negative function $c : V \times V \to \Re$, called the capacity function, and a flow function $f : V \times V \to \Re$, such that:

- Incoming flow is equal to outgoing flow. $f(u, v) = -f(v, u)$.
- The flow through an edge cannot exceed its capacity. $f(u, v) \leq c(u, v)$.

Network flow can be used to model fluid in pipes, traffic systems, image denoising, electrical current, among others applications.

7.8.1 Ford-Fulkerson Algorithm

Invented in 1956 by L. R. Ford, and D. R. Fulkerson [16]. Ford-Fulkerson's algorithm finds the maximum flow in a network. In order to achieve this, a path, $s_1, s_2, \cdots, s_k$, between source and sink must be found, where s_1 is the source, and s_k is the sink. This path is called *the augmenting path*. An edge (u, v) can be used in the augmenting path only if:

$$c(u, v) - f(u, v) > 0 \tag{7.6}$$

If not augmenting path is found the algorithm ends, otherwise, the maximum amount of flow (f_{max}) that can travel trough this path must be calculated using equation 7.6 for each edge of the augmenting path and add that value to the network flow.

$$f(s_i, s_{i+1}) = f(s_i, s_{i+1}) + f_{max} \tag{7.7}$$
$$f(s_{i+1}, s_i) = f(s_{i+1}, s_i) - f_{max} \tag{7.8}$$

for every $i = 1, 2, \cdots, k - 1$.

Once this is done, another augmenting path must be found, and the process is repeated until no augmenting path is found.

The execution time for this algorithm depends in the algorithm used to find the augmenting path. The code below uses a *BFS*

to do that, which has a time complexity of $O(V + E)$, and in the worst case for every augmented path found, the flow will increase in 1 unit, meaning that the time complexity of *Ford-Fulkerson* is $O(F(V + E))$, where F is the maximum flow of the network.

The `maxFlow` function in 7.16 receives the *source* and *sink* of the network, and by using matrix w as the capacity function, and matrix $flow$ as the flow function, it returns the maximum flow by using the *Ford-Fulkerson* algorithm.

Time Complexity: $O(f(m + n))$
f. Maximum flow
Input:
n. Number of nodes. $2 \leq n \leq 100$
w. Capacity function
source. Source node
sink. Sink node
Output:
The maximum flow

Listing 7.16: Ford-Fulkerson Algorithm

```
#include <algorithm>
#include <cstring>
#include <queue>
#define N 105
#define WHITE 0
#define GRAY 1
#define BLACK 2
#define oo 100000000
using namespace std;

long w[N][N];   // Adjacency matrix
long flow[N][N]; // Flow matrix
long color[N], pred[N];
long n;
queue<long> Q;

long maxFlow(long source, long sink) {
  long increment;

  // Initialize empty flow.
  long max_flow = 0;
  memset(flow, 0, sizeof(flow));

  // While there exists an augmenting path,
  // increment the flow along this path.
  while (BFS(source, sink)) {
    // Determine the amount by which we can increment the flow.
    increment = oo;

    for (long u = sink; pred[u] >= 0; u = pred[u]) {
      // Now increment the flow.
```

```
      increment = min(increment, w[pred[u]][u] - flow[pred[u]][u]);
    }

    for (long u = sink; pred[u] >= 0; u = pred[u]) {
      flow[pred[u]][u] += increment;
      flow[u][pred[u]] -= increment;
    }

    max_flow += increment;
  }

  // No augmenting path anymore. We are done.
  return max_flow;
}
```

The task of the `BFS` function is to find the augmenting path across the network. It receives the indexes of the source and sink nodes, and stores the augmenting path in the array `pred`. If the sink cannot be reached it returns `false`, otherwise returns `true`.

```
long BFS(long start, long target) {
  long u;

  memset(color, 0, sizeof(color));
  Q.push(start);
  pred[start] = -1;
  color[start] = GRAY;

  while (!Q.empty()) {
    u = Q.front();
    Q.pop();

    color[u] = BLACK;

    // Search all adjacent white nodes v. If the capacity
    // from u to v in the residual network is positive,
    // enqueue v.
    for (long v = 0; v < n; v++) {
      if (color[v] == WHITE && w[u][v] - flow[u][v] > 0) {
        Q.push(v);
        pred[v] = u;
        color[v] = GRAY;
      }
    }
  }

  // If the color of the target node is black now,
  // it means that we reached it.
  return color[target] == BLACK;
}
```

7.9 Chapter Notes

The applications of graph theory are vast, from image processing to social network. For example. The Ford-Fulkerson method [16]

can be used to remove the noise of a binary image, where the nodes connected to the source are labeled with one color, and the rest connected to the sink are labeled with another color. The Kruskal's algorithm is useful for image segmentation by creating different minimum spanning trees considering the pixels as nodes. Tarjan's algorithm [20] can be used to identify cicles in the movement of an object obtained from motion capture, where each sensor correponds to a node. et. al.

A more strict analysis of the algorithms seen in this chapter can be found in the book *"Introduction to Algorithms"* [1]. Sedgwick [5] describe some of *state of the art* algorithms and includes source code for the key parts written in `C++`. Topcoder includes tutorials [24] with tricks and hints useful to solve programming problems related to graph theory.

Problems about Graph Theory in programming contests are almost a fact, for that reason we recommend to try to solve some of the problems in the links listed bellow.

- `http://acm.timus.ru/problemset.aspx?space=1&tag=graphs`
- `https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=116`
- `https://www.urionlinejudge.com.br/judge/en/problems/index/7`

Appendix E contains variants of some of the algorithms seen in this chapter applied to solve specific problems.

7.10 Exercises

1. Given a connected graph $G = \{V, E\}$. How can we know if it is possible to color each node of the graph using two colors (black and white), in such a way that two adjacent nodes don't have the same color?

2. If a connected graph has exactly two nodes with odd degree. Is it possible to find an Eulerian path? Explain.

3. If all nodes in a connected graph have an even degree, can we find an Eulerian cycle? Explain.

4. Explain with an example why Dijkstra's algorithm doesn't work with negative weights.

5. Can we solve the *Maximum Bipartite Matching* problem using the *Ford-Fulkerson* algorithm? How?

6. A floor on a building can be represented as a grid of $n \times n$ cells. For example, $X_{1,0,0}$ refers to the position $(0, 0)$ on the 1^{st} floor. Each position can take two possible values, 0 if it is an empty space, or 1 if there is a wall. What algorithm would you use to find the minimum cost to go from one location on the building to another? Knowing that the a person can move only to the neighboring cells in the same floor (north, east, south and west) and to the same location in the floor above and in the floor below, as long as there is no wall. For example, if a person is located in $X_{2,1,1}$ it can move to locations $X_{2,1,0}, X_{2,1,2}, X_{2,0,1}, X_{2,2,1}, X_{1,1,1}, X_{3,1,1}$. The cost of moving to another cell is 1 unit. Justify your answer.

7. What happens when we raise to the k^{th} power the adjacency matrix of a graph? What does it represent?

8. Explain why finding the maximum flow in a graph is equivalent to find the sum of the edge weights in the minimum cut?

8

Geometry

"If everyone is thinking alike, then somebody isn't thinking."

– George S. Patton

Computational geometry is vastly used in the areas of computer vision, computer graphics, geographic systems, et.al. For example, detecting a collision between objects in a video game, identifying regions in a map, identifying objects or persons in an image, and so on. Most of the topics that are boarded in this chapter have to do with polygons, like calculating the area of polygon, identify if a point is inside or outside a polygon, find the convex hull of a cloud of points, et.al.

A real-life application is the identification of historical images. In some cases it is necessary to know how a certain area of the map has changed trough the years, in order to study phenomenons like deforestation, forest fire, population growth, et.al. To do that is necessary to compare an actual image and a previous image and check if it is the same area, and that involves analyzing the polygons inside the images. This problem requires to develop an algorithm that receives two polygons and measure how similar they are, of course is necessary to be aware of the scale, rotation, noise, etc.

Another application is to obtain the location of a robot given its current position and direction. Consider the case showed in figure 8.1, where a robot is located in position (x_1, y_1) with an

angle of θ with the x-axis, and rotate an angle of β about the origin. What is the new position (x_2, y_2) of the robot?

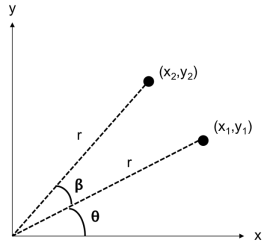


Figure 8.1: Rotation of a point by an angle of β.

According to the figure we have that the current position is given by

$$\begin{aligned} x_1 &= r\cos(\theta) \\ y_1 &= r\sin(\theta), \end{aligned} \tag{8.1}$$

and the new position is obtained with

$$\begin{aligned} x_2 &= r\cos(\theta+\beta) \\ y_2 &= r\sin(\theta+\beta). \end{aligned} \tag{8.2}$$

Using geometry identities and equations in 8.1 we have that equations in 8.2 can be transformed into

$$\begin{aligned} x_2 &= r\left[\cos(\theta)\cos(\beta) - \sin(\theta)\sin(\beta)\right] = x_1\cos(\beta) - y_1\sin(\beta) \\ y_2 &= r\left[\cos(\theta)\sin(\beta) + \sin(\theta)\cos(\beta)\right] = x_1\sin(\beta) + y_1\cos(\beta) \end{aligned} \tag{8.3}$$

Equations in 8.3 can be vectorized and represented as matrix multiplications.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \tag{8.4}$$

The 2×2 matrix in 8.4 is called *"rotation matrix"* and is used to obtain the new position of a point after being rotated an angle of β about the origin. Now suppose that we

want to rotate a polygon not just a point, well, in order to do that we just need to multiply each one of the vertices by the rotation matrix and the whole polygon will rotate about the origin.

Is important to mention that this chapter focus on algorithms that are commonly used in programming competitions. It doesn't contain the classic topics covered in a geometry book, so it is expected that the reader have some basic knowledge about geometry and arithmetic.

8.1 Point Inside a Polygon

In this section we will review algorithms that can help us to find out if a point is inside a polygon. It seems like an easy problem, but depending on the polygon and the problem's specifications we must try different approaches.

8.1.1 Point in Convex Polygon

Given a point z and a convex polygon P with its coordinates sorted counterclockwise

$$p_0, p_1, \cdots, p_n = (x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)$$

With $p_n = p_0$. Calculate the signed area of every triangle formed by points (z, p_i, p_{i+1}), for $i = 0, \ldots, n-1$. If every signed area computed is positive, then z is inside the polygon, if at least one signed area is zero, then it is in the border, in other case it is outside the polygon.

The signed area of the triangle formed by the points A, B and C is obtained using the determinant of the matrix formed by the three points using 1 as their z-coordinate. See 8.5

$$\begin{aligned} Area(A,B,C) &= \frac{1}{2}\begin{vmatrix} A_x & A_y & 1 \\ B_x & B_y & 1 \\ C_c & C_y & 1 \end{vmatrix} \\ &= \frac{1}{2}\left(A_xB_y + B_xC_y + C_xA_y - C_xB_y - A_xC_y - B_xA_y\right) \\ &= \frac{1}{2}\left((B_X - A_x)(C_y - A_y) - (C_x - A_x)(B_y - A_y)\right) \end{aligned} \tag{8.5}$$

The function `isInsidePolygon` in 8.1 receives a point p and identifies if it is inside of a convex polygon P, which is an array of n points as described above, while the function `area` calculates the signed area of a triangle using equation 8.5, notice that we don't multiply by 0.5 because we are only interested in the sign.

Time Complexity: $O(n)$
Input:
n. Number of vertices. $1 \leq n < 100$.
P. Array of points that conform the polygon.
p. Point that we want to know if is inside the polygon.
Output:
function `isInsidePolygon` returns 1 if p is inside, otherwise return 0.

Listing 8.1: Point Inside a Convex Polygon

```
#define N 101

class Point {
 public:
  long x;
  long y;

  Point(long x = 0, long y = 0) {
    this->x = x;
    this->y = y;
  }
};

long n; // Number of vertices
Point P[N]; // Polygon

long area(Point a, Point b, Point c) {
  return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
}

long isInsidePolygon(Point p) {
  // Vertices should be sorted and P[n] = P[0]
  for (long i = 0; i < n; i++) {
    if (area(p, P[i], P[i + 1]) <= 0) {
      return 0;
    }
  }
  return 1;
}
```

8.1.2 Point in Polygon

Given a point z and the line segments that conform the polygon (in any specific order), this algorithm checks if z is inside, in the

border, or outside the polygon.

The idea of the algorithm is to draw an imaginary vertical line starting from point z, if the number of crossed lines is odd, then it is inside the polygon, if it is even, it is outside the polygon. If there is a line segment that contains z, then the point is in the border. See image 8.2.

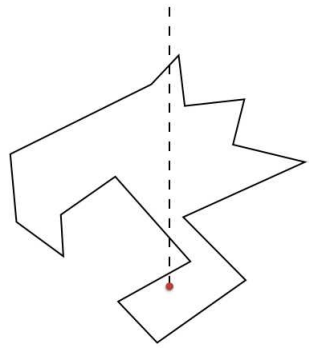


Figure 8.2: The imaginary line crosses three line segments. We can assure that the red point is inside the polygon.

To identify if a point is above or below a line, we can use equation 8.5 to obtain the signed area of the triangle formed by the point and the coordinates of the line segment, if is negative then the point is below, if is positive is above, and if it is zero, then the point is on the border.

The program 8.2 reads one number n, indicating the number of line segments that conforms the polygon. The following n lines contain 4 numbers x_1, y_1, x_2, y_2, that represents the coordinates of the end points of a line segment formed by (x_1, y_1) and (x_2, y_2). The last line contains two numbers x, y that represents the coordinates of the point p. The output is a message indicating if p is inside, outside or in the border of the polygon.

Time Complexity: $O(n)$
Input:
n. Number of line segments. $1 \leq n < 10000$.
line. Array of line segments
p. Point that we want to know if is inside the polygon.

Output:

The message "BORDER" if p is in the border of the polygon, "OUTSIDE" if it is outside the polygon, and "INSIDE" if it lies inside the polygon.

Listing 8.2: Point Inside a Polygon

```
#include <cstdio>
#define N 10001
using namespace std;

enum STATE { OUTSIDE,
             BORDER,
             INSIDE };

typedef struct stPoint {
  long x;
  long y;
} Point;

typedef struct stLine {
  Point A;
  Point B;
} Line;

Line line[N];
long n;

long area(Point, Point, Point);

int main() {
  long k, aux, c;
  Point temp, p;

  scanf("%ld", &n);
  for (long i = 0; i < n; i++) {
    scanf("%ld %ld %ld %ld", &line[i].A.x, &line[i].A.y, &line[i].B.x,
          &line[i].B.y);

    if (line[i].A.x > line[i].B.x) {
      temp = line[i].A;
      line[i].A = line[i].B;
      line[i].B = temp;
    }
  }

  scanf("%ld %ld", &p.x, &p.y);

  c = 0;
  aux = 0;
  for (long i = 0; i < n; i++) {
    if (p.x >= line[i].A.x && p.x < line[i].B.x) {
      k = area(line[i].A, line[i].B, p);
      if (k > 0) {
        c++;
      } else if (k == 0) {
        aux = 1;
        break;
      }
    }
  }
```

```
54
55    if (aux == 1) {
56      printf("BORDER\n");
57    } else if (c % 2 == 0) {
58      printf("OUTSIDE\n");
59    } else {
60      printf("INSIDE\n");
61    }
62
63    return 0;
64  }
65
66  long area(Point a, Point b, Point c) {
67    return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
68  }
```

8.1.3 Point Inside a Triangle

Calculate the areas formed by point z and the triangle vertices. If the sum of the three areas is equal to the area of the triangle, then z is inside the triangle. See image 8.3. To calculate the area of a triangle formed by three points we can use equation 8.5.

The Function `isInsideTriangle` in 8.3 receives a point `p0` and an array of points `v` that conforms the triangle. The function returns 1 if `p0` is is inside the triangle, otherwise return 0.

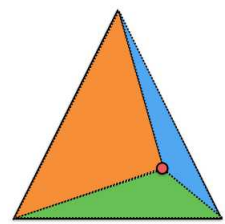


Figure 8.3: The sum of the colored areas is equal to the area of the triangle, indicating that the red point is inside the triangle.

Time Complexity: $O(1)$
Input:
`p0`. Given point.
`v`. Array of three points representing the vertices of the triangle.
Output:
The function `isInsideTriangle` returns 1 if `p0` is inside the triangle, otherwise returns 0.

Listing 8.3: Point Inside a Triangle

```
int isInsideTriangle(Point p0, Point *v) {
  if (fabs(area(v[0], v[1], p0) + area(v[0], v[2], p0) + area(v[2], v[1],
       p0) - At) < EPS) {
    return 1;
  }
  return 0;
}

double area(Point a, Point b, Point c) {
  return fabs(a.x * c.y + b.x * a.y + c.x * b.y - b.x * c.y - c.x * a.y -
       a.x * b.y);
}
```

8.2 Area of a Polygon

Given a polygon with n vertices numbered from 0 to $n-1$ sorted counterclockwise, the area of the polygon is given by the following expression.

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i-1}y_i - x_i y_{i-1}) \tag{8.6}$$

With $x_n = x_0$, and $y_n = y_0$.

The function `areaPolygon` in 8.4 calculates the area of the polygon `poly` of n vertices using equation 8.6.

Time Complexity: $O(n)$
Input:
`n`. Number of vertices of the polygon.
`poly`. Array of points that represents the vertices of the polygon.
Output:
The function `areaPolygon` returns the area of the polygon.

Listing 8.4: Area of a Polygon

```
double areaPolygon(Point *poly, int n) {
  double A = 0.0;

  for (int i = 1; i <= n; i++) {
    A = A + poly[i - 1].x * poly[i].y - poly[i].x * poly[i - 1].y;
  }
  return A * 0.5;
}
```

8.3 Line Intersection

Line intersection is a common problem in geometry but still there are some problems that can be present at the moment to implement a programmatic solution. For example, the program must find out if the lines are parallel, or calculating the slope of a vertical line.

Algorithm 1

The code in 8.5 receives four points a, b, c, and d, where ab represents one line segment, and cd another line segment. It returns 1 if the line segments intersect each other, otherwise return 0. In case they intersect, the point where they crossed is stored in `p0`.

Basically what this algorithm does is try to find a solution for the system of equations for two lines of the form $ax + by + c = 0$.

$$a_1x_1 + b_1y_1 + c_1 = 0$$
$$a_2x_2 + b_2y_2 + c_2 = 0$$

The line equation given two points on the line is given by:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) \tag{8.7}$$

The function `intersection` in 8.5 receives two line segments and determines if those line segments intersect each other.

Time Complexity: $O(1)$
Input:
`a,b,c,d`. Two line segments, one formed by points ab and other by cd.
Output:
The function `intersection` returns 0 if the lines don't intersect each other, otherwise returns 1 and stores the crossing point in `p0`.

Listing 8.5: Line Intersection 1

```
1 int intersection(Point a, Point b, Point c, Point d, Point &p0) {
2   double Ax0 = min(a.x, b.x);
3   double Bx0 = min(c.x, d.x);
4   double Ax1 = max(a.x, b.x);
5   double Bx1 = max(c.x, d.x);
6   double Ay0 = min(a.y, b.y);
```

```
  double By0 = min(c.y, d.y);
  double Ay1 = max(a.y, b.y);
  double By1 = max(c.y, d.y);

  double a1 = b.y - a.y;
  double b1 = a.x - b.x;
  double c1 = b.x * a.y - a.x * b.y;

  double a2 = d.y - c.y;
  double b2 = c.x - d.x;
  double c2 = d.x * c.y - c.x * d.y;

  double den = a1 * b2 - a2 * b1;
  if (fabs(den) < 0.000001) {
    return 0;
  }

  p0.x = (b1 * c2 - b2 * c1) / den;
  p0.y = (a2 * c1 - a1 * c2) / den;

  if (p0.x >= Ax0 && p0.x <= Ax1 && p0.x >= Bx0 && p0.x <= Bx1) {
    if (p0.y >= Ay0 && p0.y <= Ay1 && p0.y >= By0 && p0.y <= By1) {
      return 1;
    }
  }

  return 0;
}
```

Algorithm 2

The function `intersection` in 8.6 receives four points a, b, c, and d, where ab represents one line segment, and cd another line segment. The function returns 1 if the line segments intersect each other, -1 if they don't intersect, and 0 if they have any collinear points.

The idea behind this algorithm is to use signed triangle areas (8.5) to check if there is a crossing point, for this we take one point of one line segment and obtain the signed triangle area using the points of the other segment, we do the same for the other point, and the signs of both areas must be different if the lines cross each other. See figure 8.4. We do the same for the points in the other segment.

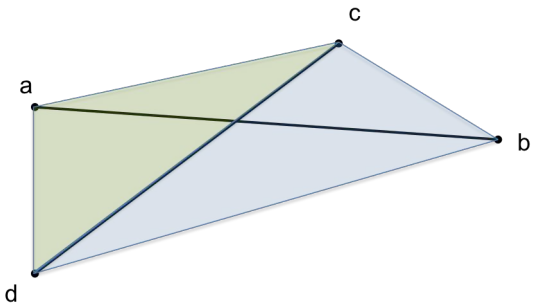


Figure 8.4: The signed area of the green triangle (acd) have a different sign to the one of the blue triangle (bcd).

The function `intersection` implemented in 8.6 receives two line segments and using the algorithm described above determines if those line segments intersect each other.

Time Complexity: $O(1)$
Input:

`a,b,c,d`. Two line segments, one formed by points ab and other by cd.

Output:

The function `intersection` returns 1 if the lines intersect each other, 0 if and end point of one segment is over the other line segment, and -1 if the two line segments don't intersect each other.

Listing 8.6: Line Intersection 2

```
int intersection(Point a, Point b, Point c, Point d) {
  long A1 = area(c, a, b) * area(d, a, b);
  long A2 = area(a, c, d) * area(b, c, d);

  if (A1 < 0 && A2 < 0) {
    return 1;
  } else if (A1 < 0 && A2 == 0) {
    return 0;
  } else if (A2 < 0 && A1 == 0) {
    return 0;
  } else if (A1 == 0 && A2 == 0) {
    return 0;
  }

  return -1;
}
```

8.4 Horner's Rule

A polynomial in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j. \tag{8.8}$$

We call the values $a_0, a_1, \cdots, a_{n-1}$ the coefficients of the polynomial.

The operation of evaluating the polynomial $A(x)$ at a given point x_0 consists of computing the value of $A(x_0)$. We can evaluate a polynomial in $O(n)$ time using Horner's rule:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots)) \tag{8.9}$$

8.5 Centroid of a Convex Polygon

Given a convex polygon defined by n vertices $(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})$, sorted counterclockwise. The coordinates of the centroid C (center of mass) is given by:

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \tag{8.10}$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \tag{8.11}$$

where A is the area of the polygon defined in 8.6, and vertex (x_n, y_n) is assumed to be the same as (x_0, y_0).

8.6 Convex Hull

The convex hull of a set of points X is the convex polygon of minimum area that encloses X. See image 8.5. In this section we will analyze two different algorithms to find the convex hull of a set of points, both based on triangle areas.

8.6.1 Andrew's Monotone Convex Hull Algorithm

Invented by A. M. Andrew in 1979 [25]. Given a cloud of n points, this algorithm sort the points by their x-coordinate, and in case of a tie, by y-coordinate, and then constructs the upper and lower hulls. The upper hull is represented by the red line in figure 8.5. While the lower hull is the remaining part of the convex hull displayed as a blue line.

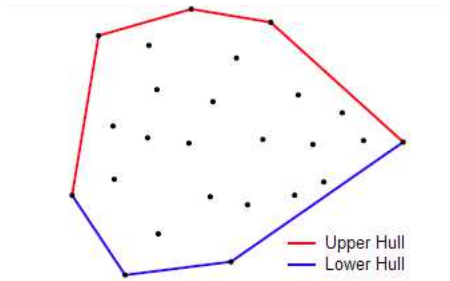
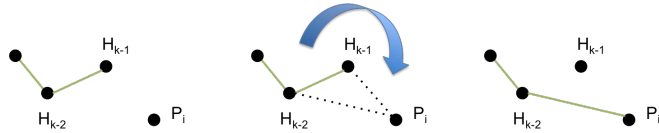


Figure 8.5: Convex hull with the upper hull in red, and the lower hull in blue.

To build the lower hull we move through the array of points P adding every new point to the lower hull, this is valid because the points were previously sorted, then we need to remove all previously added points that don't form a convex polygon with the new point as reference using triangle areas. Line 29 in program 8.7 removes from the convex hull the point H_{k-1} if it forms a "right turn" (negative triangle area) with points H_{k-2}, H_{k-1} and P_i. Figure 8.6 shows the process of how a new point is added to the convex hull causing that some points to be removed. To construct the upper hull the process is the same but moving backwards in the array of points.



(a) Point P_i is added to the convex hull.

(b) There is a right turn (clockwise direction) between points H_{k-2}, H_{k-1} and P_i.

(c) Point H_{k-1} is removed from the convex hull.

Figure 8.6: Process of adding a point into the convex hull. The green line represents the convex hull.

Sorting the points takes $O(n \log n)$ and traversing the array of points to construct the upper and lower hull takes $O(n)$, then the complexity of the algorithm depends on the sorting algorithm used. The function `convexHull` in 8.7 receives a vector of points P, and returns a vector of points H with the points of P that are part of the convex hull. The function `cross` computes the signed area of a triangle given its vertices.

Time Complexity: $O(n \log n)$
$O(n \log n)$. Sort
$O(n)$. Build the upper and lower hull.
Input:
`P`. Vector of points.
Output:
`H`. Vector of points that form the convex hull.

Listing 8.7: Andrew's Convex Hull

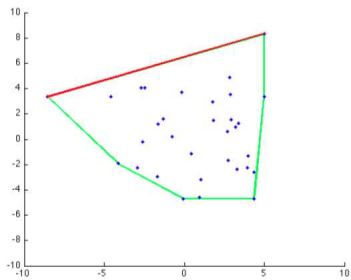
```
class Point {
 public:
  int x;
  int y;

  Point(int x = 0, int y = 0) {
    this->x = x;
    this->y = y;
  }

  bool operator<(const Point &p) const {
    return this->x < p.x || (this->x == p.x && this->y < p.y);
  }
};

int cross(const Point &O, const Point &A, const Point &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}
```

```
19
20  vector<Point> convexHull(vector<Point> P) {
21    int n = P.size(), k = 0;
22    vector<Point> H(2 * n);
23
24    // Sort points by x-coordinate first, and by y-coordinate second
25    sort(P.begin(), P.end());
26
27    // Build lower hull
28    for (int i = 0; i < n; ++i) {
29      while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) < 0) {
30        k--;
31      }
32      H[k++] = P[i];
33    }
34
35    // Build upper hull
36    for (int i = n - 2, t = k + 1; i >= 0; i--) {
37      while (k >= t && cross(H[k - 2], H[k - 1], P[i]) < 0) {
38        k--;
39      }
40      H[k++] = P[i];
41    }
42
43    H.resize(k);
44    return H;
45  }
```

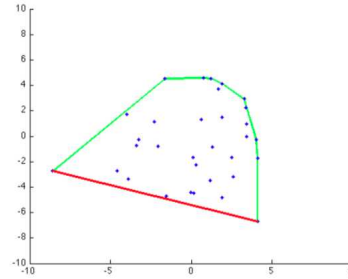
8.6.2 Graham's Scan

Named after Ronald L. Graham, who described it in 1972 [26]. Graham's Scan is an algorithm that finds the convex hull of a set of n points, but in order to use it the points must be sorted counterclockwise. This can be done in $O(n \log n)$ time.

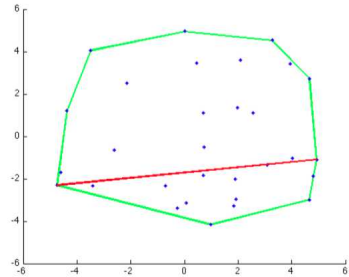
To sort the set of points, first we need to find two pivot points, P_l, and P_r, that corresponds to the most left point and the most right point respectively, if there are more than one point, choose the one with lowest y-coordinate. Then an imaginary straight line is traced between these two points, resulting in three possible cases. See image 8.7.



(a) All points below the line



(b) All points above the line



(c) Points above and below the line

Figure 8.7: Cases to sort a set of points

In each of the three cases the way in which the points must be sorted change, that's why is important to identify on which of the three cases we are. To sort the points is easier to use one of the pivot points to calculate triangle areas, we can use any sorting algorithm, the code in 8.8 use the `sort` function of the `algorithm` library.

Once we know in which case we are and the points have been sorted according to that, the next step consists on applying the Graham's Scan to obtain the convex hull, this is done in $O(n)$ time.

Graham's scan starts by adding the first two points to the convex hull H. The following points are added to the convex hull, but each time a point is added we need to remove all those points that form a "left turn" (positive triangle area) with their previous point and the recently added point. This process is similar to the

one used in Andrew's algorithm showed in figure 8.6. Basically we need to remove the point H_i from the convex hull if the signed area of the triangle formed by H_i, H_{i-1} and P_i is positive.

The code in 8.8 read a number n representing the number of points, then n pair of numbers follow giving the coordinates of the points. The program prints the number of points in the convex hull and the coordinates of those points.

Time Complexity: $O(n \log n)$
$O(n \log n)$. Sort
$O(n)$. Graham's Scan
Input:
`n`. Number of points
`point`. Array of coordinates
Output:
`stack`. Array of point index that form the convex hull.

Listing 8.8: Graham's Scan

```
#include <algorithm>
#include <cstdio>
#define N 1001

using namespace std;

typedef struct stPoint {
  long x;
  long y;
} Point;

Point point[N];
Point LH[N], UH[N], ZH[N];
Point pl, pr;
long stack[N];

bool compare(Point, Point);
long area(Point, Point, Point);
void sortPoints(long);
long convexHull(long n);

int main() {
  long n, m;

  scanf("%ld", &n);
  for (long i = 0; i < n; i++) {
    scanf("%ld %ld", &point[i].x, &point[i].y);
  }

  sortPoints(n);
  m = convexHull(n);

  printf("%ld\n", m);
  for (long i = 1; i <= m; i++) {
```

```
    printf("%ld %ld\n", point[stack[i]].x, point[stack[i]].y);
  }

  return 0;
}
```

Inside the `sortPoints` function each point is assigned to its corresponding hull, and then those hulls are sorted counterclockwise using the `sort` function of the *algorithm* library. After that all the points from every hull are merged into a single array according to the cases described in figure 8.7.

```
void sortPoints(long n) {
  long pos1, pos2;
  long n1, n2, n3;
  long A;

  pos1 = pos2 = 0;

  for (long i = 1; i < n; i++) {
    if (point[i].x < point[pos1].x) {
      pos1 = i;
    } else if (point[i].x == point[pos1].x && point[i].y < point[pos1].y) {
      pos1 = i;
    }

    if (point[i].x > point[pos2].x) {
      pos2 = i;
    } else if (point[i].x == point[pos2].x && point[i].y < point[pos2].y) {
      pos2 = i;
    }
  }

  n1 = n2 = n3 = 0;
  pl = point[pos1];
  pr = point[pos2];

  for (long i = 0; i < n; i++) {
    A = area(pl, point[i], pr);

    if (A > 0) {
      LH[n1++] = point[i];
    } else if (A < 0) {
      UH[n2++] = point[i];
    } else {
      ZH[n3++] = point[i];
    }
  }

  sort(LH, LH + n1, compare);
  sort(UH, UH + n2, compare);
  sort(ZH, ZH + n3, compare);

  // Merge LH, ZH and UH
  n = 0;
  if (n1 == 0) {
    for (long i = 0; i < n3; i++) {
      point[n++] = ZH[i];
    }
```

```
    for (long i = 0; i < n2; i++) {
      point[n++] = UH[i];
    }
  } else if (n2 == 0) {
    point[n++] = pl;
    for (long i = 0; i < n1; i++) {
      point[n++] = LH[i];
    }
    for (long i = n3 - 1; i > 0; i--) {
      point[n++] = ZH[i];
    }
  } else {
    point[n++] = pl;
    for (long i = 0; i < n1; i++) {
      point[n++] = LH[i];
    }
    for (long i = 1; i < n3; i++) {
      point[n++] = ZH[i];
    }
    for (long i = 0; i < n2; i++) {
      point[n++] = UH[i];
    }
  }
}
```

Finally the function `convexHull` performs the Graham's scan and stores in `stack` the indexes of the points that are part of the convex hull.

```
long convexHull(long n) {
  long ps;

  stack[1] = 0;
  stack[2] = 1;
  stack[3] = 2;
  ps = 3;

  for (long i = 3; i < n; ++i) {
    while (area(point[stack[ps]], point[stack[ps - 1]], point[i]) > 0) {
      ps--;
      if (ps == 1) {
        break;
      }
    }

    ps++;
    stack[ps] = i;
  }

  if (area(point[stack[ps]], point[stack[ps - 1]], point[stack[1]]) > 0) {
    ps--;
  }

  stack[ps + 1] = stack[1];
  ps++;
  return ps;
}
```

The functions `compare` and `area` are auxiliary functions. The first one specify the rules used when two points are compared, allowing that the points can be sorted counterclockwise. Meanwhile the `area` function receives three points and returns the signed area of the triangle formed by them.

```
bool compare(Point sp1, Point sp2) {
  long k, d1, d2, A;

  k = area(pl, sp1, sp2);
  if (k > 0) {
    return true;
  } else if (k == 0) {
    d1 = (sp1.x - pl.x) * (sp1.x - pl.x) + (sp1.y - pl.y) * (sp1.y - pl.y);
    d2 = (sp2.x - pl.x) * (sp2.x - pl.x) + (sp2.y - pl.y) * (sp2.y - pl.y);
    A = area(pl, sp1, pr);

    if (A < 0) {
      if (d1 < d2) {
        return false;
      } else {
        return true;
      }
    } else {
      if (d1 < d2) {
        return true;
      } else {
        return false;
      }
    }
  } else {
    return false;
  }
}

long area(Point a, Point b, Point c) {
  return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
}
```

8.7 Chapter Notes

Geometric algorithms are sometimes difficult to analyze and also difficult to implement, since there are things to consider, like the precision error, divisions by zero, variable overflow while doing calculations, et. al. For that reason when coding a solution for a geometry problem, is important to keep operations as simple as possible. There are a great amount of tips and tricks in the forums of online judges, we recommend to take a look to the ideas and solutions shared there for geometric problems.

There are a lot of great books about geometry and geometric

algorithms, some of the ones we used as reference while writing this book were: *"Analytic Geometry"* by Lehmann [27], that explains very clearly the basic concepts about geometry. *"Introduction to Algorithms"* by Cormen, Leiserson, Rivest, and Stein [1], describing the performance and behavior of some of the most known geometric algorithms. *"Introduction to the Design and Analysis of Algorithms"* by Lee, Tseng, Chang, and Tsai [12], which share interesting ideas and analysis of geometric algorithms.

The following links contain a list of problems about geometry that are worth to try.

- `https://www.urionlinejudge.com.br/judge/es/problems/index/8`
- `http://acm.timus.ru/problemset.aspx?space=1&tag=geometry`
- `https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=452`

8.8 Exercises

1. Three points are collinear if a line crosses trough all of them. Given a list of n points, $(3 \leq n \leq 100)$. Write a program that finds the maximum amount of points that are collinear. All the coordinates are integer values.

2. A hawk is looking for his favorite food, gophers. Given the location of the hawk, his attack radius, and the location of all gophers. Write a program that finds how many gophers are in danger. A gopher is in danger if the distance between him and the hawk is less or equal than the hawk's attack radius. The number of gophers is at most 100. All coordinates are integer values.

3. Given the lengths of the sides of a triangle, how can we obtain the triangle's area?

4. The city hall has a closed polygonal shape. Is it possible to place security cameras in such a way that the entire hall can be watched? Describe an algorithm that finds the solution? You can assume that cameras can rotate 360^o.

5. For problem 4. Find an upper bound for the cameras needed to watch over the city hall? Explain your answer.

6. Describe an algorithm to find the minimum distance from a point to a polygon. Both the point an the polygon are given as input.

9

Number Theory and Combinatorics

> *"A smart man makes a mistake, learns from it, and never makes that mistake again. But a wise man finds a smart man and learns from him how to avoid the mistake altogether."*
>
> – Roy H. Williams

During this chapter we will see some properties that numbers have and that most of the time remain unknown. We will put emphasis on prime numbers given their importance and the application they have for different problems. We will also cover modular arithmetic that allow us to be able to make calculations without worrying about memory overflow. Another important topic is divisibility which includes the Euclidean algorithm, that finds the greatest common divisor *(gcd)* of two numbers. There are a lot of properties and hidden tricks inside numbers and it will take a whole book, or more, to include all of them, but we will focus on those we think are the basis in computer programming.

Here is a small example of the tricks that one can find in number theory. Suppose we have a number n and we want to write a program that identifies if n is divisible by 3. The first choice is to obtain the modulus of $n/3$, if it is zero, then n is divisible by 3. Easy right? To make it more interesting what if n contains 100000 digits? Well that makes the things more interesting but not impossible, we just need to make a division

as we were taught in school, or better, we can use the following property: *"A number is divisible by three if the sum of its digits is divisible by three."* Then if n contains 100000 digits, then the sum of its digits it a most 900000, which easily fits in an integer variable.

Another example of a clever trick is how to calculate the sum the first n numbers. Suppose that $n = 100$, then we can start calculating the sum $1+2+3+4+5+6+\ldots+100$. I don't know you, but I lost track of the count quickly. Well, Gauss, the German mathematician thought in an easier approach, start adding the first number and the last one and get $1+100 = 101$, then add the second with the second last and get $2 + 99 = 101$ again, the next would be $3 + 98 = 101$ once again, and so on. The sum is always 101, and there are 50 pairs, then the sum of the first 100 numbers is 50×101. Generalizing this approach we have that the sum S of the first n numbers is given by:

$$S = \frac{n(n+1)}{2} \tag{9.1}$$

These are only some examples of tricks and properties that numbers have, and like these there are a lot of them, it is incredible the amount of secrets that numbers hide. The goal of this chapter is that the reader learn to take advantage of those secrets, properties and tricks in order to solve complex problems.

9.1 Prime Numbers

A prime number is a number that is only divided by 1 and by itself. Prime numbers are of great importance, and they can be used to solve different problems. That's is why it is useful to have an efficient way to generate prime numbers.

In this section we will review some algorithms to generate prime numbers, and study some problems where prime numbers are needed to implement the solution.

To check if a number n is a prime number we only need to generate the prime numbers up to the square root of n, if one of them divide n, then n is not a prime number.

Suppose n can be expressed as the product of two prime numbers p and q, then

$$n = pq$$

where $p \leq q$, then

$$\begin{aligned} p &\leq q \\ pp &\leq qp \\ p^2 &\leq n \\ p &\leq \sqrt{n} \end{aligned} \tag{9.2}$$

9.1.1 Sieve of Eratosthenes

One of the most used algorithms to generate prime numbers. Builds a 0/1 array P, where $P_k = 0$ if k is a prime number, and $P_K = 1$ if k is not a prime number. The first ten elements looks like this:

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

To find all the prime numbers less than or equal to N, we need an array P of $N + 1$ elements, $P_0, \ldots, P_N$. We can start marking P_0 and P_1 with 1, since we know they are not prime numbers. After that, we go trough all the numbers starting from 2 up to $\sqrt{N}$. If P_i is zero then i is a prime number, so we mark with 1 all its multiples starting from i^2. At the end, all prime numbers will be marked with 0, and the rest with 1.

The code in 9.1 implements the Sieve of Eratosthenes to find the prime numbers up to 100 and print the resulting array.

Time Complexity: $O(n \log \log n)$
Input:
`n`. An integer.
Output:
`P`. Sieve with the primes numbers up to N.

Listing 9.1: Sieve of Eratosthenes

```
1 #include <cstdio>
2 #include <cmath>
3 #define n 100
4 using namespace std;
5
```

```
int P[n + 1];
void sieve();

int main() {
  sieve();
  for (int i = 0; i <= n; i++) {
    printf("%d\n", P[i]);
  }
  return 0;
}

void sieve() {
  int lim = (int)sqrt(double(n));

  for (int i = 2; i <= lim; i++) {
    if (P[i] == 0) {
      for (int j = i * i; j <= n; j += i) {
        P[j] = 1;
      }
    }
  }
  P[0] = 1;
  P[1] = 1;
}
```

9.1.2 Prime Number Generator

In some problems we need to store all the prime numbers up to some given number n in an array, so we can easily get any prime number. The algorithm 9.2 starts by adding 2 to the vector of primes, which is initially empty. Then we go trough all odd numbers starting from 3 up to n and check for all them if there is a prime number that divides it, if there is none, then it is a prime number and it should be added to the vector of primes.

To know if some number k is prime, we should check for all prime less than or equal to $\sqrt{k}$ if none of them divides it, then k is a prime number.

We only look up to $\sqrt{k}$, because according to 9.2 every non-prime number m can be expressed as the product of two numbers, p and q, and one of them will be always less or equal than $\sqrt{m}$.

The algorithm in 9.2 computes all prime numbers less or equal to a given number n. The time complexity is not clear, since for each number num we have to search for any divisor up to $\sqrt{num}$, and that would run in $O(\sqrt{n})$, but there is no need no check all numbers, just the prime numbers which are much less, and because the amount of primes changes through the process is hard

to define an upper bound for the running time, so for simplicity we would define an upper bound of $O(\sqrt{n})$, and since we have to search divisors for all numbers the time complexity would be $O(n\sqrt{n})$. We should keep in mind that the function `sqrt` is an estimate of the square root and it implies an internal procedure that can take some time, so one thing that can be done is to rise to the power of two both sides of the inequality, so line 8 would look like this

```
else if(num < P[i]*P[i]).
```

This way we will avoid computation of the square root and speed up the running time of the algorithm, we just need to be careful that the product `p[i]*p[i]` doesn't cause and overflow.

Time Complexity: $O(n\sqrt{n})$
Input:
`n`. An integer.
Output:
`P`. Vector with prime numbers smaller or equal to n.

Listing 9.2: Prime Number Generator

```
1  void generatePrimes() {
2    P.push_back(2);
3    P.push_back(3);
4    for (int num = 5; num <= n; num += 2) {
5      for (int i = 0; i < P.size(); i++) {
6        if (num % P[i] == 0) {
7          break;
8        } else if (sqrt((double)num) < (double)P[i]) {
9          P.push_back(num);
10         break;
11       }
12     }
13   }
14 }
```

9.1.3 Euler's Totient Function

Euler's totient function or Euler's $\phi(n)$ function, counts the number of integers in $a = 1, 2, \cdots, n$ such that $gcd(a, n) = 1$. In other words, counts the number of positive integers equal or smaller than n that are relatively prime to n.

Be $P(n) = p_1, \cdots, p_m$ the set of all prime factors of n, the

Euler's function is defined by:

$$\phi(n) = n \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right) \tag{9.3}$$

The Euler's function for the first ten numbers are:

$n =$	1	2	3	4	5	6	7	8	9	10
$\phi(n) =$	1	1	2	2	4	2	6	4	6	4

The Euler's function is multiplicative, meaning that if $gcd(a, b) = 1$, then $\phi(ab) = \phi(a)\phi(b)$.

Every number n can be expressed as the product of its prime factors.

$$n = p_1^{k_1} \times p_2^{k_2} \times \cdots \times p_m^{k_m}$$

,and because $p_i^{k_i}$ has exactly $p_i^{k_i-1}$ multiples less than or equal to $p_i^{k_i}$ $(p_i, 2p_i, \ldots, p_i^{k_i-1}p_i)$, the number of relatively prime numbers to $p_i^{k_i}$ is given by:

$$\begin{aligned}
\phi(p_i^{k_i}) &= p_i^{k_i} - p_i^{k_i-1} \\
&= p_i^{k_i-1}(p_i - 1) \\
&= p_i^{k_i}\left(1 - \frac{1}{p_i}\right)
\end{aligned}$$

then

$$\begin{aligned}
\phi(n) &= \phi(p_1^{k_1} \times p_2^{k_2} \times \cdots \times p_m^{k_m}) \\
&= \phi(p_1^{k_1}) \times \phi(p_2^{k_2}) \times \phi(p_m^{k_m}) \\
&= p_1^{k_1}\left(1 - \frac{1}{p_1}\right) p_2^{k_2}\left(1 - \frac{1}{p_2}\right) \cdots p_m^{k_m}\left(1 - \frac{1}{p_m}\right) \\
&= p_1^{k_1} p_2^{k_2} \cdots p_m^{k_m}\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_m}\right) \\
&= n \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right)
\end{aligned}$$

9.1.4 Sum of Divisors

Given an integer n, calculate the sum of all divisors of n. For example, for $n = 6$ there are four divisors: 1, 2, 3, and 6, the sum of all of them is $1 + 2 + 3 + 6 = 12$

A number n can be written by the product of its prime factors $(p_1, \cdots, p_m)$.

$$n = \prod_{i=1}^{m} p_i^{k_i}, \tag{9.4}$$

where m is the number of different prime factors, and k_i is the number of times p_i divides n. For example, $12 = 2^2 \times 3^1$. Here $p_1 = 2, p_2 = 3$, and $k_1 = 2, k_2 = 1$.

Be $s(n)$ the sum of all divisors of n and is defined by:

$$\begin{aligned} s(n) &= s\left(\prod_{i=1}^{m} p_i^{k_i}\right) \\ &= \prod_{i=1}^{m} s\left(p_i^{k_i}\right) \\ &= \prod_{i=1}^{m} \left(1 + p_i + p_i^2 + \cdots + p_i^{k_i}\right) \\ &= \prod_{i=1}^{m} \frac{p_i^{k_i+1} - 1}{p_i - 1} \end{aligned} \tag{9.5}$$

Some examples:

$$\begin{aligned} s(12) = s(2^2 3^1) &= \frac{2^3 - 1}{2 - 1} \times \frac{3^2 - 1}{3 - 1} \\ &= \frac{7}{1} \times \frac{8}{2} = 7 \times 4 \\ &= 28 = 1 + 2 + 3 + 4 + 6 + 12 \end{aligned}$$

$$\begin{aligned} s(8) = s(2^3) &= \frac{2^4 - 1}{2 - 1} \\ &= 15 = 1 + 2 + 4 + 8 \end{aligned}$$

$$s(36) = s(2^2 3^2) = \frac{2^3 - 1}{2 - 1} \times \frac{3^3 - 1}{3 - 1}$$
$$= 91 = 1 + 2 + 3 + 4 + 6 + 9 + 12 + 18 + 36$$

Listing 9.3: Sum of Divisors

```
int sumOfDivisors(int n) {
  int p = 0, k = 0;
  int res = 1, f = 1;
  int prev = n;

  if (n == 0 || n == 1) {
    return n;
  }

  while (n >= P[k] * P[k]) {
    if (n % P[k] == 0) {
      prev = P[k];
      n /= P[k];
      f *= P[k];
    } else {
      if (f > 1) {
        f *= P[k];
        res *= (f - 1) / (P[k] - 1);
      }
      k++;
      f = 1;
    }
  }

  if (n == prev) {
    f *= n * n;
    res *= (f - 1) / (n - 1);
  } else {
    if (f > 1) {
      f *= prev;
      res *= (f - 1) / (prev - 1);
    }

    res *= (n * n - 1) / (n - 1);
  }

  return res;
}
```

9.1.5 Number of Divisors

As we've seen, a number n can be expressed as the product of its prime factors

$$n = \prod_{i=1}^{m} p_i^{k_i}$$

be $D(n)$ the number of divisors of n, then

$$\begin{aligned} D(n) &= D\left(\prod_{i=1}^{m} p_i^{k_i}\right) \\ &= \prod_{i=1}^{m} D\left(p_i^{k_i}\right) \\ &= \prod_{i=1}^{m} (k_i + 1) \end{aligned} \tag{9.6}$$

9.2 Modular Arithmetic

It's very common to face problems that involve large integers, or handling operations that causes an overflow error. *Modular Arithmetic* is a technique that allow us to simplify those operations, keeping the results in integers that can represented by the computer.

Given a positive integer m, called the *modulo*, the goal is to obtain the remainder of a large number, product of some calculations, when divided by m. e.g. Consider n 32-bit integers, $a_0, a_1, \ldots, a_{n-1}$. What would be the remainder of the sum of all n numbers when divided by some "small" integer m?

$$(a_0 + a_1 + \cdots + a_{n-1}) \bmod m = ?$$

Now, we know the result will be in the range $[0, m-1]$, but the sum of all numbers can be very large, and that can cause an overflow error. Here is when we take advantage of one of the properties of modular arithmetic that states:

$$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m, \tag{9.7}$$

which causes that all operations be in the range $[0, m-1]$, avoiding that way any possible overflow error. To simplify things we are going to denote $a \bmod m$, as $[a]$, for some integers a and m, where $m > 0$. The we can rewrite equation 9.7 as

$$[a + b] = [[a] + [b]]. \tag{9.8}$$

We also can apply modular arithmetic for subtraction and multiplication, which are defined as follows:

$$[a - b] = [[a] - [b]] \tag{9.9}$$
$$[a \times b] = [[a] \times [b]] \tag{9.10}$$

The exponentiation on the other hand is defined by

$$[a^b] = [[a]^b]. \tag{9.11}$$

Notice that the modulo is not applied to the exponent, since $[a^b] \neq [[a]^{[b]}]$. For example, consider the case where $a = 2, b = 4$, and $m = 3$. For one side we have that $[2^4] = 1$, of for the other side we have that $[[2]^{[4]}] = [2^1] = 2 \neq 1$.

The program in 9.4 reads n numbers and prints their sum modulo m by using the addition property of modular arithmetic.

Time Complexity: $O(n)$
Input:
`n`. The number of integers to read.
`m`. The modulus.
`num`. Variable used to read the n numbers.
Output:
`sum`. The sum of all n numbers in the input modulus m.

Listing 9.4: Modular Arithmetic

```
#include <cstdio>
using namespace std;

int main() {
  int n, m, num, sum;

  scanf("%d %d", &n, &m);
  sum = 0;
  for (int i = 0; i < n; i++) {
    scanf("%d", &num);
    sum = (sum + num % m) % m;
  }

  printf("%d\n", sum);
  return 0;
}
```

9.3 Euclidean Algorithm

Euclid's algorithm is an efficient method of calculating the greatest common divisor *(gcd)* of two numbers. It was published in Book VII of Euclid's Elements around 300 BC, and is based in on the following simple observation.

$$gcd(a, b) = gcd(b, r), \tag{9.12}$$

where $a \geq b$ and r is the reminder of dividing a by b. e.g. Calculate the greatest common divisor of 18 and 14. By 9.12 we get

$$\begin{aligned} gcd(18, 14) &= gcd(14, 4) \\ gcd(14, 4) &= gcd(4, 2) \\ gcd(4, 2) &= gcd(2, 0) \end{aligned}$$

Once the smallest value is zero, we stop the search and the result is the largest value. For our example the greatest common divisor of 18 and 14, is 2. The function `gcd` in 9.5 receives two positive integers a and b, then makes sure a to be larger or equal than b and executes the *Euclidean Algorithm*. The function returns the greatest common divisor of the two numbers given.

Time Complexity: The worst case scenario to obtain the value of *gcd(a,b)* is when a and b are consecutive *Fibonacci numbers*. If a is the n^{th} *Fibonacci number*, the *Euclidean algorithm* needs n iterations to get to the result.

Input:

The function `gcd` receives two positive integers a and b.

Output:

The function `gcd` returns the *gcd* of a and b.

Listing 9.5: Euclidean Algorithm

```
1  int gcd(int a, int b) {
2    if (a < b) {
3      swap(a, b);
4    }
5
6    while (b > 0) {
7      int temp = a;
8      a = b;
9      b = temp % b;
10   }
```

```
11
12    return a;
13  }
```

9.3.1 Extended Euclidean Algorithm

Given two integers a and b find the values of x and y such that

$$ax + by = gcd(a, b) \tag{9.13}$$

To find the values of x and y we can use the Euclidean algorithm. Using 9.12 we know that

$$gcd(a, b) = gcd(b, a \bmod b)$$

Making $a_1 = a, b_1 = b, a_2 = b$ and $b_2 = a \bmod b$ we have that

$$gcd(a_1, b_1) = gcd(a_2, b_2)$$

Using 9.13 we get the following:

$$\begin{aligned} gcd(a_1, b_1) &= a_1x_1 + b_1y_1 = d \\ gcd(a_2, b_2) &= a_2x_2 + b_2y_2 = d \end{aligned} \tag{9.14}$$

where $d = gcd(a, b)$. We also know that

$$\begin{aligned} a_2 &= b_1 \\ b_2 &= a_1 \bmod b_1 \\ b_2 &= a_1 - b_1k \end{aligned} \tag{9.15}$$

where $k = \left\lfloor \frac{a_1}{b_1} \right\rfloor$. Replacing 9.15 in 9.14 we get:

$$\begin{aligned} d &= a_1x_1 + b_1y_1 \\ d &= b_1x_2 + (a_1 - b_1k)y_2 \end{aligned} \tag{9.16}$$

Then

$$\begin{aligned} a_1x_1 + b_1y_1 &= b_1x2 + (a_1 - b_1k)y_2 \\ &= a_1y_2 + b_1(x_2 - ky_2) \end{aligned}$$

This is true only if

$$\begin{aligned} x_1 &= y_2 \\ y_1 &= x_2 - ky_2 \end{aligned} \tag{9.17}$$

Generalizing 9.17 we have

$$\begin{aligned} x_i &= y_{i+1} \\ y_i &= x_{i+1} - ky_{i+1} \end{aligned} \tag{9.18}$$

We only need the initial values to obtain the values of x_1 and y_1, but in the last iteration of the Euclidean algorithm the values of a_n and b_n are already known.

$$\begin{aligned} d &= a_n x_n + b_n y_n \\ &= dx_n + 0y_n \end{aligned}$$

Then, for the initial values we have

$$\begin{aligned} x_n &= 1 \\ y_n &= 0 \end{aligned}$$

The code in 9.6 reads two positive integers A and B ($A, B < 1000000001$) and prints X, Y, and $gcd(A, B)$, if there is more than one solution prints the solution where $|X| + |Y|$ is minimal (primarily) and $X \leq Y$ (secondly).

Time Complexity: The worst case scenario is the same of the *Euclidean algorithm*, it happens when A and B are consecutive *Fibonacci* numbers.

Input:

`A,B`. Two positive integers.

Output:

Three numbers `X Y D`, representing a solution for the equation $AX + BY = D$, where $D = gcd(A, B)$.

Listing 9.6: Extended Euclidean Algorithm

```
1 #include <cstdio>
2 using namespace std;
```

```
3
4  long x, y, d;
5  void extendedEuclidean(long, long);
6
7  int main() {
8    long a, b;
9
10   scanf("%ld %ld", &a, &b);
11   extendedEuclidean(a, b);
12   printf("%ld %ld %ld\n", x, y, d);
13
14   return 0;
15 }
16
17 void extendedEuclidean(long a, long b) {
18   long x1, y1;
19
20   if (b == 0) {
21     d = a;
22     x = 1;
23     y = 0;
24   } else {
25     extendedEuclidean(b, a % b);
26     x1 = x;
27     y1 = y;
28     x = y1;
29     y = x1 - (a / b) * y1;
30   }
31 }
```

9.4 Base Conversion

Base conversion is very straight-forward to implement, but given that is common to face problems that ask to convert a number from one base to another, we decided that it is a good idea to incorporate a section about that.

We are used to work with the decimal system, which are numbers in base 10 that can contain 10 different digits: $0, 1, \ldots, 9$. So when we see a number, for example, 483, it can be seen in the following way:

$$\begin{aligned} 483 &= (4 \times 100^2) + (8 \times 10^1) + (3 \times 10^0) \\ 483 &= 400 + 80 + 3 \end{aligned}$$

The same process is applied for different bases, for example, for base 2 we can use only two digits, 0, and 1, and if we want to represent any number, like 43, we do the following:

$$43 = (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$
$$43 = 32 + 8 + 2 + 1$$

So the binary representation for 43 is 101011. The program in 9.7 reads an integer b representing a base, a string B representing a number written in base b, and a decimal number num. The output consists on two lines, the first is the decimal representation of B, and the second is the number num represented in base b.

Time Complexity: $O(k)$

k. The number of digits

Input:

`numToDec`. A string *str* and an integer b.

`decToNum`. Two numbers n and b.

Output:

`numToDec`. A number with the 10-based representation of b-based number *str*.

`decToNum`. A string with the b-based representation of 10-based number n.

Listing 9.7: Base Conversion

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#define N 255
using namespace std;

long numToDec(char *, long);
char *decToNum(long, long);

int main() {
  char B[N];
  long num, b;

  scanf("%ld", &b);
  scanf("%s", B);
  scanf("%ld", &num);

  printf("%ld\n", numToDec(B, b));
  printf("%s\n", decToNum(num, b));

  return 0;
}
```

The function `numToDec` receives a string *str* and a long integer b, representing a number *str* written in base b, and returns a long

integer with the decimal representation of *str*.

```
long numToDec(char *str, long b) {
  long j, k, p;
  long len = strlen(str);
  long s = 0;

  j = 0;
  p = 1;
  for (int i = len - 1; i >= 0; i--) {
    if (str[i] >= 'A' && str[i] <= 'Z') {
      k = str[i] - 'A' + 10;
    } else {
      k = str[i] - '0';
    }
    s = s + k * p;
    p *= b;
  }

  return s;
}
```

The function `decToNum` receives a long integer n and a long integer b, where n is a decimal number, the function returns a string representing n written in base b.

```
char *decToNum(long n, long b) {
  long num, k = 0;
  char temp;
  char *s = new char[N];

  do {
    num = n % b;
    if (num >= 10) {
      s[k++] = (num - 10) + 'A';
    } else {
      s[k++] = num + '0';
    }

    n /= b;
  } while (n > 0);

  s[k] = '\0';
  for (int i = 0, j = k - 1; i < k / 2; i++, j--) {
    swap(s[i], s[j]);
  }

  s[k] = '\0';
  return s;
}
```

9.5 Sum of Number of Divisors

For any integer k consider the function $F(k)$ which returns the number of divisors of number k. Given an integer n, the goal is to obtain the value of $H(n)$, where H is a function defined by 9.19.

$$H(n) = \sum_{i=1}^{n} F(i) \tag{9.19}$$

Code in 9.8 reads an integer n and prints the value of $H(n)$. The idea is to count how many numbers have 1 as divisors, that's easy, there are n numbers. Now, how many numbers have 2 as divisor? Well, there are $n/2$ numbers. What about the numbers with 3 as divisor? There are $n/3$ of them, and so on. The result is the sum of all those values. The algorithm consists on a loop that continue to execute while

$$\frac{n}{k} > k - 1.$$

Solving for k we have that $k < \frac{1+\sqrt{1+4n}}{2}$. Then the time complexity of the algorithm is $O(\sqrt{n})$.

Time Complexity: $O(\sqrt{n})$
Input:
An integer n.
Output:
The value of $H(n)$.

Listing 9.8: Sum of Number of Divisors

```
#include <cstdio>
using namespace std;

long long H(long long);

int main() {
  long long n;

  scanf("%lld", &n);
  printf("%lld\n", H(n));

  return 0;
}

long long H(long long n) {
  long long num, k;
  long long prev = n;
  long long i = 1;
  long long j = 0;
  long long s = 0;
```

```
21
22    do {
23      num = n / i;
24      k = prev - num;
25
26      s += k * j;
27      if (num > j) {
28        s += num;
29      }
30
31      prev = num;
32      i++;
33      j++;
34    } while (num > j);
35
36    return s;
37  }
```

9.6 Combinations

When we are asked to obtain the number of combinations, we basically are asked the question *"In how many ways can I select k elements from a set S of n elements?*. Then given a set S of n elements, a combination is a subset of S of k elements, where $k \leq n$. For example, consider a set of three colors {red, green, blue}. How many combinations of two colors can be formed? The answer is three: {red, blue}, {red, green}, and {blue, green}. The number of ways to select k elements out of n elements is defined by equation 9.20.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{9.20}$$

For the example mentioned above, we have three colors ($n = 3$), and we want to select two colors ($k = 2$), then the number of ways to do that is

$$\binom{3}{2} = \frac{3!}{2!(3-2)!} = \frac{6}{2(1)} = 3.$$

In order to write a program that calculates the number of combinations by using equation 9.20, is necessary to be careful of an overflow error, since the value of $n!$ can exceeds an integer capacity, even when the final result is in the range of an integer. One thing that can help to handle larger values of n, is to divide the value of $n!$ as we are doing the multiplications. For example, for $n = 7$ and $k = 3$ we can do the following:

$$\begin{aligned}\binom{7}{3} &= \frac{7!}{3!4!} \\ &= \frac{\cancel{1} \times \cancel{2} \times \cancel{3} \times \cancel{4} \times 5 \times 6 \times 7}{(1 \times 2 \times 3)(\cancel{1} \times \cancel{2} \times \cancel{3} \times \cancel{4})} \\ &= \frac{7 \times 6 \times 5}{1 \times 2 \times 3} \\ &= (((7/1) \times 6)/2 \times 5)/3 \\ &= 35\end{aligned}$$

This will keep the operations smaller instead of calculating the value of $n!$, also this method would run faster, since we are dividing at the same time we are multiplying. The function `comb` implemented in 9.9 receives two integers n and m and returns the value of $\binom{n}{k}$.

Time Complexity: $O(n)$
Input:
Two numbers n and k, where $k \leq n$.
Output:
The value of $\binom{n}{k}$.

Listing 9.9: Combinations

```
int comb(int n, int k) {
  int m = max(k, n - k);
  int res = 1;
  for (int i = n, j = 1; i > m; i--, j++) {
    res *= i;
    res /= j;
  }

  return res;
}
```

9.6.1 Pascal's Triangle

The first rows of *Pascal's Triangle* can be seen in figure 9.1. The triangle can be built easily. Notice that the i^{th} row contains exactly $i + 1$ elements with a 1 in the first column, and for $j > 0$ we have that the element in the position (i, j) is the sum of elements in positions $(i - 1, j - 1)$ and $(i - 1, j)$. Suppose that empty cells are zeros.

	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
8	1	8	28	56	70	56	28	8	1

Figure 9.1: Pascal's Triangle

There are some hidden properties inside the *Pascal's Triangle*, for example, the third column contains all triangular numbers. Another property is that if we trance a diagonal from $(i, 0)$ to $(0, i)$, and sum the values crossed we get the i^{th} Fibonacci number. Also notice that the element in position (n, k), where $k \leq n$ contains the value of $\binom{n}{k}$, which is very convenient, because we can apply the properties of modular arithmetic, something that we cannot do using equation 9.20.

9.7 Catalan Numbers

In combinatorial mathematics, the Catalan numbers named after Eugène Charles Catalan, is a sequence of natural numbers that can be applied to various problems and have the following form:

$$\begin{aligned} C_0 &= 1 \\ C_{n+1} &= \frac{2(2n+1)}{n+2} C_n \end{aligned} \tag{9.21}$$

The first 10 Catalan numbers are: $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862$. Following we list some applications of Catalan numbers.

Balanced Parentheses

The n^{th} Catalan number gives us the number of ways we can arrange $2n$ parentheses (n open and n close). See image 9.2.

$n = 0$:	*	1 way
$n = 1$:	()	1 way
$n = 2$:	()(), (())	2 ways
$n = 3$:	()()(), (())(), ()(()), (()()), ((()))	5 ways
$n = 4$:	()()()(), ()(())(), ()()()(), ()(()()), ()((())), (())()(), (())(()), (()())(), ((()))(), (()()()), ((()))(), ((())()), (((())())), ((()()())), (((()))()), ((((()))))	14 ways
$n = 5$:	()()()()(), ()()(())(), ()()()()(), ()()(()()), ()()((())), ()(())()(), ()(())(()), ()(()())(), ()(()()()), ()((()))(), ()((())()), ()(((())())), ()(((()()))), (())()()(), (())()(()), (())(())(), (())(()()), (()())()(), (()())(()), (()()())(), (()()()()), (((())))(), (((()))()), (((()()))), (((())()())), (((()))()), (((()())())), (((()()()))), ((((()))())), ((((())()))), ((((()()))), ((((())))), ((((())()))), (((((()))))), (((((())))))	42 ways

Figure 9.2: In how many ways can arrange n open parentheses and n close parentheses?

Mountains

How many mountains can we form using n up-strokes and n down-strokes? This is another way to see the parentheses problem. See image 9.3.

$n = 0$:	*	1 way
$n = 1$:	/\	1 way
$n = 2$:	/\ /\/\, / \	2 ways
$n = 3$:	/\ /\ /\ /\ /\ /\/\/\, /\/ \, / \/\, / \, / \	5 ways

Figure 9.3: The number of mountains that can be formed using up-strokes and down-strokes.

Polygon Triangulation

The number of ways to triangulate a regular polygon with $n + 2$ sides is given by the n^{th} Catalan number. See image 9.4.

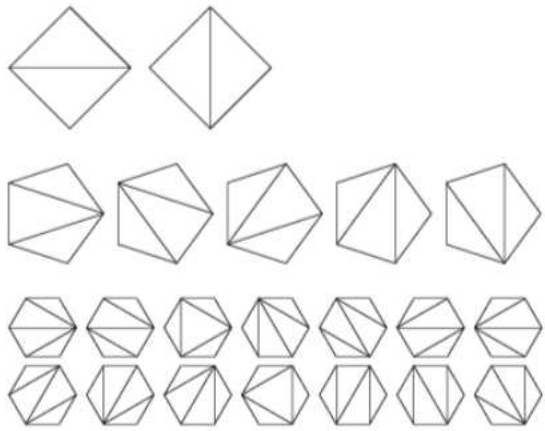


Figure 9.4: Triangulate a polygon of $n + 2$ sides.

Hands Across the Table

There is a circular table with $2n$ people seated around it. In how many ways can all of them be simultaneously shaking hands with another person is such way that none of the arms cross each other? See image 9.5.

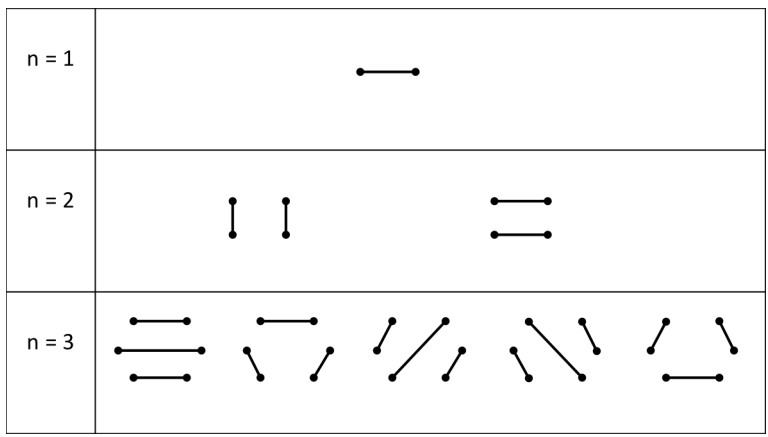


Figure 9.5: $2n$ persons shake hand without crossing their arms.

Binary Trees

The Catalan numbers count the number of rooted binary trees with n internal nodes. A rooted binary tree is a set of nodes and edges connecting them, where each node has either two children or none. Internal nodes are the ones that have two children. See image 9.6.

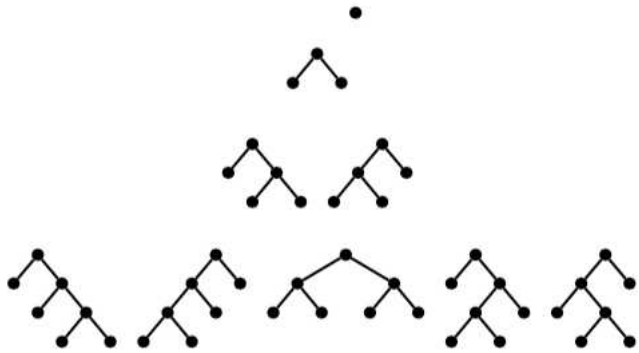


Figure 9.6: How many rooted binary trees can we form?

9.8 Josephus

There are n persons seated in a round table numbered from 1 to n. In the first round, person 1 receives a ball, then he passes the ball to the person at his left (person 2), and so on, after k steps, the person that has the ball loses and leaves the table. The second round begins with the person at the left of the person that has just lost. the process continues until a single person remains in the table, that person is considered the winner.

In image 9.7 there is a case with $n = 4$ and $k = 2$. In the first round, person 2 loses, the second round begins in person 3, so person 4 loses, finally in the third round there are only two persons at the table, person 1 and 3, beginning in person 1, person 3 loses, leaving person 1 as the only person in the table, thus the winner is person 1.

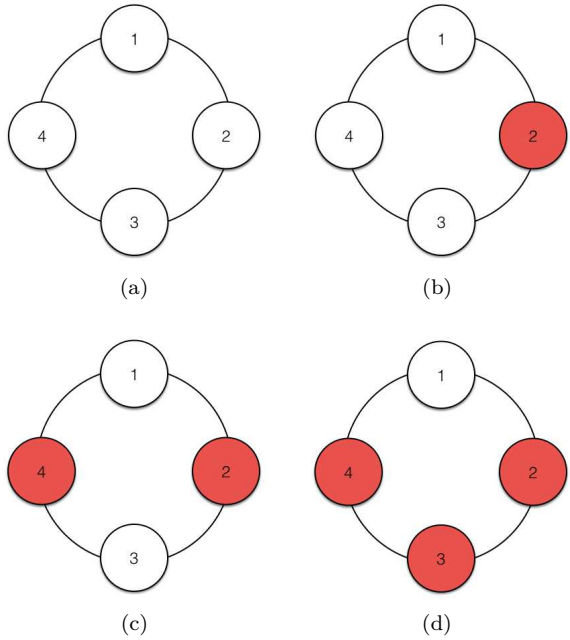


Figure 9.7: Josephus problem with $n = 4$ and $k = 2$. In every round the persons that lose is colored in red. 9.7b First round. Person 2 loses. 9.7c Second round. Person 4 loses. 9.7d Third round. Person 3 loses, and person 1 is the winner.

The code in 9.10 basically simulates the game in a recursive way. It reads two integers n and k and prints the winner of the Josephus problem.

Time Complexity: $O(n)$
Input:
Two positive integers n and k.
Output:
A number indicating the winner of the Josephus problem.

Listing 9.10: Josephus Problem

```
#include <cstdio>
using namespace std;

long josephus(long, long);

int main() {
```

```
  long n, k;
  scanf("%ld %ld", &n, &k);
  printf("%ld\n", josephus(n, k) + 1);

  return 0;
}

long josephus(long n, long k) {
  if (n == 0) {
    return 0;
  } else {
    return (josephus(n - 1, k) + k) % n;
  }
}
```

9.9 Pigeon-Hole Principle

The pigeon-hole principle states that if n items are put into m containers, with $n > m$, then at least that one item must contain more than one item. If there are n pigeons and m holes, and $n > m$, then at least one hole will contain more than one pigeon. See figure 9.8

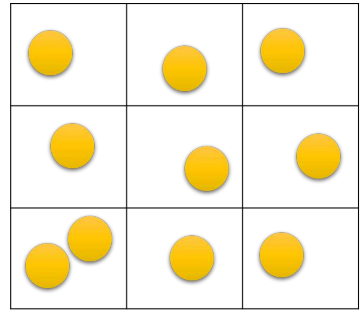


Figure 9.8: pigeon-hole principle using 10 yellow balls and 9 squares, since there are more balls than squares, at least one square will contain more than one ball.

Consider the case of a street where each year on Halloween each neighbour is only willing to give a certain total number of sweets on that day, no matter how many children call on him. To avoid conflicts, the children have decided they will put all sweets together and then divide them evenly among themselves. The children know how many sweets they get from each neighbour. Since they care more about justice than about the number of

sweets they get, they want to select a subset of the neighbours to visit, so that in sharing every child receives the same number of sweets.

The program 9.11 receives as input two integers n and m ($1 \leq n \leq m \leq 100000$), representing the number of children and the number of neighbours, respectively. The next line contains m space separated integers $x_1, ..., x_m$ ($1 \leq x_i \leq 100000$), where x_i represents the number of sweets the children get if they visit neighbour i. The output is the set of houses the children need to visit so they can divide equally the number of sweets without wasting any of them.

The trick here is to notice that that $m \geq n$, and apply modular arithmetic along with the pigeon-hole principle. store the sum of the candies modulo n, that will return a number in $[0, n-1]$, let's call that value r. If r is zero, we finished and print the numbers from 1 to k, where k is the index of the last element added to the sum. If r is not zero, mark cell $s_r = k$ if it is not already marked. in the case it is already marked print the numbers from $s_r + 1$ to k.

Time Complexity: $O(m)$
Input:
`n`. Number of children.
`m`. Number of neighbors.
`x`. Array where x_i represents the sweets the children can get from neighbor i.
Output:
The sequence of houses that the children must visit in order to split the sweets equally.

Listing 9.11: Pigeon-Hole Principle

```
#include <iostream>
#include <cstdio>
#include <cstring>
#define N 100001
using namespace std;

long x[N], s[N];

int main() {
  long k, n, m;

  scanf("%ld %ld", &n, &m);
  memset(s, -1, sizeof(s));
```

```
  k = 0;

  for (int i = 0; i < m; i++) {
    scanf("%ld", &x[i]);

    k = (k + x[i]) % n;
    if (s[k] == -1) {
      if (k == 0) {
        printf("1");
        for (int j = 1; j <= i; j++) {
          printf(" %ld", j + 1);
        }
        printf("\n");
        goto finish;
      }
      s[k] = i;
    } else {
      printf("%ld", s[k] + 2);
      for (int j = s[k] + 2; j <= i; j++) {
        printf(" %ld", j + 1);
      }
      printf("\n");
      goto finish;
    }
  }

finish:
  return 0;
}
```

9.10 Chapter Notes

Number Theory is one of those things that seems magical. Numbers have hidden properties, peculiarities, and coincidences that are hard to believe, but they actually exist, making it one of the most interesting areas in mathematics. On the other hand. *Combinatorics* is an area with a lot of applications in real-world problems, specially in the field of computer science, since it is a basic tool to analyze algorithm's performance.

Jones and Jones [28], describe and analyze properties about prime numbers, divisibility, modular arithmetic, et. al. Knuth [29] [30], do a detailed analysis of methods like the Euclid's algorithm and modular arithmetic, and dedicate a whole book to combinatorial algorithms. Lehmann [31] also mention some mathematical concepts in the area, like the Newton's binomial theorem, which include the calculus of combinations.

The following links contain a series of mathematical problems, among them some related to number theory and combinatorics,

other subjects included are game theory, probability, and numerical methods, just to mention some, that although those topics are not included in this book, it is worth to take a look to them.

- `https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=450`
- `http://acm.timus.ru/problemset.aspx?space=1&tag=numbers`
- `https://www.urionlinejudge.com.br/judge/en/problems/index/5`
- `https://projecteuler.net/archives`

9.11 Exercises

1. Write a program that given an integer n, $(1 \leq n \leq 100)$, express $n!$ as the product of its prime factors.
2. For a given number n, $(2 < n < 10)$. Is there a triplet of numbers a, b, c, such as $a^n + b^n = c^n$?
3. Describe the algorithm you would use to find the 100000000^{th} Fibonacci number modulus $10^9 + 7$.
4. Given two numbers a and b, write a program that find the least common multiple (lcm) of both.
5. Propose an algorithm that find a solution for

$$ax + by = d$$

   for some given values of a, b, and d.
6. Write a program that given an even integer n, $(2 < n < 1000)$, express that number as the sum of two primes, if it is not possible, print "No solution".

10

String Manipulation

"Common sense is not so common."

– Voltaire

A string is an array of characters, and *String Manipulation* refers to problems that involves string handling. e.g. Find a word inside of another word, reverse its characters, find out if a word is a palindrome, et.al.

When a string is printed in the screen, what the user see is a letter or a symbol, but what the computer interprets is its `ASCII` code . For example, the ASCII code for letter 'A' is 65, for letter 'B' is 66, and so on. Consider the problem to count how many times a letter appears in a word, for simplicity assume that all letters are upper-case, what would be a good implementation?

A bad implementation would be to check every possible case, that will consists on 26 cases we need to consider. A better approach is to use an array X of 26 elements as a counter, in such a way that X_0 represents how many $A's$ are contained in the word, X_1, how many $B's$, and so on. In this case $word_i$ represents the i^{th} character of the word we are working with, then for every letter in the word we do the following

$$X_k = X_k + 1,$$

where $k = word_i -' A'$. In that way if $word_i =' A'$, k would be 0, and the value of X_0 will increment in one, in the same way if

$word_i =' B'$, then k is equal to 1 and X_1 will increment in one.

When we subtract letter 'A' we are actually subtracting 65 which is the `ASCII` value of 'A'. Doing $k = word_i - 65$ would have the same effect. In conclusion we can say that strings have properties that other types of variables don't have, and we can use those properties to our advantage when solving a problem.

The library `string` in C++ contains different methods and properties that allows to perform certain taks and even do operations with strings, for example if we have the string "hello" and other string "world", is valid to do "hello" + " " + "word". The result is a single string "hello world". In this case the operator + functions as a concatenator.

During this chapter we will solve and analyze different problems involving strings, for that we would use the properties of the strings, and the `string` library. Of course those are just tools that makes things easier, every problem involves a deep thinking in order to get to the best solution.

10.1 Next Permutation

Given a string of n letters, the goal is to find the next permutation of that string. Consider string X.

$$X = ABAACB$$

The first thing to do is to identify the last element X_j that is smaller than the element at its immediate right. In this case X_3. If no element is found then there is no next permutation.

The next step is to verify if element X_n is smaller or equal than X_j, if it is, we verify with X_{n-1} and so on, until we find an element X_k larger than X_j. For the sample case X_5 is larger than X_3, then $k = 5$.

Swap elements X_j and X_k.

$$X = ABABCA$$

Now reverse the elements from position $j+1$ to n to obtain the next permutation.

$$X = ABABAC$$

The code in 10.1 receives a string *str* and prints its next permutation if there is one, otherwise prints the message "No permutation".

Time Complexity: $O(n)$

n. Number of characters.

Input:

The string which we want to obtain the next permutation from.

Output:

The next permutation if it exists, otherwise prints *No permutation.*

Listing 10.1: Next Permutation

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#define N 255
using namespace std;

char *nextPermutation(char *, int);

int main() {
  char str[N];
  int n;

  scanf("%s", str);
  n = strlen(str);
  if (nextPermutation(str, n) != NULL) {
    printf("%s\n", str);
  } else {
    printf("No permutation\n");
  }

  return 0;
}

char *nextPermutation(char *cad, int n) {
  int j = n - 2, k, r, s;
  char temp;

  while (j >= 0 && cad[j] >= cad[j + 1]) {
    j--;
  }

  if (j < 0) {
    return NULL;
  }

  k = n - 1;
  while (cad[j] >= cad[k]) {
    k--;
  }
```

```
40
41    swap(cad[j], cad[k]);
42
43    r = n - 1;
44    s = j + 1;
45    while (r > s) {
46      swap(cad[r--], cad[s++]);
47    }
48
49    return cad;
50  }
```

10.2 Knuth-Morris-Pratt Algorithm (KMP)

Created in 1977 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris, but the three published it jointly [32]. Is an algorithm used to determine if certain string W is contained inside another string S of equal or bigger size. See the example below.

$$W = ABC$$
$$S = ABAABB\mathbf{ABC}ABD$$

Notice that W is inside S starting at index 6. The KMP algorithm is faster than a brute force algorithm, because it stores information of previous characters , for example, consider the case where

$$W = abcabd$$
$$S = abcabc\ldots$$

Notice that even when S starts very similar to W they differ in S_5. What the KMP algorithm does, is to restart the search from W_3, because we already have the letters abc, so we don't need to start from the beginning of W every time a character doesn't match, that is the main advantage of the KMP algorithm.

The code 10.2 uses the KMP algorithm to find the location of a string W inside string S. The function `kmpTable` builds an array T, where T_i represents the index where the search must be

restarted in W when the character W_{i+1} doesn't match.

The time complexity of the function `kmpTable` is $O(n)$, where n is the length of string W. On the other hand the function `kmp` runs in $O(m)$ time, where m is the length of string S. Then the time complexity of the algorithm is the sum of both, which is $O(n+m)$.

Time Complexity: $O(n+m)$

`n`. Length of W.

`m`. Length of S.

Input:

Two strings W and S, both containing no more than 1000 characters.

Output:

If W is found inside of S prints the position in S where the first character of W is located, otherwise prints a message indicating that W wasn't found.

Listing 10.2: KMP Algorithm

```
#include <iostream>
#include <cstdio>
#include <cstring>
#define N 1001
using namespace std;

char W[N], S[N];
int T[N];
int lenW, lenS;

int kmp();
void kmpTable();

int main() {
  int pos;

  scanf("%s %s", W, S);
  pos = kmp();
  if (pos == -1) {
    printf("string %s not found\n", W);
  } else {
    printf("string %s found at %d\n", W, pos);
  }

  return 0;
}

int kmp() {
  int i, m;

  lenW = strlen(W);
  lenS = strlen(S);
  kmpTable();
```

```
34
35    m = i = 0;
36    while (m + i < lenS) {
37      if (W[i] == S[m + i]) {
38        if (i == lenW - 1) {
39          return m;
40        }
41        i++;
42      } else {
43        m = m + i - T[i];
44        i = (T[i] > -1) ? T[i] : 0;
45      }
46    }
47
48    return -1;
49  }
50
51  void kmpTable() {
52    int pos = 2;
53    int k = 0;
54    T[0] = -1;
55    T[1] = 0;
56
57    while (pos < lenW) {
58      if (W[pos - 1] == W[k]) {
59        k++;
60        T[pos] = k;
61        pos++;
62      } else if (k > 0) {
63        k = T[k];
64      } else {
65        T[pos] = 0;
66        pos++;
67      }
68    }
69  }
```

10.3 Manacher's Algorithm

Finding the longest palindrome inside a given string S is a common problem in computer science. Perhaps the trivial solution is to fix the initial letter and the length of the word and check if it is a palindrome. That solution runs in $O(n^3)$. A better approach is to fix the center of a word and expand to both sides increasing a counter every time two letters match. That approach has a time complexity of $O(n^2)$.

Manacher proposed in 1975 an algorithm to solve this problem in $O(n)$ time. The first step of this algorithm consist on transforming the string S into a new string T by adding a new char ('∧') at the beginning of S, and another char ('$') at the end of S. These two characters are needed to identify the boundaries of the

new string. Then we separate each pair of characters with some random char ('#'). For example if we have the string:

$$S = ABABABABA$$

We would get

$$T = \wedge\#A\#B\#A\#B\#A\#B\#A\#B\#A\#\$$$

The function `preProcess` in code 10.3 converts a given string S into a new string T following the method described above.

Listing 10.3: Manacher's Algorithm

```
// Transform S into T.
// For example, S = "abba", T = "^#a#b#b#a#$".
// ^ and $ signs are sentinels appended to each end to avoid bounds checking
string preProcess(string S) {
  int n = S.size();
  string T;

  if (n == 0) {
    return "^$";
  }

  T.resize(2 * n + 3);
  T[0] = '^';
  for (int i = 0; i < n; i++) {
    T[2 * i + 1] = '#';
    T[2 * i + 2] = S[i];
  }

  T[2 * n + 1] = '#';
  T[2 * n + 2] = '$';

  return T;
}
```

To find the the longest palindrome inside T we can use some properties of palindromes. For that consider a palindrome with its middle element in position C, and last element in position R.

1. All elements at the left of C are mirrored in the right side of C.

2. if a new palindrome is centered in position i, where $C < i \leq R$, the elements in positions i and $2C - i$ are equal.

3. Be P_k the length of the longest palindrome centered in position k. For a new palindrome centered in position i, where $C < i \leq R$ we can be sure that the value of P_i will be at least P_{2C-i}. So the value of P_i is initialized as follows:

$$P_i = \min(R - i, P_{2C-i})$$

We need to compare with the value of $R - i$ in order to stay inside the boundaries of the palindrome. Once the value of P_i is initialized we can expand to both sides of the string and increment the value of P_i if two letters match.

If the method `manacher` uses the string T created in method `preProcess` and implements the Manacher's algorithm using the properties described above. The method returns the length of the longest palindrome.

```
string manacher(string S, string T) {
  int C = 0, R = 0;
  int maxLen = 0;
  int centerIndex = 0;
  int n = T.size();
  vector<int> P(n);

  for (int i = 1; i < n - 1; i++) {
    int j = 2 * C - i;
    P[i] = (R > i) ? min(R - i, P[j]) : 0;

    // Expand palindrome centered at i
    while (T[i + 1 + P[i]] == T[i - 1 - P[i]]) {
      P[i]++;
    }

    // If a longer paldindrome is found, update center
    if (i + P[i] > R) {
      C = i;
      R = i + P[i];
    }

    // Store the center and length of longest palindrome
    if (P[i] > maxLen) {
      maxLen = P[i];
      centerIndex = i;
    }
  }

  return S.substr((centerIndex - 1 - maxLen) / 2, maxLen);
}
```

The following program reads a string S and prints the longest palindrome inside S.

Time Complexity: $O(n)$

`n`. Length of string S.

Input:

A string S containing only letters.

Output:
The length of the longest palindrome inside S.

```
#include <string>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

string preProcess(string);
string manacher(string, string);

int main()
{
        string S, T;

        cin >> S;
        T = preProcess(S);
        cout << manacher(S,T) << "\n";

        return 0;
}
```

10.4 Chapter Notes

String processing is something that is found not only in programming contests, but also in the industry. For example, machine learning algorithms need clean data to make accurate predictions or classifications, and that data sometimes consists on high volumes of text that contains typos, or missing values, so we need to do a pre-processing step before training any model. Another example is marking as spam emails containing the word "offer", for that purpose algorithms like *KMP* are very handy. Other applications are file compression, natural language processing, cryptography, et. al.

Working with strings commonly comes hand in hand with data structures, specially with hash maps, e. g. To count the number of occurrences of a certain word in a text, a map can be used with the word as the key and the occurrences as the value. Sedgewick [5] describe different algorithms about string manipulation and its applications, along with source code for some of them.

The links bellow contain problems related to the area of string processing or string manipulation, we recommend the reader to try to solve some of the problems as it sees fit.

- `https://www.urionlinejudge.com.br/judge/en/`

problems/index/3

- http://acm.timus.ru/problemset.aspx?space=1&tag=string
- https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=504

10.5 Exercises

1. Write a function that receives a string S and returns *true* if S is a palindrome, otherwise returns *false*.

2. Given a string S of length n containing only lower-case letters. Propose an algorithm that finds the longest palindrome inside S in $O(n^2)$ time.

3. Describe two different strategies to find out if two strings are anagrams.

4. Write a program that given a string S containing only letters, count the amount of upper-case letters and the amount of lower-case letters.

5. Caesar cipher is an old method of encryption, it is said that Julius Caesar used it to encrypt his messages. It consists on replacing each letter for a letter k positions ahead in the alphabet. e.g. If $k = 2$, letter A will be encrypted as C, B will be encrypted as D, and so on. Below there is a message and its encryption using $k = 2$.

   $$\begin{aligned} original &= ABCDEFGHIJKLMNOPQRSTUVWXWZ \\ encrypted &= CDEFGHIJKLMNOPQRSTUVWXYZAB \end{aligned}$$

   Write a function that receives a string S containing only upper case letters and a value of k, and returns the string encrypted.

11

Solution to Exercises

11.1 Fundamentals

1. We can use the solution of the 8-queen problem in A.1 as reference, with the difference that instead of trying to place queens, we are trying to place digits following a different set of rules. The implementation consists on trying to place a valid digit in an empty cell and then move to the next cell, if a cell is not empty we ignore it an continue with the next one. This should happen inside of a recursive function, so in case we end up in an invalid solution, the same program can go back and try other digits until a valid solution is found. Basically we are trying all possible combinations of filling the matrix complying by the rules. This approach of trying all possible combinations following a set of rules in a recursive function is called **backtracking**.

   To simplify our solution let's implement three functions that will help us to validate if is possible to place a digit in certain cell. The function `isInRow` returns true if a digit is present in a given row, false otherwise. `isInColumn` does the same but in a given column. Finally `isInSubMatrix` receives a digit, a row and a column and returns true if that figit appears in the corresponding 3×3 sub-matrix.

```
bool isInRow(int val, int row) {
  for (int i = 0; i < 9; i++) {
    if (sudoku[row][i] == val) {
      return true;
    }
  }
  return false;
}

bool isInColumn(int val, int col) {
  for (int i = 0; i < 9; i++) {
    if (sudoku[i][col] == val) {
      return true;
    }
  }
  return false;
}

bool isInSubmatrix(int val, int row, int col) {
  row = row / 3;
  col = col / 3;

  for (int i = 3 * row ; i < 3 * (row + 1); i++) {
    for (int j = 3 * col; j < 3 * (col + 1); j++) {
      if (sudoku[i][j] == val) {
        return true;
      }
```

```
    }
  }
  return false;
}
```

We are going to use the global variable `sudoku` to store the 9×9 matrix. The functions explained above will tell us if we can place a digit in a certain position during our recursive search.

The function 11.1 receives an id of the current cell, this id is just an integer representation of the cell (i, j), with $id = i \times 9 + j$. Then the function first check if we have filled all the cells in the Sudoku, if that is the case we print the current solution and stop the search. If we have not completed the matrix, we must check if we are in an empty cell, if the cell is not empty we ignore it and move to the next one, but it the cell is empty then we must try to place valid digits in that cell, once we place a valid digit we move to the next cell. Recursively we will try all possible valid digits in a cell.

Listing 11.1: Fundamentals. Exercise 1

```
1  void solveSudoku(int pos) {
2    int row = pos / 9;
3    int col = pos % 9;
4    if (row == 9) {
5      printSudoku();
6      return;
7    }
8
9    if (sudoku[row][col] != 0) {
10     solveSudoku(pos + 1);
11   } else {
12     // Try all numbers from 1 to 9
13     for (int i = 1; i <= 9; i++) {
14       if (!isInRow(i, row) && !isInColumn(i, col) &&
15           !isInSubmatrix(i, row, col)) {
16         sudoku[row][col] = i; // Place i in cell (row, col)
17         solveSudoku(pos + 1); // Move to the next cell
18         sudoku[row][col] = 0; // Set as empty to reuse
19       }
20     }
21   }
22 }
```

The `printSudoku` function, just prints the variable `sudoku` in a readable way.

```
void printSudoku() {
  cout << "===== START =====\n";
```

```
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
      cout << sudoku[i][j] << " ";
    }
    cout << "\n";
  }
  cout << "===== END =====\n";
}
```

The `main` function below reads the 9×9 matrix, with 0's representing empty cells, and call `solveSudoku(0)` to trigger the recursive search starting from the first cell.

```
int main() {
  int num;
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
      cin >> num;
      sudoku[i].push_back(num);
    }
  }

  solveSudoku(0);
  return 0;
}
```

Below is a sample input with its corresponding output. See how every digit appear once in each row, column and 3×3 sub-matrix, meaning that the rules we specified are working correctly. In case that there are more solutions the program will print all of them, for that specific input the solution is unique.

Sample Input	Sample Output
5 3 0 0 7 0 0 0 0 6 0 0 1 9 5 0 0 0 0 9 8 0 0 0 0 6 0 8 0 0 0 6 0 0 0 3 4 0 0 8 0 3 0 0 1 7 0 0 0 2 0 0 0 6 0 6 0 0 0 0 2 8 0 0 0 0 4 1 9 0 0 5 0 0 0 0 8 0 0 7 9	===== START ===== 5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9 ===== END =====

2. This problem is similar to the one in appendix A.2, the idea is to use a number as bitmask, with each bit representing a friend, in such a way that if the i^{th} bit is 1 that means that you have given a gift to friend i. This way we are using one

`int` in memory to store the information of who has received a gift and who hasn't.

We will use an integer variable `bitmask` initially equal to zero as bitmask, then if friend i receives a gift we just do the following:

```
bitmask |= (1 << i);
```

We use the `|` operator, because that way we won't modify the rest of the bits of `bitmask`, Since $(1 << i)$ is a number with 1 in the i^{th} bit and 0's in the rest of the bits. If the i^{th} bit of `bitmask` was activated, it will remain activated, and if it was not activated it will be activated.

To know if friend i has received a gift we just do

```
bitmask & (1 << i);
```

if the result is positive that means that the i^{th} bit of `bitmask` is 1, otherwise is 0. The solution for this problem is shown below.

Listing 11.2: Fundamentals. Exercise 2

```
#include <iostream>
using namespace std;

int main() {
  int n, m, q, friendId;
  int bitmask = 0;

  cin >> n >> m >> q;
  for (int i = 0; i < m; i++) {
    cin >> friendId;
    bitmask |= (1 << friendId);
  }

  for (int i = 0; i < q; i++) {
    cin >> friendId;
    if ((bitmask & (1 << friendId)) > 0) {
      cout << "YES\n";
    } else {
      cout << "NO\n";
    }
  }
  return 0;
}
```

11.2 Data Structures

1. For this problem we need to store at most k elements in a list. As we read numbers we insert the new element at front of the list and remove the last element from the back, that way we always keep the last k numbers in the list. To calculate the average we just need to store in a variable s the cumulative sum of all elements in the list, when a element is inserted we add its value to s, and when an element is removed we subtract its value from s. The ouput is s divided by the size of the list.

   The insertion and deletion operations in a list takes $O(1)$, and getting the average is also $O(1)$. Then the running time of the program is $O(n)$, since we need to read all n numbers.

Listing 11.3: Data Structures. Exercise 1

```
#include <iostream>
#include <list>
using namespace std;

int main() {
  int n, k, num, s;
  list<int> l;

  s = 0;
  cin >> n >> k;
  for (int i = 0; i < n; i++) {
    cin >> num;
    if (l.size() == k) {
      s -= l.back(); // Remove the last element from the average
      l.pop_back(); // Remove the last element from the list
    }

    s += num;          // Add the new number to the average
    l.push_front(num); // Insert the new number to the list

    cout << "mean: " << (double)(s) / l.size() << "\n";
  }

  return 0;
}
```

2. The first thing we need is to create the `Book` class, which will store the title of the book, the rank and a list of similar books.

Listing 11.4: Data Structures. Exercise 2

```
class Book {
 public:
```

```
3     string title;
4     double rank;
5     list<string> similarBooks;
6
7     Book(string title = "", double rank = 1.0) {
8       this->title = title;
9       this->rank = rank;
10    }
11
12    // Necessary for the unordered_set and unordered_map
13    bool operator==(const Book &otherBook) const {
14      return this->title == otherBook.title;
15    }
16
17    // Necessary for the priority_queue
18    bool operator<(const Book &otherBook) const {
19      // We are inverting the sign to build a min-heap
20      return this->rank > otherBook.rank;
21    }
22
23    void addSimilarBook(string title) {
          this->similarBooks.push_back(title); }
24  };
25
26  // Needed for unordered_set and unordered_map
27  struct BookHasher {
28    hash<string> hasher;
29    size_t operator()(const Book &key) const { return hasher(key.title);
          }
30  };
```

According to the overridden operators in the class, two books are considered equal if they have the same title, and when comparing two books, the one with higher rank comes first. The `BookHasher` structure uses a book title as input for the hashing function.

The `traverseGraph` function visits all connected nodes in a graph, a graph is similar to a tree, but each node can have multiple parents and children, thus is possible to have cycles. Traversing a graph is similar to traversing a tree, but since there can be cycles, we need to store which nodes have already been visited, in order to avoid visit them again, for that we use the `unordered_set visitedBooks`. This will allow us to know in $O(1)$ if a node has been visited.

At the same time we traverse the graph we can update the top k ranked books. For that we need a min-heap to store the top k ranked books, the one at the root will be the worst ranked of them (the worst ranked from the top ranked). For each book that we visit we need to compare it with the root,

if the rank's book is smaller it will also be smaller than the rest of the elements in the heap and we won't add it to the min-heap. On the other hand, if the rank's book is larger, we remove the root, since is the worst ranked element, and insert the current book, that way we always keep the k top ranked books inside the min-heap.

The cost of updating the min-heap is $O(\log k)$, and since we do that for every node visited, the time complexity of this solution is $O(n \log k)$.

```
void traverseGraph(Book book) {
  // Mark book as visited to avoid loops
  visitedBooks.insert(book);

  // Check if the current book should be added into the top-suggestions
  if (bookSuggestion.size() < k) {
    bookSuggestion.push(book); // add the book
  } else if (book.rank > bookSuggestion.top().rank) {
    bookSuggestion.pop();    // remove the book with worst ranking
    bookSuggestion.push(book); // add the current book
  }

  // Move to similar books that has not been visited
  for (auto it = book.similarBooks.begin(); it !=
       book.similarBooks.end();
      it++) {
    Book otherBook = inventory[*it];
    if (visitedBooks.find(otherBook) == visitedBooks.end()) {
      traverseGraph(otherBook);
    }
  }
}
```

The `main` function reads the numbers n and k as specified in the problem statement, then it reads the information of all the books in the inventory, and finally reads the title of certain book A.

The inventory is implemented as an `unordered_map` with the title of the book as key and its corresponding `Book` object as value. This way we can get the information of a book given the title in $O(1)$.

The program gets book A from the inventory given its title, and starts a graph traversal. Finally it prints the title and rank of the top k ranked books that are similar to A in increasing order of their rank.

```
int n, k;
unordered_map<string, Book, BookHasher> inventory;
unordered_set<Book, BookHasher> visitedBooks;
priority_queue<Book> bookSuggestion;

void traverseGraph(Book);

int main() {
  int m;
  string title, simTitle;
  double rank;

  cin >> n >> k;
  for (int i = 0; i < n; i++) {
    cin >> title >> rank;
    Book book = Book(title, rank);

    cin >> m;
    for (int j = 0; j < m; j++) {
      cin >> simTitle;
      book.addSimilarBook(simTitle);
    }
    // Add the book to the inventory with the title as the key
    inventory[title] = book;
  }

  // Read the title of currentBook and traverse the graph
  // searching for the top k ranked similar books.
  cin >> title;
  traverseGraph(inventory[title]);

  while (!bookSuggestion.empty()) {
    Book book = bookSuggestion.top();
    bookSuggestion.pop();
    cout << book.title << " " << book.rank << "\n";
  }

  return 0;
}
```

3. For this problem we will use a Trie to store the words in the dictionary. The time complexity to find a word in a trie is $O(n)$, where n is the length of the word, but also we can benefit from the fact that a Trie store prefixes, so if we are looking for a word in the puzzle and the prefix is not in the Trie, we can stop the search.

   From section 3.2.7, we will use without change the class `TrieNode` 3.34 and the function `addWord` 3.35.

   The function `findWords` receives a position in the puzzle and a direction, as we move across the puzzle in the given direction, at the same time we traverse the Trie, counting each

word that we found, both, the letter in the puzzle and the letter in the Trie should match, at the moment that the letters don't match we stop the search.

```
// This function moves trough the puzzle in a given
// direction and check if words are in the Trie.
// It returns the number of words found
int findWords(int row, int col, int dirY, int dirX) {
  int nWords = 0;
  int k = 0;

  while (row >= 0 && row < nRows && col >= 0 && col < nColumns) {
    int p = puzzle[row][col] - 'A';
    if (trie[k].ref[p] != -1) {
      k = trie[k].ref[p];
    } else {
      break;
    }

    if (trie[k].isWord) {
      nWords++;
    }
    row += dirY;
    col += dirX;
  }

  return nWords;
}
```

In the `main` function below, the first thing we do is to insert the root of the Trie to the vector `trie`, that way the root will be always the first element of the vector. After that we read the number of words in the dictionary and the dimensions of the puzzle. Next we read the words of the dictionary and add them into the trie using the function `addWord`. After we read each row of the puzzle, we traverse the puzzle starting a search at every cell in all directions (vertically, horizontally and diagonally), and if a word is found, we print the current location in the puzzle, that represents the first letter of the word(s) found, and how many words were found.

Listing 11.5: Data Structures. Exercise 3

```
int main() {
  string word, puzzleRow;
  int wordsFound;

  trie.push_back(TrieNode());
  // Read dictionary and add words to trie
  cin >> n >> nRows >> nColumns;
  for (int i = 0; i < n; i++) {
    cin >> word;
    addWord(0, word, 0);
  }
```

```
12
13    // Read word puzzle
14    for (int i = 0; i < nRows; i++) {
15      cin >> puzzleRow;
16      puzzle.push_back(puzzleRow);
17    }
18
19    // For every letter start a search in all 8 directions
20    for (int i = 0; i < nRows; i++) {
21      for (int j = 0; j < nColumns; j++) {
22        wordsFound = findWords(i, j, -1, 0);
23        wordsFound += findWords(i, j, 0, 1);
24        wordsFound += findWords(i, j, 1, 0);
25        wordsFound += findWords(i, j, 0, -1);
26        wordsFound += findWords(i, j, -1, 1);
27        wordsFound += findWords(i, j, 1, 1);
28        wordsFound += findWords(i, j, 1, -1);
29        wordsFound += findWords(i, j, -1, -1);
30
31        // Print the location of the starting letter and how many
32        // words were found
33        if (wordsFound > 0) {
34          cout << "(" << i << "," << j << "): " << wordsFound << "\n";
35        }
36      }
37    }
38
39    return 0;
40  }
```

For the input

```
16 8 12
BELT
BEAR
SHOE
HAND
BALL
MICE
TOYS
BAT
DOG
TOP
HAT
COW
ZAP
GAL
BOY
CAT
JLIBPNZQOAJD
KBFAMZSBEARO
OAKTMICECTQG
YLLSHOEDAOGU
SLHCOWZBTYAH
MHANDSAOISLA
TOPIFYPYAGJT
EZTBELTEATAH
```

the output is

```
(0,3): 1
(0,11): 1
(1,1): 1
(1,7): 1
(2,4): 1
(2,8): 1
(2,9): 1
(3,3): 1
(3,10): 1
(4,3): 1
(4,6): 1
(4,7): 2
(4,8): 1
(4,11): 1
(5,1): 1
(6,0): 1
(7,3): 1
(7,11): 1
```

11.3 Sorting Algorithms

1. Sort the regions in ascending order according to their number of citizens. The new leader only need to gain control of the majority of the regions by winning the regions with less citizens to assure the victory.

   For this case there is no need to overload an operator or using a class. We just need an array to store the citizens of each region, and call the `sort` function as showed in line 19 to sort the array.

Listing 11.6: Sorting Algorithms. Exercise 1

```
1  #include <cstdio>
2  #include <algorithm>
3  #define N 100001
4  using namespace std;
5
6  int R[N];
7  int n, m;
8
9  int main() {
10   int nVotes;
11
12   scanf("%d", &n);
13   for (int i = 0; i < n; i++) {
14     scanf("%d", &R[i]);
15   }
16
17   sort(R, R + n);
18
19   nVotes = 0;
20   m = n / 2;
```

```
  for (int i = 0; i <= m; i++) {
    nVotes += (R[i] / 2 + 1);
  }

  printf("%d\n", nVotes);
  return 0;
}
```

2. We can use *Counting Sort* for this problem, for every vote the i^{th} cardinal get, increment the value of X_i, where X is an array initialized with 0's. At the end we just need to find the value of k, such that $X_k > \frac{2}{3}n$.

Listing 11.7: Sorting Algorithms. Exercise 2

```
#include <cstdio>
#define N 1001
using namespace std;

int X[N];
int n, m;

int main() {
  int i, k;

  scanf("%d", &n);
  m = (2 * n) % 3 == 0 ? (2 * n) / 3 : (2 * n) / 3 + 1;

  for (i = 0; i < n; i++) {
    scanf("%d", &k);
    X[k]++;
  }

  for (i = 1; i <= n; i++) {
    if (X[i] >= m) {
      printf("%d\n", i);
      return 0;
    }
  }

  printf("No pope elected\n");
  return 0;
}
```

3. Sorting the points according to their distance to the origin and print the first k is a $O(n \log n)$ solution. A better approach that runs in $O(n \log k)$, since $k \leq n$, consists on storing the k closest points inside a heap, being the root the most distant of them, then, the only thing we must do is compare each new point to the root, if is is closer to the origin we remove the root from the heap and insert the new point, otherwise we do nothing.

Listing 11.8: Sorting Algorithms. Exercise 3

```
#include <algorithm>
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

class Point {
 public:
  int x;
  int y;
  int d2;

  Point(int x = 0, int y = 0) {
    this->x = x;
    this->y = y;
    this->d2 = (x * x) + (y * y);
  }

  bool operator<(const Point &b) const {
    return this->d2 < b.d2;
  }
};

priority_queue<Point> heap;
int n, k;

int main() {
  int i, x, y;

  scanf("%d %d", &n, &k);
  for (i = 0; i < n; i++) {
    scanf("%d %d", &x, &y);
    Point p = Point(x, y);
    if (heap.size() < k) {
      heap.push(p);
    } else if (heap.top().d2 > p.d2) {
      heap.pop();
      heap.push(p);
    }
  }

  while (!heap.empty()) {
    printf("%d %d\n", heap.top().x, heap.top().y);
    heap.pop();
  }

  return 0;
}
```

11.4 Divide and Conquer

1. There are several approaches to solve this problem, one that is not a very good one is to walk trough the interval $[a, b]$ adding all values, that would run in $O(nq)$ which is very poor. Another solution is based in *Dynamic Programming*,

and consists on using an array S to store the cumulative sum, so the value S_i would represent the sum of the first i elements of X, and for each query the answer would be given by $S_b - S_{a-1}$. This last method will need n operations to generate the array S, and for each query we can get the answer in constant time. So the running time for this algorithm would be $O(n + q)$.

A solution that makes use of the *Divide and Conquer* technique consists on using a *Segment Tree*, which is a binary tree where the leaf elements corresponds to the elements in X, and the rest of the nodes store the sum of the elements at its left and the elements at its right. The tree structure makes the thing easier for us, because when a node is inside the interval $[a, b]$, we just retrieve the value of that node, otherwise we must verify the children nodes until we get to a node contained in the interval $[a, b]$.

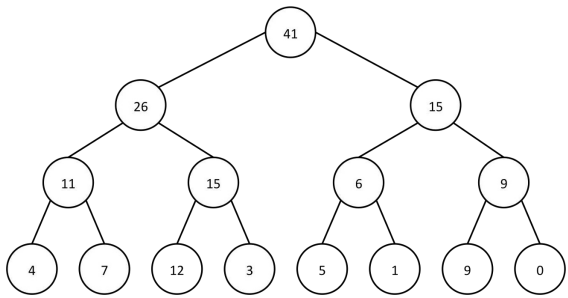


Figure 11.1: Example of a segment tree, where a parent node stores the sum of its children nodes.

Consider the case where $X = [4, 7, 12, 3, 5, 1, 9]$, the *Segment Tree* for this array is showed in figure 11.1. The first thing to notice is that the number of elements in the array is not a power of two, so one option is to fill the missing elements with 0's in order to make it a full binary tree. Now suppose we want to calculate the sum of elements with index in the interval $[1, 5]$. The search starts in the root, some elements are in the left side of the root and others in the right side, so the search continues in both sides. The node with value 26 covers the range $[0, 3]$, so the right side is inside the interval, but the left is not, so we will add 15 and 7 from the searches

in the right and left sub-trees respectively. When the search returns to the root by the recursion we will have $7 + 15 = 22$ for the sum in the left side. On the other hand, when the search in the right side of the root reaches node with value 6 that value is returned, since the interval is covered, so the answer is given by $22 + 6 = 28$.

For each query the *Segment Tree* will take $O(\log n)$ time to get the result, then the running time for the problem would be $(q \log n)$, without considering the cost of building the binary tree. In conclusion, there are different approaches for the same problem, for this specific case the *Dynamic Programming* solution would be the best of the three, but there are sometimes where a *Segment Tree* is a better fit.

2. The optimal strategy is to execute a double *Binary Search*, one for the rows and another for the columns, that would give us an upper bound for the number of opportunities needed to win the prize. Then if $\log_2 n + \log_2 m \leq k$, we can assure that the participant can win the prize, otherwise we cannot be sure.

3. For this problem we can use the RMQ algorithm, taking advantage of the fact that the sequence X is given in non-decreasing order. For example, suppose we have the following sequence:

X =	-1	-1	1	1	1	1	3	10	10	10

We can build another array F containing the frequency of all elements in the given sequence, which would look like this.

F =	1	2	1	2	3	4	1	1	2	3

Then we can run the RMQ algorithm to find the greatest value in F inside a given interval. We just need to be careful when the initial value of the interval is greater than 1, for example, the most frequent number between X_4 and X_7 is 1 with a total of two occurrences, but the value returned by the RMQ algorithm will be 3. In order to handle this problem we can build another array E, with E_k indicating the index

of the last appearance of element X_k. For the example above the array E would be as follows:

$$\mathrm{E} = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 1 & 5 & 5 & 5 & 5 & 6 & 9 & 9 & 9 \\ \hline \end{array}$$

Let be $k = \min(E_a, b)$, The solution for the problem is given by

$$answer = \begin{cases} max(RMQ(a,k) - F_a + 1, RMQ(k+1,b)) & k < b \\ RMQ(a,k) - F_a + 1 & k = b \end{cases}$$

where $RMQ(a,b)$ returns the maximum value in $F_a, \ldots, F_b$.

Listing 11.9: Divide & Conquer. Exercise 3

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#define N 100001
using namespace std;

long long X[N], Y[N], E[N];
long long M[N][20];
long long n, q;

long long answerQuery(long long, long long);

int main() {
  long long a, b, k, x2, val1, val2;

  scanf("%lld %lld", &n, &q);
  for (long long i = 0; i < n; i++) {
    scanf("%lld", &X[i]);

    if (i > 0 && X[i] == X[i - 1]) {
      Y[i] = Y[i - 1] + 1;
    } else {
      Y[i] = 1;
    }
  }

  E[n - 1] = n - 1;
  for (long long i = n - 2; i >= 0; i--) {
    if (X[i] == X[i + 1]) {
      E[i] = E[i + 1];
    } else {
      E[i] = i;
    }
  }

  for (long long i = 0; i < n; i++) {
    M[i][0] = i;
  }

```

```
  // compute values from smaller to bigger intervals
  for (long long j = 1; 1 << j <= n; j++) {
    for (long long i = 0; i + (1 << j) - 1 < n; i++) {
      if (Y[M[i][j - 1]] > Y[M[i + (1 << (j - 1))][j - 1]]) {
        M[i][j] = M[i][j - 1];
      } else {
        M[i][j] = M[i + (1 << (j - 1))][j - 1];
      }
    }
  }

  for (long long i = 0; i < q; i++) {
    scanf("%lld %lld", &a, &b);

    x2 = E[a];
    x2 = max(x2, b);
    val1 = answerQuery(a, x2);
    val2 = answerQuery(x2 + 1, b);

    if (val2 < 0) {
      k = Y[val1] - (Y[a] - 1);
    } else if (Y[val1] - (Y[a] - 1) > Y[val2]) {
      k = Y[val1] - (Y[a] - 1);
    } else {
      k = Y[val2];
    }

    printf("%lld\n", k);
  }

  return 0;
}

long long answerQuery(long long i, long long j) {
  long long ans, k;

  if (i > j) {
    return -1;
  }

  ans = 0;
  k = (long long)floor(log(double(j - i + 1)) / log(2.0));
  if (Y[M[i][k]] >= Y[M[j - (1 << k) + 1][k]]) {
    ans = M[i][k];
  } else {
    ans = M[j - (1 << k) + 1][k];
  }

  return ans;
}
```

11.5 Dynamic Programming

1. For $n = 1$ we have that there is only one way to fill the board of 2×1 with 2×1 domino tiles, so $f(1) = 1$. For $n = 2$ there are two ways, it can be two tiles vertically aligned, or two tiles horizontally aligned, then $f(2) = 2$. Based on these two

cases we can get the the result for any other n. For example, for $n = 3$ we can use the two ways for $n = 2$ and place a vertical tile at the end, also we can use the result for $n = 1$ and place two tiles horizontally at the end, then for $n = 3$ we have that $f(3) = f(2) + f(1) = 2 + 1 = 3$. Using the same principle we get that $f(n) = f(n-1) + f(n-2)$, for any $n > 2$. See figure 11.2. So we can say that the result is a Fibonacci sequence , but with the initial values being $f_1 = 1$, and $f_2 = 2$.

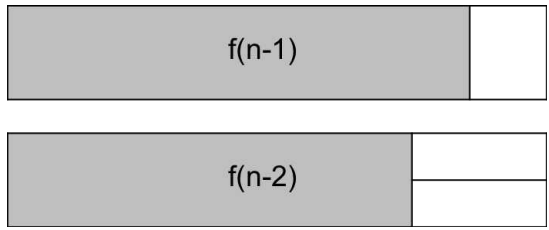


Figure 11.2: The number of ways to fill a board of size $2 \times n$ using domino tiles of 2×1 is equal to n^{th} Fibonacci number starting with $f_1 = 1$, and $f_2 = 2$.

2. If there are k *bad-luck* numbers with n digits, we can add any digit to the right of those numbers and they will still be *bad luck* numbers. Then the recursive formula should look something like this:

$$f(n) = 10f(n-1) + h(n-1)$$

The problem resides on finding the value of $h(n-1)$, which represents all numbers of $n-1$ digits that aren't *bad luck* numbers and whose right most digit is a 1, and adding a 3 to the right will generate $h(n-1)$ *bad luck* numbers of n digits.

For $n = 1$ we have that $f(1) = 0$, and $h(1) = 1$. Then $f(2) = 10f(1) + h(1) = 0 + 1 = 1$, so we have one *bad luck* number, which is the 13. Now we have to calculate the value of $h(2)$, for that we just need to add a 1 to the right of every non *bad luck* number, then $h(2) = 9$, (11, 21, 31, 41, 51, 61, 71, 81, 91). Then $f(3) = 10f(2) + h(2) = 19$, (130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 113, 213, 313, 413, 513, 613, 713, 813, 913).

3. For a cone with only one scoop there are two choices, chocolate or strawberry. With two scopes we still have two choices, chocolate-strawberry, or strawberry-chocolate. If we focus only in the last scoop, we can deduce that is valid to add chocolate above strawberry, or we can add strawberry above chocolate, also we can add a vanilla-chocolate pair above a strawberry scoop, or we can add a vanilla-strawberry pair above a chocolate scoop. Then for $n = 3$ we have four choices. Table 11.1 shows the number of possible ice cream cones up to 8 scoops based in the flavor at the top.

Flavor \ n	1	2	3	4	5	6	7	8
S	1	1	2	3	5	8	13	21
C	1	1	2	3	5	8	13	21

Table 11.1: Possible ice cream cones based on the flavor at the top.

The number of ways we can make an ice cream cone with n scoops is twice the n^{th} Fibonacci number.

4. There is only one way to reach any cell in the first column or in the first row. For the rest of the cells, be $X_{i,j}$ the number of ways to reach cell $A_{i,j}$, and is equal to the number of ways to reach the cell bellow, plus the number of ways to reach the cell at its left. Then

$$X_{i,j} = X_{i-1,j} + X_{i,j-1}.$$

Table 11.2 shows the number of ways to reach each one of the cells in a 4×4 matrix. There are 20 ways to reach cell $A_{4,4}$.

1	4	10	20
1	3	6	10
1	2	3	4
1	1	1	1

Table 11.2: Possible ways to reach a cell only by moving right and up.

5. This problem is very similar to the *Coin Change Problem* , with the difference that this time we don't have an infinite

amount of coins. Let's assume that Marie have at most 100 coins, and the coin with greatest denomination is of 20 pesos, so the maximum amount of money that Marie can have is 2000 pesos.

The idea is to compute if it is possible to obtain k pesos with the coins we have, and return the amount that is closest to the half of the money. In order to do that we need to set $X_0 = 1$, since by definition there is one way to obtain 0 pesos, and then we just need to traverse the array backwards instead of forwards for every coin. See line 25 of the code in 11.10. In that way we are only using the available coins, instead of using an infinite number of coins.

Listing 11.10: Dynamic Programming. Exercise 5

```
1  #include <cstdio>
2  using namespace std;
3
4  int C[101], X[1001];
5  int nCoins;
6
7  int main() {
8    int k, money, halfMoney;
9
10   money = 0;
11   scanf("%d", &nCoins);
12   for (int i = 0; i < nCoins; i++) {
13     scanf("%d", &C[i]);
14     money += C[i];
15   }
16
17   halfMoney = money / 2;
18   X[0] = 1;
19   for (int i = 0; i < nCoins; i++) {
20     k = C[i];
21     for (int j = halfMoney; j >= k; j--) {
22       X[j] |= X[j - k];
23     }
24   }
25
26   for (int i = halfMoney; i >= 0; i--) {
27     if (X[i] != 0) {
28       printf("%d %d\n", i, money - i);
29       break;
30     }
31   }
32
33   return 0;
34 }
```

11.6 Graph Theory

1. This is a classic problem in *Graph Theory*, and we can use a graph traversal method like *DFS* to solve it. First choose an initial node and paint it black or white, and start a *DFS* from there, when a node is explored, it is painted with the opposite color of the node where the search comes from, if an adjacent node results to have the same color then is impossible to color the graph, otherwise it is possible.

2. Yes. The two nodes with odd degree are the initial node and the destination node. The rest of the nodes, since all of them have an even degree, the path "enters" to a node from one edge and use another edge to "exit", and at the end all edges will be visited only once.

3. Yes. If all nodes have an even degree, then we can "enter" to every node from one edge and "exit" from another edge, at the end all edges will be visited and we will finish in the initial node, forming that way an Eulerian cycle .

4. Dijkstra's algorithm with node 0 as the initial node doesn't work for the following graph. The minimum cost to reach node 3 is obtained by visiting nodes $0 - 1 - 3$, with a cost of 3, but according to the algorithm the cost is 4. Because of its greedy behavior, Dijkstra's algorithm doesn't work for negative weights.

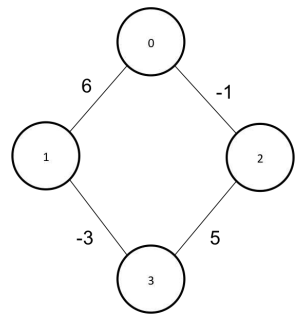


Figure 11.3: Example of a graph with negative weights that doesn't work for Dijkstra's algorithm.

5. A bipartite graph consists on two sets of nodes U and V in such a way that there is no edge connecting nodes of the

same set. We can convert the *Maximum Bipartite Matching* problem in a maximum flow problem by adding a *source* connected to all nodes in U, and a *sink* connected to all nodes in V. All the edges in the graph will have a capacity of 1. See figure 11.4, the edges in red are the ones that were added to the graph. Solving the maximum flow problem for the new graph will tell us the size of the maximum matching of the original graph.

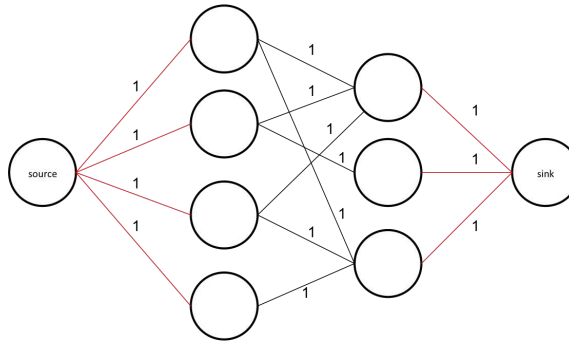


Figure 11.4: An example of a maximum bipartite matching problem transformed into a maximum-flow problem.

6. *BFS*. We can represent the building as a three-dimensional space where each location can be seen as a node. If the starting point is $X_{a,b,c}$, then we start adding that location to the queue, and then add those empty locations that are adjacent to it, and so on. To keep track of the cost we can mark $X_{a,b,c}$ with 0, and the locations adjacent to it with 1, and then with 2, we continue doing this until we reach the destination. In other words, if a person is in $X_{k,i,j}$, then locations $X_{k-1,i,j}, X_{k+1,i,j}, X_{k,i-1,j}, X_{k,i+1,j}, X_{k,i,j-1}, X_{k,i,j+1}$, will be marked with $X_{i,j,k}+1$, just if they are empty spaces. In that way the destination point will be marked with the cost of reaching that cell from $X_{a,b,c}$. It is important to notice that we initially must mark all the cells with a value representing that a cell has not been visited, a negative value is a good choice.

7. The adjacency matrix A contains all the paths of length 1 from any node to the another. A^2 represents the paths of length 2, A^3 the paths of length 3, and so on.

8. Given a network, a cut divides the vertices in two groups, where the source s is in group A, and the sink, t, is in group B. The flow though the cut, $F(A, B)$, is defined by the sum of the flows of the edges connecting nodes from A to B minus the flow of edges connecting nodes from B to A.

$$F(A, B) = \sum_{u \in A} \sum_{v \in B} f_{u,v} - \sum_{v \in B} \sum_{u \in A} f_{v,u},$$

and the capacity of the cut, $C(A, B)$ is represented as the sum of all capacities from all outgoing edges from A to B.

$$C(A, B) = \sum_{u \in A} \sum_{v \in B} c_{u,v}$$

This implies that

$$F(A, B) \leq C(A, B)$$

If there is no augmenting path from s to t we can separate the nodes in two groups, those that are reachable from s are in group A, and those that are not reachable from s forms group B, with s and t located in different groups. All outgoing edges from A to B are full, no more flow can pass through them, and all ongoing edges have zero flow. And since there is no augmenting path, the flow is maximum. Then

$$|f*| = F(A, B) = C(A, B)$$

where $|f*|$ represents the maximum flow. That give us a lower bound for the cut capacity, and that implies that the maximum flow is equal to the minimal cut.

11.7 Geometry

1. Choose two points and imagine a line that crosses both points, then search trough the rest of the points and see if they are crossed by that line too, if that happens increments a counter of collinear points. We can use the triangle area described before to identify if three points are collinear, just check if the area is zero. The process is repeated for every pair of points in the input.

Listing 11.11: Geometry. Exercise 1

```cpp
#include <cstdio>
#include <algorithm>
#define N 101
using namespace std;

class Point {
 public:
  int x;
  int y;
  Point(int x = 0, int y = 0) {
    this->x = x;
    this->y = y;
  }
};

Point P[N];
int n;

int area(Point, Point, Point);

int main() {
  int nCollinear, maxCollinear;

  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &P[i].x, &P[i].y);
  }

  maxCollinear = 0;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      nCollinear = 0;
      for (int k = 0; k < n; k++) {
        int A = area(P[i], P[j], P[k]);
        if (A == 0) {
          nCollinear++;
        }
      }

      maxCollinear = max(maxCollinear, nCollinear);
    }
  }

  printf("%d\n", maxCollinear);
  return 0;
}

int area(Point a, Point b, Point c) {
  return (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
}
```

2. Here we need to obtain the distance for every gopher to hawk, if that distance is less or equal to the hawk's radius attack, then increment a counter indicating the number of gophers in danger. For the i^{th} gopher we have to check if the following inequality holds.

$$\sqrt{(gopher[i]_x - hawk_x)^2 + (gopher[i]_y - hawk_y)^2} \leq R,$$

where R is the hawk's attack radius.

One thing that can help us to keep all operations with integer values is to square both sides of the inequality, in that way we don't have to worry about floating precision and the program will run faster, since we avoid the step of calculating the square root.

$$(gopher[i]_x - hawk_x)^2 + (gopher[i]_y - hawk_y)^2 \leq R^2.$$

Listing 11.12: Geometry. Exercise 2

```
#include <cstdio>
#include <algorithm>
#define N 101
using namespace std;

class Point {
 public:
  int x;
  int y;
  Point(int x = 0, int y = 0) {
    this->x = x;
    this->y = y;
  }
};

Point Gopher[N];
Point hawk;
int nGophers, R;

int euclideanDistance2(Point, Point);

int main() {
  int dis, gophersInDanger = 0;

  // Read the location of the hawk, the attack radius and
  // the number of gophers
  scanf("%d %d %d %d", &hawk.x, &hawk.y, &R, &nGophers);

  // Read the location of every gopher and calculate
  // the distance to the hawk.
  for (int i = 0; i < nGophers; i++) {
    scanf("%d %d", &Gopher[i].x, &Gopher[i].y);
    dis = euclideanDistance2(hawk, Gopher[i]);
    if (dis <= R * R) {
      gophersInDanger++;
    }
  }
```

```
38
39    printf("%d\n", gophersInDanger);
40    return 0;
41  }
42
43  int euclideanDistance2(Point a, Point b) {
44    return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
45  }
```

3. The area of a triangle with side lengths a, b, and c can be obtained by Heron's formula.

$$Area = \sqrt{s(s-a)(s-b)(s-c)},$$

where

$$s = \frac{a+b+c}{2}$$

4. This is known as the *Art Gallery Problem* . When the polygon is convex we only need one camera, the problem is when the polygon is non-convex like the one showed in 11.5a. An algorithm to obtain a solution for this problem is described bellow.

   (a) Triangulate the polygon. See figure 11.5b.

   (b) Construct a graph using the triangles from the last step, where each triangle represents a node, and if two triangles share a side then there is an edge connecting the corresponding nodes. See figure 11.5c.

   (c) Choose a node and mark the vertices of the corresponding triangle with different colors, in figure 11.5d we are using the letters R, G, B to identify the colors. Then start a *DFS* from the chosen node, and for every visited node mark all unmarked vertices of the corresponding triangle in such a way that two adjacent vertices don't have the same color.

   (d) Find the color less used (in the example are colors R and G) and place the cameras in the vertices marked with that color.

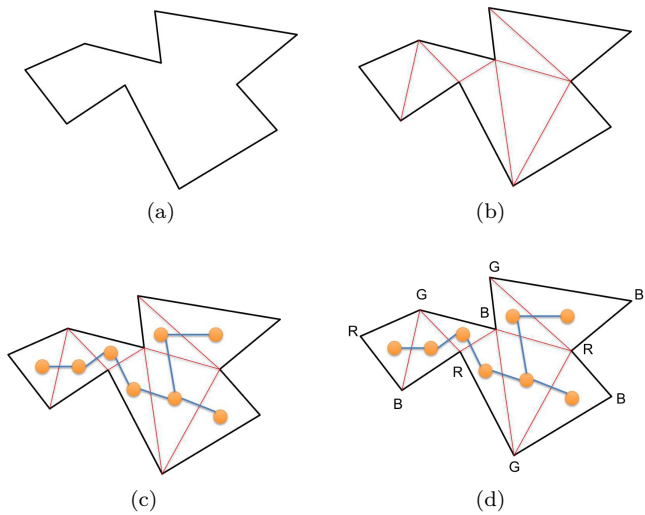


Figure 11.5: Algorithm to find a solution to the *Art Gallery Problem*

5. Using the solution explained in 4. Since we are coloring the n vertices of the polygon with three colors, and we place the cameras in the vertices marked with the least frequent color, then we would need at most $n/3$ cameras to watch over the city hall.

6. We need to find the minimum distance between each line segment of the polygon to the point. The minimum distance from line $ax + by + c = 0$ to point p is given by:

$$\frac{ap_x + bp_y + c}{\sqrt{a^2 + b^2}}$$

11.8 Number Theory and Combinatorics

1. Since $n!$ can be a very large number, it would be difficult to store it in a variable and then factorize it. One option is to obtain the prime factors of all numbers from 1 to n and then sort them. For example, 10! can be written as:

$$\begin{aligned}10! &= 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10 \\ &= 1 \times 2 \times 3 \times (2 \times 2) \times 5 \times (2 \times 3) \times 7 \times (2 \times 2 \times 2) \\ &\times (3 \times 3) \times (2 \times 5) \\ &= 1 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 3 \times 3 \times 3 \times 3 \times 5 \times 5 \\ &= 2^8 \times 3^4 \times 5^2\end{aligned}$$

2. This problem is known as *"Fermat's Last Theorem"* or *"Fermat's Conjecture"* . Some people called it *the most difficult mathematical problem*, and remained unsolved for centuries. Until in 1995 Professor Andrew Wiles published the proof, stating that there is no triplet of numbers a, b, c, such that, $a^n + b^n = c^n$, for $n > 2$.

   One of the mysteries surrounding this problem is that Fermat wrote in a margin of a book that he had a proof, but that it was too large and there wasn't enough space to write it down. The full Fermat's statement reads *"Cubum autem in duos cubos, aut quadrato-quadratum in duos quadrato-quadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos eiusdem nominis fas est dividere cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet"*. Which translated says *"It is impossible to represent a cube as the sum of two cubes, a fourth power as the sum of two fourth powers, or in general any number that is power greater than the second as the sum of two like powers. I have a marvelous proof for this problem that this margin is too narrow to contain."*

3. For this problem we would use modular arithmetic and the *Divide and Conquer* technique. Consider the following matrix F:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

   If we multiply F by a 2×1 vector containing two consecutive Fibonacci numbers f_{k-1} and $f_k$we get

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} f_{k-1} \\ f_k \end{bmatrix} = \begin{bmatrix} f_k \\ f_{k-1} + f_k \end{bmatrix} = \begin{bmatrix} f_k \\ f_{k+1} \end{bmatrix}$$

This way we can obtain the following Fibonacci number. Now what happens if we calculate the value of F^2 and F^3?

$$F^2 = F \times F = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$
$$F^3 = F^2 \times F = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$$

Notice that F^n has the following form

$$F^n = \begin{bmatrix} f_{n-2} & f_{n-1} \\ f_{n-1} & f_n \end{bmatrix}$$

for $n > 1$, and with $f_0 = 1, f_1 = 1$. Now, to calculate the n^{th} Fibonacci number modulus m, we just need to obtain the value of F^n, which can be done efficiently by using binary exponentiation, and because there are only multiplications and additions, we can use the properties of modular arithmetic.

4. The least common multiple of two numbers a, and b is equal to their product divided by the their greatest common divisor. Then

$$mcm(a, b) = \frac{ab}{gcd(a, b)}$$

The following code use the Euclidean algorithm to obtain the least common multiple of two numbers.

Listing 11.13: Number Theory and Combinatorics. Exercise 4

```
#include <cstdio>
#include <algorithm>
using namespace std;

int gcd(int, int);

int main() {
  int a, b, mcm;

  scanf("%d %d", &a, &b);
  mcm = (a * b) / gcd(a, b);
  printf("%d\n", mcm);

```

```
  return 0;
}

int gcd(int a, int b) {
  int temp;

  if (a < b) {
    swap(a, b);
  }

  while (b > 0) {
    temp = a % b;
    a = b;
    b = temp;
  }

  return a;
}
```

5. This is a special case of the *Extended Euclidean Algorithm* called **Diophantine Equation** . The fist thing to do is to check if that equation has an integer solution, if it does then d must be divisible by the greatest common divisor of a and b.

   Be m the greatest common divisor of a and b, then we can rewrite the equation as follows

$$\left(\frac{m}{d}\right)(ax + by) = d\left(\frac{m}{d}\right)$$
$$\left(\frac{m}{d}\right)(ax + by) = m$$
$$ax^* + by^* = m$$

   where

$$x^* = \frac{m}{d}x$$
$$y^* = \frac{m}{d}y$$

   The problem is now reduced to the *Extended Euclidean* problem, and we can get the values of x^* and y^* that solve the new equation, multiply them by d/m and we get the values of x and y that solve the original equation.

6. The *Goldbach's Conjecture* states that every even integer n greater than 2 can be expressed as the sum of two prime numbers. The conjecture was written in 1742 in a letter to Euler, and to this day it has not been proved, making it one of the most famous problems in mathematics.

   To solve this specific problem, where $n \leq 1000$, one option is to identify all prime numbers up to 1000 using the Sieve of Eratosthenes. This will generate an array P, where $P_k = 0$ if k is prime, otherwise $P_k \neq 0$. After that we can assign values to a variable a such that:

   $$n = a + b$$

   Solving for b we have that

   $$b = n - a$$

   If $P_a = 0$ and $P_b = 0$, then n can be expressed as the sum of prime numbers a and b.

   Listing 11.14: Number Theory and Combinatorics. Exercise 6

```cpp
#include <cstdio>
#include <algorithm>
#define N 1001
using namespace std;

int P[N];

void sieve();

int main() {
  int n;

  sieve();
  scanf("%d", &n);

  for (int i = 2; i < N; i++) {
    int j = n - i;
    if (P[j] == 0) {
      printf("%d + %d = %d\n", i, j, n);
      break;
    }
  }

  return 0;
}

void sieve() {
```

```
    P[0] = P[1] = 1;
    for (int i = 2; i <= 32; i++) {
      if (P[i] == 0) {
        for (int j = i * i; j < N; j += i) {
          P[j] = 1;
        }
      }
    }
  }
```

11.9 String Manipulation

1. There are different ways to solve this problem, one option is to reverse the string and compare if the resulting string is equal to the original, if that happens then it is a palindrome. That solution involves to duplicate the memory space since another variable is needed to store the reversed string. A better solution is to place an index i at the beginning of the string, and another index j at the end, then start moving forward the index i and backwards the index j, if in any moment $S_i \neq S_j$ then S it is not a palindrome, once both indexes cross each other the search can be stopped and affirm that S is a palindrome.

Listing 11.15: String Manipulation. Exercise 1

```
bool isPalindrome(string S) {
  int n = S.length();

  for (int i = 0, j = n - 1; i < n / 2; i++, j--) {
    if (S[i] != S[j]) {
      return false;
    }
  }
  return true;
}
```

2. The trivial solution consists on generating all possible words and check if they are palindromes, and return the longest one. For example consider the string $S = xkaaky$. The palindrome *"kaak"* is inside S and the words that can be formed are:

x
xk
xka
$xkaa$
$xkaak$
$xkaaka$
k
ka
kaa
$kaak$
...

Generating all those words takes $O(n^2)$ time, and check if a string is a palindrome, as we seen in exercise 1 takes $O(n)$ time. Then the running time of this approach is $O(n^3)$.

A better solution is to start the search from position k supposing that S_k is the middle of the palindrome, this in the case where the palindrome has odd length. On the other hand we can start the search from positions k and $k+1$, just if $S_k = S_{k+1}$. That said, for the previous example, if we look for a palindrome of even length we must look for two equal letters placed one after the other, in this case the letters *"aa"*, and start the search from there, for that we place an index i at the left of the first "a", and an index j at the right of the second "a", and keep moving the indexes, i backward, and j forward, while $S_i = S_j$ then there is a palindrome in $[i, j]$. This approach runs in $O(n^2)$ time.

3. Two strings are anagrams if they contain the same letters, but in different order. There are more than one way to check if two strings are anagrams, bellow we list two of them.

   - Sort both strings lexicographically, if they are anagrams then the resulting strings must be the equal.
   - Use an array C of 26 elements, initially all in zeros. For every letter in the first string add one to the corresponding position in C, a is zero, b is one, and so on. Then for every letter in the second string subtract one to the corresponding position in C. If both string were anagrams then C must contain only zeros.

4. The C library `ctype` contains very useful functions to be used with `char` variables. For this specific problem we can use the functon `isupper` that receives a `char` and returns *true* if the character is upper case, otherwise returns *false*.

Listing 11.16: String Manipulation. Exercise 4

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main() {
  string str;
  int nMayus, nMinus;

  cin >> str;
  nMayus = 0;
  for (int i = 0; i < str.length(); i++) {
    if (isupper(str[i])) {
      nMayus++;
    }
  }

  nMinus = str.length() - nMayus;
  cout << nMayus << " " << nMinus << "\n";
  return 0;
}
```

5. For this problem we can make us of the ASCII code. For each letter subtract the value of letter 'A', this way 'A' would have a value of zero, 'B' a value of 1, and so on. After that add k to that value and obtain the modulus 26 (because there are 26 letters). Finally add the value of 'A' in order to display the correct character.

Listing 11.17: String Manipulation. Exercise 5

```
string caesar(string S, int k) {
  for (int i = 0; i < S.length(); i++) {
    int c = ((S[i] - 'A') + k) % 26;
    S[i] = c + 'A';
  }
  return S;
}
```

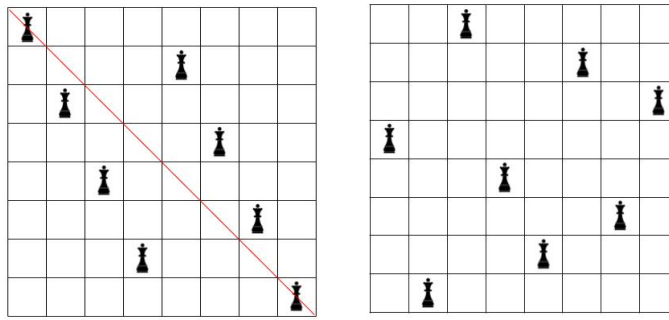
Appendix A

Recursion and Bitwise

In this section we will work on two problems, one about recursion and the other about bitwise. We will go trough a solution for these problems and analyze their complexity.

A.1 The 8-Queen problem

Given an empty chess board, place 8 queens in such a way that none of the queens threatens another queen. We say that a queen threaten another queen if both are in the same row, or same column, or same diagonal. Figure A.1a shows a wrong solution following a simple strategy, placing the first queen in the top-left corner, the next queen is placed in the next column and two rows below the current one, once we reach the bottom of the board, we place the next queen in the next column and in the first available row (row with no queen) starting from the top, and repeat the process. As we can see, the queen in the first column and the queen in the last column are in the same diagonal, which violates the rules of the problem. On the other hand figure A.1b corresponds to a correct solution of the problem, since no queen threatens another in the board.



(a) Invalid solution (b) valid solution

Figure A.1: left. An invalid solution since two queens are in the same diagonal. right. A valid solution where no queen threaten another.

We will use recursion to find a solution for the 8-queen problem. The strategy consists on placing only one queen in each column, and check if there is no other queen in the same row and in the four possible diagonals.

We define our chess board as a matrix of integers, X, where each cell can have a value of 0 or 1, indicating an empty cell or a queen respectively. To assure that only one queen will be placed in a row we can just add the values of that row, if the sum is zero means that we can place a queen in that row. The function bellow identify how many queens are located to the left of position (i, j), this can be used to determine if cell (i, j) is a valid location to place a queen.

```
1 int checkLeft(int i, int j) {
2   int k = 0;
3   for (int c = j - 1; c >= 0; c--) {
4     k += X[i][c];
5   }
6
7   return k;
8 }
```

If the value returned by the function is zero, means that there is no queen at the left of that cell. We can do a similar thing to check at the right of the cell by using function *checkRight*, and if both values, *checkLeft* and *checkRight* are zero, it means that it is safe to place a queen in cell (i, j).

```
int checkRight(int i, int j) {
  int k = 0;
  for (int c = j + 1; c < 8; c++) {
    k += X[i][c];
  }

  return k;
}
```

Now, it looks like we are repeating code, which is a bad practice. Let's integrate both functions in one function called *getNumberOfQueens*.

Listing A.1: 8-Queen Problem Validation

```
int getNumberOfQueens(int i, int j, int deltaX, int deltaY) {
  int r = i + deltaY;
  int c = j + deltaX;
  int k = 0;

  while (r >= 0 && r < 8 && c >= 0 && c < 8) {
    k += X[r][c];
    r += deltaY;
    c += deltaX;
  }

  return k;
}
```

The variables *deltaX* and *deltaY* in code A.1 specify the step size between columns and rows respectively. For example, to check if there is a queen at the left of cell $(4, 4)$, we call *getNumberOfQueens(4, 4, -1, 0)*. Since *deltaX* is -1, then we will move to the left cell by cell summing all values. On the other hand, if we want to know how many queens are at the right we do *getNumberOfQueens(4, 4, 1, 0)*. But why do we do this? isn't this more complicated? The answer is, for simplicity of the code, since we can use the same function to also find the number of queens on the diagonals. For example, if we call *getNumberOfQueens(4, 4, 1, 1)* we get the number of queens in the diagonal that goes down and right from $(4, 4)$, since *deltaX* is 1, and *deltaY* is 1, this causes to "move" in X one position right and one position down on each iteration of the *while* loop.

Code A.2 find all solutions for the 8-queen problem, it consists on placing one queen on column *column*, the algorithm iterates through each one of the rows and verify if it is a valid position, and if it is, a queen is placed on that cell and the same process starts in the next column. When the recursion returns to the current

cell, the queen is removed from that cell (because we already put it there), and continue the search for valid positions across the remaining rows.

Listing A.2: Solution to 8-Queen Problem

```
void solve8QueenProblem(int column) {
  int nQueens;

  if (column == 8) {
    // We have placed the 8 queens
    printBoard();
    return;
  }

  for (int i = 0; i < 8; i++) {
    nQueens = getNumberOfQueens(i, column, -1, 0);
    nQueens += getNumberOfQueens(i, column, 1, 0);
    nQueens += getNumberOfQueens(i, column, -1, -1);
    nQueens += getNumberOfQueens(i, column, 1, -1);
    nQueens += getNumberOfQueens(i, column, -1, 1);
    nQueens += getNumberOfQueens(i, column, 1, 1);

    // If there are no queens, then (i, column) is a safe location to place a
    // queen
    if (nQueens == 0) {
      X[i][column] = 1;             // Place a queen
      solve8QueenProblem(column + 1); // Move to the next column
      X[i][column] = 0;             // Remove the queen
    }
  }
}
```

The function *getNumberOfQueens* makes the code simpler and readable. The algorithm calls the function to obtain the number of queens at the left, right and at each one of the four diagonals of a specific location. The *printBoard* function has the purpose to display the chess board on the screen, it can be done on different ways, the code bellow is just one way of doing it.

```
void printBoard() {
  for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
      printf("%d ", X[i][j]);
    }
    printf("\n");
  }

  printf("===== END OF SOLUTION =====\n");
}
```

Let's assume that the time complexity of checking if a queen is in the same row is $O(1)$, and that diagonals will be ignored. Given that, be n the size of the chess board, so in the first column we have n possible rows to place a queen, in the second column

we have $n-1$ possible rows, since we cannot place a queen in the same row as the previous one. In the third column we have $n-2$ possible rows, then $n-3$, $n-4$, and so on, until we reach the last column with only one row available. Our recursion tree will have $n!$ nodes, and the time complexity would be $O(n!)$. Now, unfortunately for solution A.2, checking if a cell is valid takes $O(n)$, since it iterates through rows and diagonals. and this is done on each row on every column. So for each column the time complexity is $O(n^2)$, since there are n rows. The process is repeated every time *solve8QueenProblem* is called. This makes that the time complexity of the solution presented to be $O(n^{2n})$.

This solution can be improved if we optimize the way of checking if a position is valid or not. Some ideas of doing this, avoid checking at the right, since the algorithm iterates from left to right, we can be sure that there are no queens in the right side. Also if we use an array to flag the rows that already have a queen, we can know if there is a queen in a given row in $O(1)$. Also some solutions are mirrors of another solutions, we can use that property to reduce our recursive tree.

A.2 Vacations

You are planning your next vacations, and you have narrowed the possible destinations to N different cities numbered from 0 to $N-1$. You have asked to your friends and family for suggestions, but their opinion about a city can be different, your best friend said that she likes city 0, but your brother said that city 0 is awful, and you should go to city 2 instead, but your dad said that city 2 is too boring, and that in city 1 you can find the best tacos in the world. You are more confused than ever, so you have decided to visit a city that none of your friends or family has visited before.

Input

The input of this problem consists of a number N ($2 \leq N \leq 20$), indicating the number of possible destinations for your vacations, and a number M ($1 \leq M \leq 10^6$), indicating the number of suggestions from your friends and family. The next M numbers are in the range $[0, N-1]$, representing the suggestions you get.

No one will suggest a place they haven't visited before.

Output

Print the cities that you will consider to visit on your vacations, that is, the cities that none of your friends or family have visited before. Print them in ascending order.

Solution

Something that immediately should caught our attention is that N is a small number, when that happens is good to try to understand why is that. For this specific case we can receive a high amount of suggestion for a small number of cities, meaning that some cities can be suggested more than once. It is not necessary to store all the suggestion in memory, and we are not interested on knowing who went to each place, we only care if a city was visited by someone (no matter who). Suppose that $N = 5$, and we get the following suggestions: 0, 1, 1, 0, 3. This means that two persons suggested city 1, other two suggested city 0 , and another suggested city 3, leaving city 2 and 4 as our possible destinations.

We can represent our suggestions as a boolean array X, with $X_k = 1$ if city k has been suggested by someone, otherwise $X_k = 0$. For the example above we would have

	X[4]	X[3]	X[2]	X[1]	X[0]
$X =$	0	1	0	1	1

Here we are using N integers in memory to represent our vacation destinations, well, taking advantage that N is small, we can represent the same information with a single integer. We can achieve this if we see X as a number, and its bits as the array cells, this is called **bit masking**, then for the same example we have that $X = 11$ which in binary is 01011, that way if we want to know if city k has been suggested, then we should look for the k^{th} bit. Code A.3 shows a solution for this problem using bit masking.

Listing A.3: Vacations: Use case of bit masking

```
1 #include <iostream>
2 using namespace std;
3
```

```cpp
int main() {
  int n, m, city;
  int bitmask = 0;

  cin >> n >> m;
  for (int i = 0; i < m; i++) {
    cin >> city;
    bitmask |= (1 << city); // activate the corresponding bit of city
  }

  for (int i = 0; i < n; i++) {
    // Check if the i-th bit is activated
    if ((bitmask & (1 << i)) == 0) {
      cout << i << "\n";
    }
  }

  return 0;
}
```

Appendix B

Data Structures Problems

This appendix contains problems where choosing the right data structure is indispensable to implement the optimal solution.

B.1 Find two numbers whose sum is k

Given an array X of positive integers, and a number k, write a program that determine if there are two numbers in the array whose sum is equal to k.

Input

The first line contains an integer n ($2 \leq n \leq 10^6$) indicating the amount of numbers in the array. The next n numbers represents the elements of the array, all elements will be positive. Finally the last line consists of the integer k.

Ouput

If k can be represented as the sum of two elements of the array, print those values, otherwise prints -1.

Solution

Given that the value of n is too large, the $O(n^2)$ solution of checking all possible pairs is discarded.

The $O(n)$ solution consists on going trough all the elements of the array and subtract the current element to k, if the result is also inside of X, then we have a solution. In order to verify if a number is in X we can use a map, that operation would take $O(1)$. For this approach we would need to do a first pass trough X to store its elements in the map.

Is important to avoid using the same number twice in our solution, for example, if $k = 10$ and $X = [2, 3, 5]$, $5 + 5$ is not a valid answer, since there is only one 5 in X. In program B.1 we handled that scenario by using the frequency of each number as value of the map, then to avoid repeating a number we just decrease by 1 its value. Below is a summary of the what we need to implement:

1. Insert every element in X to map M.
2. For each element x_i in X check if $k - x_i$ is in M, if it is, a solution to the problem is the pair $x_i, k - x_i$, only if they are not the same element.
3. If no pair was found print -1.

Listing B.1: Find two numbers that sum k

```cpp
#include <iostream>
#include <map>
#include <vector>
using namespace std;

int main() {
  int n, num, k;
  vector<int> X;
  map<int, int> M;

  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> num;
    X.push_back(num);
    M[num]++; // Increment num frequency in map
  }
  cin >> k;

  for (int i = 0; i < n; i++) {
    num = X[i];
    M[num]--; // Remove X[i] to avoid using it twice
    if (M[k - num] > 0) {
      cout << num << " " << k - num << "\n";
      return 0;
    }
    M[num]++; // Re-insert X[i]
  }
```

```
28
29    cout << "-1\n";
30    return 0;
31  }
```

B.2 Shunting-yard Algorithm

The shunting-yard algorithm is used to convert an infix expression into a postfix expression. This algorithm was developed by Edsger Dijkstra and it uses a stack of operators to reorder the expression. The rules are the following:

1. If the incoming symbol is an operand, print it.
2. If the incoming symbol is a left parenthesis, add it to the stack.
3. If the incoming symbol is a right parenthesis, print all the symbols in the stack until a left parenthesis appear. Pop that left parenthesis.
4. If the incoming symbol is an operator, continue to pop symbols from the stack and print each one of them until a left parenthesis appears, or until an operator with lower priority appears. Add the incoming symbol to the stack.
5. Finally, pop and print the rest of the elements in the stack.

Write a program that changes an infix expression to a postfix expression.

Input

The input expression is given one character per line. For example, $(7 + 4) * 5$ would be in the form:
(
7
+
4
)
*
5

The program will handle the binary operators $+, -, *, /$, and the operands will be one-digit numerals. The operators $*$ and $/$ have the highest priority. The operators $+$ and $-$ have the lowest priority. Parentheses have the function of grouping symbols that override the operator priorities. The input ends with a blank line, and there will be no more than 50 lines in the input.

Output

The output is the postfix expression all on one line.

Sample Input	Sample Output
(3 + 2) * 5	$32 + 5*$

Solution

For this problem we just need to follow the rules described in the problem statement. See code B.2.

Listing B.2: Shunting-yard Algorithm

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

int main() {
  int i, j, n, m;
  char car;
  string str;
  stack<char> S;

  while (getline(cin, str) && str.length() > 0) {
    car = str[0];
    if (car >= '0' && car <= '9') {
      cout << car; // is digit, print it
    } else {
      if (S.size() == 0) {
        S.push(car);
      } else {
        if (car == ')') {
          // Pop everything until left parenthesis appear
          while (S.top() != '(') {
            cout << S.top();
            S.pop();
```

```
          }
          S.pop(); // pop the left parenthesis
        } else if (car == '(') {
          S.push(car); // left parenthesis - add to stack
        } else {
          while (!S.empty()) {
            // Stop pop if left parenthesis appear
            if (S.top() == '(') {
              break;
            }
            if (car == '+' || car == '-') {
              cout << S.top(); // Continue printing the top element
            } else if (car == '*' || car == '/') {
              if (S.top() == '*' || S.top() == '/') {
                cout << S.top(); // Print operator if priority is the same
              } else {
                break; // Stop if priority is lower
              }
            }
            S.pop();
          }
          // Add the operator into the stack
          S.push(car);
        }
      }
    }
  }

  // Print remaining elements in the stack
  while (!S.empty()) {
    cout << S.top();
    S.pop();
  }
  cout << "\n";

  return 0;
}
```

B.3 Find the median

Given an array X of n integers, the median of X is the middle element after all elements are sorted in increasing order, only if n is odd, otherwise the median is obtained calculating the average of the two elements at the middle.

For example, the median of elements $1, 3, 6, 8, 10$ is 6. On the other hand, the median of elements $1, 3, 6, 8, 9, 10$ is 7 $((6 + 8)/2)$.

Write a program that read and array X of n integers and print the median as elements are inserted into the array.

Input

The input consists of a number n ($1 \leq n \leq 10^5$), indicating the amount of elements in the array. n lines follow, each one with one integer, representing the elements in the array, ($0 \leq X_i \leq 10^5$).

Output

n numbers indicating the median as elements are inserted into the array. Each number must be in a separate line.

Solution

To solve this problem we can use the fact that the elements at the left of the median are all smaller or equal than the median, and the elements at the right of the median are larger or equal then the median.

If we use a heap to store the elements at the left, the root of that heap will be the largest of all those elements smaller or equal than the median. Equally for the right side, but using a min-heap instead (a node value is smaller or equal than its children values), so the root represents the smallest of all numbers equal or larger than the median.

The idea is to maintain both heaps balanced. The amount of elements between them cannot differ by more than one. If both heaps have the same size, then the median is the average between the roots of the two heaps, if the left heap is larger, then the median is the root of the left heap, otherwise the median would be the root of the right heap.

To implement the min-heap we can just multiply by -1 its elements and handle it as regular heap.

Listing B.3: Find the median

```
#include <cstdio>
#include <queue>
#define N 100001
using namespace std;

int n;
int A[N];
priority_queue<int> LH, UH;

void getMedians();
```

```
11
12  int main() {
13    scanf("%d", &n);
14    for (int i = 0; i < n; i++) {
15      scanf("%d", &A[i]);
16    }
17
18    getMedians();
19    return 0;
20  }
21
22  void getMedians() {
23    int k;
24    double median;
25
26    // The first median is the first number of the array
27    printf("%.1lf\n", (double)A[0]);
28
29    // Add the first number to the lower heap
30    LH.push(A[0]);
31
32    for (int i = 1; i < n; i++) {
33      if (A[i] <= LH.top()) {
34        LH.push(A[i]); // Add A[i] to the lower heap
35      } else {
36        UH.push(-A[i]); // Add A[i] to the upper heap
37      }
38
39      // Do we have more elements in the lower heap?
40      if ((int)LH.size() - (int)UH.size() >= 2) {
41        k = LH.top();
42        LH.pop();    // Remove the largest element of the lower heap
43        UH.push(-k); // Add it to the upper heap
44      } else if ((int)UH.size() - (int)LH.size() >= 2) {
45        k = -UH.top();
46        UH.pop();   // Remove the smallest element of the upper heap
47        LH.push(k); // Add it to the lower heap
48      }
49
50      // Get the median
51      if ((int)LH.size() == (int)UH.size()) {
52        median = (LH.top() - UH.top()) / 2.0;
53      } else if ((int)LH.size() > (int)UH.size()) {
54        median = (double)LH.top();
55      } else {
56        median = -1.0 * UH.top();
57      }
58
59      printf("%.1lf\n", median);
60    }
61  }
```

Appendix C

Sorting Problems

In this section we present special cases of sorting problems that can be solved using the algorithms covered in in this chapter, but that don't have a trivial solution. This with the objective that the reader be aware that there can be other approaches when facing problems that involves sorting, solutions that sometimes are easier to implement and have better performance. Said that, we encourage the reader to try to solve the problems first, before reading the solution.

C.1 Marching in the school

N students are standing in one line, some of them are facing left, and other facing right. All of them must face to the same direction, so when a student see the face of another student understands that he has made a mistake and turns around. The process continues until all the students don't see any other student's face. Write a program that calculates the number of times when a pair of students turned around. If the process is infinite, print "NO".

Table C.1: Marching in the school

Formation	Comments	Number of turns
>><<><<	Initial formation	2
><><<>	One second has passed	2
<><><>	Two seconds has passed	2
<<><>>	Three seconds has passed	1
<<<>>>	Final formation	0
		Total: 7

Input

The first line of the input contains the number of students N ($1 \leq N \leq 30000$). The rest of the input contains only " < ", " > " characters. There is exactly N " < " and " > " characters in the input file.

Output

Write the number of turns.

Sample Input	Sample Output
6 >><<><<	7

Solution

The solution for this problem consists on identifying when a < character appears, and when that happens increase the result by the number of > characters before it. This way we are just counting the swaps needed without implementing a sorting algorithm. This is an example of just answering what is asked, avoiding more complicated implementations.

Listing C.1: Marching in the School

```
#include <cstdio>
using namespace std;

int main() {
  long n, k, r, sum;
  int c;

  scanf("%ld", &n);

  k = 0;
```

```
  r = 0;
  sum = 0;
  while (k < n) {
    c = getc(stdin);
    if (c == '>' || c == '<') {
      k++;
    }

    if (c == '>') {
      r++;
    } else if (c == '<') {
      sum += r;
    }
  }

  printf("%ld\n", sum);
  return 0;
}
```

C.2 How Many Swaps?

Given an array X of n elements write a program that determines the number of swaps needed to sort the array X using *Bubble Sort*.

Input

The first line of the input contains n ($1 \leq n \leq 100$), indicating the number of elements in the array X. The next n numbers represent the elements of X, all of them 32-bit integers.

Output

The number of swaps made by the *Bubble Sort* method.

Solution

This example is similar to the previous one. One option is to implement the *Bubble Sort* algorithm and count the number of swaps that occur. Another option is to count the number of misplaced elements in the array. The later option consists on counting for each element in the array, the number of elements at its right that are smaller. Consider the following array

$$X = [3, 7, 2, 6, 1, 4, 3, 5]$$

If we look at the elements at the right of the first element (3), we notice that there are two elements that are smaller (1 and 2).

So we increase our counter by two. We do the same for the second element (7), this time we have six elements smaller than it, so we increase our result by six. If we follow the same process for all elements we obtain that our answer is:

$$2 + 6 + 1 + 4 + 0 + 1 + 0 + 0 = 14$$

Listing C.2: How Many Swaps

```
#include <cstdio>
#define N 100
using namespace std;

int main() {
  int i, j, n, ans;
  int X[N];

  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    scanf("%d", &X[i]);
  }

  ans = 0;
  for (i = 0; i < n; i++) {
    for (j = i + 1; j < n; j++) {
      if (X[i] > X[j]) {
        ans++;
      }
    }
  }

  printf("%d\n", ans);
  return 0;
}
```

C.3 Closest K Points to the Origin?

The statement of this problem is simple, given n coordinates of points, find the k points that are closest to the origin.

Input

Tho numbers n and k, followed by n lines, each one with two numbers x and y, defining the coordinates of a point.

We are not going to define any constraints in the input for this problem, to simulate a real interview.

Output

Print the k points closest to the origin, starting with the nearest one, and ending with the furthest. Print each point in a separate line.

Solution

Let's try to solve this problem as if we were in a real interview. So, the first think to know is how large the value of n can be. If you ask this to the interviewer, probably will say "assume a very large number".

Since this chapter is about sorting algorithms let's go with the following solution.

1. Store all numbers in an array.
2. Sort the array according to the distance to the origin.
3. Print the first k elements of the array.

The time complexity of this solution in the best scenario is $O(n \log n)$ if we use an algorithm like Merge Sort or Heap Sort. We will see later that there is a better solution, but first let's try this one. Don't forget to tell all these steps to the interviewer, remember that is important that they know what and how are you thinking.

For step 1 we can use a `Point` class to store the point's coordinates and the distance to the origin, then we can just define an array of Point objects where we can store the points as we read the input.

```
class Point {
 public:
  int x;
  int y;
  int d2;

  bool operator<(const Point &b) const { return this->d2 < b.d2; }
};
```

To avoid dealing with floating numbers, we can use the square of the Euclidean distance to the origin and keep everything with integers. The closest point, which has the smallest Euclidean

distance, will also have the smallest square Euclidean distance.

For step 2 we can use a sorting algorithm with time complexity of $O(n \log n)$, for that we can use the `sort` function, which requires to overload the $<$ operator of the `Point` class.

Finally for step 3, we can just iterate through the array and print the first k elements of the array. The solution will look like this:

Listing C.3: Closest K Points to the Origin

```cpp
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

class Point {
 public:
  int x;
  int y;
  int d2;

  bool operator<(const Point &b) const { return this->d2 < b.d2; }
};

int main() {
  int n, k;
  Point point;
  vector<Point> P;

  scanf("%d %d", &n, &k);

  for (int i = 0; i < n; i++) {
    scanf("%d %d", &point.x, &point.y);
    point.d2 = point.x * point.x + point.y * point.y;
    P.push_back(point);
  }

  sort(P.begin(), P.end());

  for (int i = 0; i < k; i++) {
    printf("%d %d\n", P[i].x, P[i].y);
  }

  return 0;
}
```

This same problem can be found in the Exercises section of this chapter, we encourage the reader to try to come up with a better solution. Hint: $O(n \log k)$.

Appendix D

Divide and Conquer Problems

In this part we are going to use some of the algorithms seen in this section to solve problems. As we mentioned before, *Divide and Conquer* is just a tool that can make our life easier when trying to solve a specific problem, sometimes is easy to see where can be applied, but sometimes it is not a plain sight and reacquires more analysis and scratching your head a little while.

D.1 Polynomial Product

The following program computes the product of two polynomials and prints the resulting polynomial.

Input

The first line contains two numbers n, and m indicating the number of coefficients of polynomials A and B respectively. Meaning that A has a degree of $n-1$, meanwhile B has a degree of $m-1$

The next n numbers represents the coefficients of A, and the next m numbers represents the coefficients of B.

Output

The coefficients of the product of polynomials A and B. The resulting polynomial will have a degree of $n + m - 2$.

Solution

To solve this problem we need to apply the *Fast Fourier Transform* for polynomial multiplication as described before. The class `ComplexNumber` in code D.1 represents a complex number with the operators overloaded to handle operations between complex numbers. The method `SquareDiff` returns the square of its magnitude, meanwhile the method `bar` returns its conjugate.

Listing D.1: Polynomial Multiplication (FFT)

```
#include <cstdio>
#include <cmath>
#include <cstring>
#define MAX (1 << 19)
using namespace std;

class ComplexNumber {
 public:
  double a;
  double b;

  ComplexNumber(double a = 0.0, double b = 0.0) {
    this->a = a;
    this->b = b;
  }

  double SquareDiff() const { return a * a + b * b; }
  ComplexNumber bar() const { return ComplexNumber(this->a, -this->b); }
  ComplexNumber operator+(ComplexNumber b) const {
    return ComplexNumber(this->a + b.a, this->b + b.b);
  }
  ComplexNumber operator-(ComplexNumber b) const {
    return ComplexNumber(this->a - b.a, this->b - b.b);
  }
  ComplexNumber operator*(ComplexNumber b) const {
    return ComplexNumber(this->a * b.a - this->b * b.b,
                         this->a * b.b + this->b * b.a);
  }
  ComplexNumber operator/(ComplexNumber b) const {
    ComplexNumber r = ComplexNumber(this->a, this->b) * b.bar();
    return ComplexNumber(r.a / b.SquareDiff(), r.b / b.SquareDiff());
  }
};

const double two_pi = 4 * acos(0);
int n, m;
double C[MAX + 100]; // Cos array
double S[MAX + 100]; // Sin array
ComplexNumber a[MAX + 100], b[MAX + 100];
ComplexNumber A[MAX + 100], B[MAX + 100];
ComplexNumber P[MAX + 100], INV[MAX + 100];
```

The function `angle` returns the complex number w_n^k for a given k. The value of n for this case is defined in the constant `MAX`. Meanwhile the `FFT` function has five parameters, `in`, which represents the coefficients of the polynomial, `out` refers to the resulting vector y after applying the FFT, `step` is a power of two that is used to get the correct points, `size` is the number of points we will use, remember that in each iteration that quantity is reduced by half. Finally `dir` is 1 if we are obtaining the FFT, and -1 for the inverse FFT (FFT^{-1}).

```
ComplexNumber angle(int dir, int k) {
  return ComplexNumber(C[k], dir * S[k]);
}

void FFT(ComplexNumber *in, ComplexNumber *out, int step, int size, int dir)
     {
  if (size < 1) {
    return;
  }

  if (size == 1) {
    out[0] = in[0];
    return;
  }

  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);

  for (int i = 0; i < size / 2; i++) {
    ComplexNumber even = out[i];
    ComplexNumber odd = out[i + size / 2];
    out[i] = even + angle(dir, i * step) * odd;
    out[i + size / 2] = even - angle(dir, i * step) * odd;
  }
}
```

The `main` function reads the coefficients of both polynomials A and B, then calculates the points $w_n^0, \ldots, w_n^{n-1}$. Once we have the values of w_n we can apply the FFT to both polynomials and represent them as complex numbers. That will allow us to multiply them element by element. The result is then used to calculate the inverse FFT and transform it from complex number representation to coefficients.

```
int main() {
  int temp;

  scanf("%d %d", &n, &m);

  temp = 0;
  memset(a, 0, sizeof(a));
  memset(b, 0, sizeof(b));
  for (int i = 0; i < n; i++) {
    scanf("%d", &temp);
```

```
    a[i] = temp;
  }

  for (int i = 0; i < m; i++) {
    scanf("%d", &temp);
    b[i] = temp;
  }

  // Generate Complex Numbers
  for (int i = 0; i <= MAX; i++) {
    C[i] = cos(two_pi * i / MAX);
    S[i] = sin(two_pi * i / MAX);
  }

  // Get the FFT of coefficientes a and b
  FFT(a, A, 1, MAX, 1);
  FFT(b, B, 1, MAX, 1);

  // Multiply FFT(a) * FFT(b)
  for (int i = 0; i < MAX; i++) {
    P[i] = A[i] * B[i];
  }

  // Calculate the FFT inverse of P
  FFT(P, INV, 1, MAX, -1);


  // Scale the coefficients
  for (int i = 0; i < MAX; i++) {
    INV[i] = INV[i] / MAX;
  }

  for (int i = 0; i < n + m - 1; i++) {
    printf("%.2lf ", INV[i].a);
  }
  printf("\n");

  return 0;
}
```

D.2 Wi-Fi Connection

The residents of the main street of Kusatsu want to install a Wi-Fi connection on the street, so that every house has Internet access. Given the number of houses in the street, the number of routers and the locations of the houses, write a program that finds where they should place the routers. The signal should be as strong as possible in each house. They would like to place the routers so that the maximum distance between any house to the router closest to it is as small as possible. Keep in mind that the street is a perfectly straight road.

Input

The first line contains two positive integers n, the number of routers, and m, the number of houses on the street. The following m lines contain distance of each house to the beginning of the street There will be no more than 100000 houses, and no house numbers is located more than one million meters from the beginning of the street.

Output

A line containing the maximum distance between any house and the router nearest to it. Round the number to the nearest tenth of a metre, and output it with exactly one digit after the decimal point.

Sample Input	Sample Output
2 4 1 6 7 31	3.0

Solution

One possibility is to solve this problem using binary search , place a coverage range of L which is the distance from the last house to the first house of the street. If you can cover all the houses decrease the length by half, otherwise increase the range by half. Keep doing that until you find the right range. To print it with the right decimal places just multiply all the distances by 10 at the beginning.

Listing D.2: Wi-Fi Connection (Binary Search)

```
#include <cstdio>
#include <algorithm>
#define N 100001
using namespace std;

long x[N];
long n, m;

bool isAllStreetWithWiFi(long);

int main() {
  long a, b, mid;
```

```
13
14    scanf("%ld %ld", &m, &n);
15    for (long i = 0; i < n; i++) {
16      scanf("%ld", &x[i]);
17      x[i] *= 10;
18    }
19
20    sort(x, x + n);
21
22    if (m >= n)
23      printf("0.0\n");
24    else {
25      a = 0;
26      b = x[n - 1] - x[0];
27
28      while (a < b - 1) {
29        mid = (a + b) / 2;
30        if (isAllStreetWithWiFi(2 * mid)) {
31          b = mid;
32        } else {
33          a = mid;
34        }
35      }
36
37      printf("%ld.%ld\n", b / 10, b % 10);
38    }
39
40    return 0;
41  }
42
43  bool isAllStreetWithWiFi(long coverage) {
44    long nRouters = 1;
45    long wifiRange = x[0] + coverage;
46
47    for (long i = 0; i < n; i++) {
48      if (x[i] > wifiRange) {
49        nRouters++;
50        wifiRange = x[i] + coverage;
51      }
52    }
53
54    return nRouters <= m;
55  }
```

Appendix E

Graph Theory Problems

Some of the algorithms seen in the chapter *Graph Theory* can be modified to solve specific kind of problems. In this section we will review some of these problems.

E.1 Minmax and Maxmin

The algorithms of minmax and maxmin are variants of shortest path algorithms. In minmax, the edge with maximum value is found for each path and returns the minimum of those values. In a similar way, the maxmin algorithm finds the edge with minimum value for all paths and returns the largest among those values. Following there are two examples that illustrates the concepts of minmax and maxmin algorithms.

E.1.1 Credit Card (minmax)

For the graph in E.1 suppose that Johnny wants to travel from city A to city G using his credit card. That credit card has a limit of K dollars, and Johnny can use it as many times he wants as long he doesn't surpass K dollars in a single purchase.

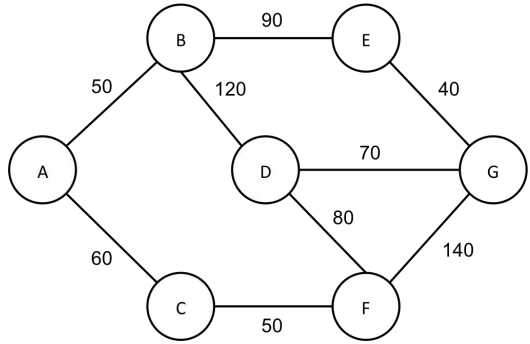


Figure E.1: Toll fares of roads across cities.

To get from city A to city G Johnny may follow the following path: $A - C - F - G$. In that case the credit card limit must be at least 140 dollars. For the paths $A - B - E - G$, $A - B - D - G$ and $A - C - F - D - G$ The credit card's limit must be at least of 90, 120 and 80 dollars respectively. There are other paths, too. However, it is clear that $A - C - F - D - G$ is the one that needs a minimum value of K of 80 dollars.

Given the road connections between cities and the toll fare of each one of the roads, we must find the minimum value of K that is necessary to Johnny to get from a starting city to a destination city. The first line contains three numbers C ($2 \leq C \leq 100$), S ($1 \leq S \leq 1000$) and Q ($1 \leq Q \leq 10000$), indicating the number of cities, roads, and queries respectively. S lines follow, each one with three numbers a, b and c, meaning that there is a road connecting city a with city b, with a toll fare of c dollars. Finally there are Q lines, each one with two numbers C_{start} and C_{end} indicating the starting city and the destination city for Johnny.

For each query we must print the minimum value of K necessary to reach the destination city. In case there is no path just print "no path".

The solution for this problem is showed in E.1 and consists on just applying the minmax algorithm. They are asking for the minimum edge weight among the maximum edge weight of all the paths. To accomplish this one option is to apply the Floyd-Warshall algorithm modified.

$$W_{ij} = \min(W_{ij}, \max(W_{ik}, W_{kj}))$$

The only condition is to initialize the weighted matrix W with ∞, except for the main diagonal that will remain with 0's.

Listing E.1: Minmax Algorithm

```
#include <cstdio>
#include <algorithm>
#define MAX 32767
using namespace std;

int w[101][101];

void initialize(int);
void floydWarshall(int);

int main() {
  int C, S, Q, a, b, d, c1, c2;

  scanf("%d %d %d", &C, &S, &Q);
  initialize(C);

  for (int i = 0; i < S; i++) {
    scanf("%d %d %d", &c1, &c2, &d);
    w[c1][c2] = d;
    w[c2][c1] = d;
  }

  floydWarshall(C);

  for (int i = 0; i < Q; i++) {
    scanf("%d %d", &a, &b);
    if (w[a][b] == MAX) {
      printf("no path\n");
    } else {
      printf("%d\n", w[a][b]);
    }
  }

  return 0;
}

void initialize(int n) {
  for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= n; j++) {
      w[i][j] = (i == j) ? 0 : MAX;
    }
  }
}

void floydWarshall(int n) {
  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        w[i][j] = min(w[i][j], max(w[i][k], w[k][j]));
      }
    }
  }
```

```
53 }
```

E.1.2 LufeMart (maxmin)

Mr. Lufe is the owner of a big store called *LufeMart* that can be found in different locations across the country. Mr. Lufe wants to send a cargo from one city to another, but the amount of cargo weight that can be transported changes depending on the road.

Given the start and destination cities, along with the weight constraints of the roads, our job is to determine the maximum load that can be transported between the two specified cities. In other words we must find the edge with minimum cost for each path that goes from the origin to the destination, and return the edge with maximum cost among them.

For the graph in E.1, if each edge is a road and the cost of each edge is the weight restriction for that road, the maximum load that can be transported from city A to city G is 50 tons. Because even when we can transport 60 tons from A to C, from C to F we can only transport 50 tons, so we would have to get rid of 10 tons. The paths $A-B-D-G$, $A-B-D-F-G$, $A-C-F-G$, and $A-C-F-D-G$, are solutions for this specific case.

The code in E.2 reads two numbers n ($2 \leq n \leq 200$) and r ($1 \leq r \leq 19900$) representing the number of cities and the number of roads respectively. r lines follow, each one with two strings a, b and one number c, indicating that there is a road connecting city a with city b and with a weight constraint of c tons. Finally there are two strings representing the start and destination cities. The output of the program is the maximum number of tons that can be transported from the start city to the destination city. The name of the cities only contains lower-case letters and don't have more than 30 characters.

Any algorithm to find the shortest path in a graph can be used to solve this problem. In E.2 Dijkstra's algorithm is used, it just needs that the weighted matrix W to be initialized with 0's, and instead of finding the minimum value across the non-visited nodes, it looks for the maximum. The vector is then updated using the following formula:

$$D_j = \max(D_j, \min(D_k, W_{kj})),$$

where k is the position of the maximum value in D among the non-visited vertices.

Listing E.2: Maxmin Algorithm

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#define N 201
#define MAX 2147483647
using namespace std;

long w[N][N];
long s[N], d[N];
char city[N][35];

long getIndex(char *, long);
void addInList(char *, long);
void dijkstra(long, long, long);
long maxValue(long);

int main() {
  long i, j, k, n, r, a, b, cont = 1;
  char cad[35], cad2[35];

  scanf("%ld %ld", &n, &r);
  memset(w, 0, sizeof(w));

  j = 0;
  for (i = 0; i < r; i++) {
    scanf("%s %s %ld", cad, cad2, &k);

    a = getIndex(cad, j);
    if (a < 0) {
      addInList(cad, j);
      a = j;
      j++;
    }

    b = getIndex(cad2, j);
    if (b < 0) {
      addInList(cad2, j);
      b = j;
      j++;
    }

    w[a][b] = k;
    w[b][a] = k;
  }

  scanf("%s %s", cad, cad2);

  a = getIndex(cad, j);
  b = getIndex(cad2, j);

  dijkstra(a, b, n);

  printf("%ld tons\n\n", d[b]);
```

```
  return 0;
}
```

The function `getIndex` receives a string `cad` and the number of cities already stored in the array `city`. If `cad` is in `city`, it returns its position in the array, otherwise returns -1. On the other hand, the function `addInList` receives a string `cad` and an integer `pos`, and stores `cad` in the array `city` in position `pos`.

```
long getIndex(char *cad, long n) {
  for (long i = 0; i < n; i++) {
    if (!strcmp(cad, city[i])) {
      return i;
    }
  }
  return -1;
}

void add_in_list(char *cad, long pos) { strcpy(city[pos], cad); }
```

To perform the *maxmin* we used a modified Dijkstra's algorithm. Notice that instead of finding the minimum element in vector `d` like in the standard version of Dijkstra, now we find the maximum value across the non-visited nodes, for that we use the function `maxValue`. Also the way the vector of distances is updated is different, since for this case we store the maximum weight that can be transported from city to city `i`.

```
void dijkstra(long a, long b, long n) {
  long pos;

  for (long i = 0; i < n; i++) {
    d[i] = w[a][i];
    s[i] = 0;
  }

  s[a] = 1;
  while (s[b] == 0) {
    pos = maxValue(n);
    s[pos] = 1;
    for (long i = 0; i < n; ++i) {
      if (s[i] == 0) {
        d[i] = max(d[i], min(d[pos], w[pos][i]));
      }
    }
  }
}

long maxValue(long n) {
  long pos, max;

  max = 0;
  pos = 0;
  for (long i = 0; i < n; ++i) {
    if (s[i] == 0 && d[i] > max) {
```

```
28        max = d[i];
29        pos = i;
30      }
31    }
32    return pos;
33  }
```

E.2 Money Exchange Problem

The money exchange problem consists on changing certain amount of money between different currencies, change it again to the initial currency, and check if there is a profit.

Each currency represents a node, and the exchange rate between two currencies is the cost of the edge connecting those currencies. See the following example

Example. Forex

Forex is the foreign exchange market, and arbitrage consists of buying an asset and selling it to profit from a difference in the price, so in the case of Forex, it can be defined as the profit obtained from currency exchange rates to transform one unit of a currency into more than one unit of the same currency.

The program E.3 reads an integer n ($1 \leq n \leq 30$) representing the number of different currencies. n lines follow, each one with the name of each currency. The next line consists of an integer m representing the possible exchanges that can be made. Each of the next m lines contains a string a, a real number r, and another string b, representing the exchange rate from currency a to currency b. The program prints "Yes" if a profit is possible, otherwise prints "No". The name of the currencies consist of only letters and their length don't exceed 200 characters.

To solve this problem we can modify one of the algorithms to find the shortest path in a graph. For this case we used *Floyd-Warshall* with a change in the way the matrix of weights is updated, which consists on multiplying the values in the matrix instead of adding them, because to change from one currency to another is

necessary to multiply by the exchange rate. See E.1.

$$W_{ij} = \max\left(W_{ij}, W_{ik} \times W_{kj}\right) \tag{E.1}$$

Another important modification takes place in the initialization of the matrix of weights, and that is that the main diagonal must be filled with $1's$, and the rest with $0's$, because exchanging from some currency to the same currency should give us the same amount, and exchanging to a currency that is not possible to convert, it should return no profit at all.

Listing E.3: Money Exchange Problem

```
#include <cstdio>
#include <cstring>
#include <algorithm>

// Definitions
#define N 31
#define MAX 100000000.0
using namespace std;

// Global Variables
char coin[N][255];
double w[N][N];

// Function Prototipes
void initialize(int);
int getCoin(char *, int);
void floydWarshall(int);

int main() {
  int i, n, m, a, b, aux, count = 1;
  double c;
  char cad1[255], cad2[255];

  scanf("%d", &n);

  initialize(n);
  for (i = 0; i < n; i++) {
    scanf("%s", coin[i]);
  }

  scanf("%d", &m);
  for (int i = 0; i < m; i++) {
    scanf("%s %lf %s", cad1, &c, cad2);
    a = getCoin(cad1, n);
    w[a][b] = c;
  }

  floydWarshall(n);

  aux = 0;
  b = getCoin(cad2, n);
  for (i = 0; i < n; i++) {
    if (w[i][i] > 1.0) {
      aux = 1;
```

```
      break;
    }
  }

  if (aux == 1) {
    printf("Yes\n");
  } else {
    printf("No\n");
  }

  return 0;
}
```

The only objective of the `initialize` function is to set the initial values of the matrix `w`, for this case the main diagonal has $1's$, since there are no profit changing to same currency. The rest cells contain $0's$, to make them unsuitable to convert money to those currencies.

```
void initialize(int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      w[i][j] = (i == j) ? 1.0 : 0.0;
    }
  }
}
```

The `getCoin` function look for a given string `cad` inside the `coin` array, and returns its position. In case `cad` is not found it returns -1.

```
int getCoin(char *cad, int n) {
  for (int i = 0; i < n; i++) {
    if (!strcmp(coin[i], cad)) {
      return i;
    }
  }
  return -1;
}
```

The function `floydWarhsall` is a modified version of the *Floyd-Warshall's* algorithm, using multiplications instead of sums in order to obtain the maximum profit to change the from one currency to another.

```
void floydWarshall(int n) {
  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        w[i][j] = max(w[i][j], w[i][k] * w[k][j]);
      }
    }
  }
}
```

Appendix F

Number Theory Problems

This section contains problems with non-trivial solutions and in order to solve them is needed to make use of some properties and concepts instead of using a brute force approach.

F.1 Sum of Consecutive Numbers

Given an integer n, represent n as the sum of k consecutive integers. See the following examples:

$$\begin{aligned} 9 &= 2+3+4 \\ 17 &= 8+9 \\ 22 &= 4+5+6+7 \\ 15 &= 1+2+3+4+5 \\ 8 &= 8 \end{aligned}$$

The solution consists on finding two numbers a and b, such that

$$\begin{aligned} n &= \frac{b\,(b+1)}{2} - \frac{a\,(a+1)}{2} \\ 2n &= b(b+1) - a(a+1) \end{aligned} \tag{F.1}$$

where $b > a$, and there is a number k such that $b = a + k$. So we can express the equation F.1 as follow

$$\begin{aligned} 2n &= (a+k)(a+k+1) - a(a+1) \\ &= k^2 + 2ak + k \\ &= k(k+2a+1) \end{aligned} \tag{F.2}$$

The code in F.1 read an integer n and prints n as the sum of consecutive integers. This by using F.2 and trying different values of k and obtaining a until a valid solution is found.

For the case when $a = 0$, we have that $k = b$, which is obtained by:

$$n = \frac{b(b+1)}{2}$$

Solving for b we have that

$$b = \frac{1 + \sqrt{1+8n}}{2} = k.$$

That is the upper bound of k, so the time complexity of the solution would be $O(\sqrt{n})$.

Time Complexity: $O(\sqrt{n})$
Input:
`n`. An integer.
Output:
n as the sum of consecutive integers.

Listing F.1: Sum of Consecutive Integers

```
#include <cstdio>
#include <cmath>
using namespace std;

int main() {
  long long n, m, a, b, k;

  scanf("%lld", &n);
  m = 2 * n;
  k = (long long)((sqrt(8.0 * n + 1.0) + 1.0) / 2.0);

  for (long long i = k; i >= 1; i--) {
    if (m % i == 0) {
      a = m / i - (i + 1);
      if (a % 2 == 0) {
        a /= 2;
```

```
      } else {
        continue;
      }

      b = a + i;
      if (a >= 0 && b >= 0 && (a + b + 1) == m / i) {
        printf("%lld = %lld + ... + %lld\n", n, a + 1, b);
        break;
      }
    }
  }

  return 0;
}
```

F.2 Two Queens in Attacking Positions

In how many ways can you place two queens in a chessboard of $n \times m$, so they attack each other? Two queens attack each other if they are in the same row, column, or diagonal.

The solution in F.2 reads two integers, n and m, indicating the size of the chessboard, and prints the number of ways two queens can be places in attacking positions.

The first step is to assure that $m \geq n$, that will help us to avoid some validations in our code. Now let's consider the case where both queens are in the same row. There are $m(m-1)$ ways to do it, now considering all rows, we have $nm(m-1)$ different ways. We do the same for the columns and obtain that there are $mn(n-1)$ different ways to place two queens in the same column.

In the board there are $2(m-n+1)$ diagonals with n cells. Then there are $2(m-n+1)n(n-1)$ ways to place two queens in attacking position in those diagonals.

For the rest of the diagonals, there are four diagonals of size $n-1$, other four with size $n-2$, and so on. we will call S to the number of ways of placing two queens in attacking position in those diagonals, and is given by:

$$
\begin{aligned}
S &= 4((n-1)(n-2) + (n-2)(n-3) + \cdots (2)(1)) \\
&= 4\left(\sum_{k=1}^{n-2} k(k+1)\right) \\
&= 4\left(\sum_{k=1}^{n-2} k^2 + \sum_{k=1}^{n-2} k\right) \\
&= 4\left(\frac{(n-2)(n-1)(2n-3)}{6} + \frac{(n-2)(n-1)}{2}\right) \\
&= 4\left(\frac{n(n-1)(2n-1)}{6} - \frac{n(n-1)}{2}\right)
\end{aligned}
$$

Time Complexity: $O(1)$
Input:

`n`. Number of rows in the chess board

`m`. Number of columns in the chess board

Output:

The number of ways of placing two queens in attacking positions.

Listing F.2: Two Queens in Attacking Positions

```
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
  unsigned long long n, m, temp;
  unsigned long long a, b, c, d;

  scanf("%llu %llu", &n, &m);
  if (m < n) {
    swap(n, m);
  }

  a = m * (m - 1) * n;             // 2 queens in the same row
  b = n * (n - 1) * m;             // 2 queens in the same column
  c = 2 * n * (n - 1) * (m - n + 1); // 2 queens in a diagonal of size n
  d = n * (n - 1) * (2 * n - 1) / 6 -
      n * (n - 1) / 2; // 2 queens in a diagonal of
  d = 4 * d;          // size k (2<=k<n)

  printf("%llu\n", a + b + c + d);

  return 0;
}
```

F.3 Example. Sum of GCD's

Given an integer n, find the sum of all *greatest common divisors* (GCD's) of any pair of numbers a and b (where $a \neq b$) smaller or equal to n.

Be $D(x) = d_1, \cdots, d_n$ a set of all the divisors of x less than x. The function $f(x)$ that returns the sum of all GCD's of any pair of numbers smaller than x is given equation F.3.

$$f(x) = f(x-1) + \sum_{i=1}^{n} d_i \phi(x/d_i), \tag{F.3}$$

where $\phi(x)$ is the Euler's function defined in 9.3.

For F.3 is clear to see that $f(x)$ will be at least $f(x-1)$, because the pair of numbers that are smaller than $x-1$ there are also smaller than x, but the value of $\sum_{i=1}^{n} d_i \phi(x/d_i)$ is not that clear, and is easier to understand with an example. If $x = 6$, then $D(6) = \{1, 2, 3, 4, 6\}$, and we know that

$$\sum_{i=1}^{n} gcd(6, i) = 1 + 2 + 3 + 2 + 1 + 6 = 15$$

Then the result of $\sum_{i=1}^{n} d_i \phi(x/d_i)$ should be 15 as well. Expanding the sum we obtain that

$$\sum_{i=1}^{n} d_i \phi(x/d_i) = 1\phi(6) + 2\phi(3) + 3\phi(2) + 6\phi(1) = 2 + 4 + 3 + 6 = 15$$

Both results are the same, but in the second way the values of $\phi(x)$ can be precalculated and saved in memory, and then we can do something similar to the *Sieve of Eratosthenes* to obtain the values of $f(x)$.

F.4 Find B if LCM(A,B) = C

This problem is very simple, given A and C we must write a program that finds the value of B such that

$$LCM(A, B) = C$$

where $LCM(A, B)$ is the lowest common multiple of A and B.

First we must check if C is divisible by A, if not, there is no solution, otherwise, to solve this problem we need to create two hash maps M_1 and M_2, the first will contain the prime factors of A as keys, and the number of occurrences as values. M_2 will contain the prime factors of C as keys, and the number of occurrences as values.

The next step consists of iterate trough all the elements of M_2, if the values are different in both maps for the same key (prime number p), the result is multiplied by p the number of occurrences in M_2. Is important to mention that the variable with the result must be initialized with 1.

For the case where $A = 4$ and $C = 12$ we have that

$$4 = 2 \times 2 = 2^2$$
$$12 = 2 \times 2 \times 3 = 2^2 \times 3^1,$$

, and the value of B would be

$$B = 3^1 = 3.$$

Getting that $LCM(4, 3) = 12$, which is correct. The program in F.3 reads two numbers A and C $(A, C \leq 32000)$, and prints the value of B if there is a solution, otherwise prints "NO SOLUTION".

Time Complexity: $O(\sqrt{(}n))$
Input:
Two positive integers A and C.
Output:
The value of B, such as $LCM(A, B) = C$. If there is no solution prints the message *"NO SOLUTION"*.

Listing F.3: Find B if *LCM(A,B)* = C

```
#include <iostream>
#include <map>
#include <vector>
#define MAX 32000
using namespace std;

map<long long, long long> M1, M2;
vector<long long> P;

void generatePrimes();
```

```
void factorize(long long, bool);

int main() {
  long long A, B, C;

  generatePrimes();

  cin >> A >> C;

  if (C % A != 0) {
    cout << "NO SOLUTION" << endl;
    return 0;
  }

  factorize(A, true);
  factorize(C, false);

  B = 1;
  for (auto it = M2.begin(); it != M2.end(); it++) {
    if (it->second != M1[it->first]) {
      for (long long i = 0; i < it->second; i++) {
        B *= it->first;
      }
    }
  }

  cout << B << endl;
  return 0;
}
```

The `factorize` function finds the prime factors of a given number *num*. In case `flag` is true, then all factors are added to map `M1`, otherwise they are added to map `M2`.

```
void factorize(long long num, bool flag) {
  long long k = 0;

  while (num >= P[k] * P[k]) {
    if (num % P[k] == 0) {
      num /= P[k];

      if (flag) {
        M1[P[k]]++;
      } else {
        M2[P[k]]++;
      }
    } else {
      k++;
    }
  }

  if (flag) {
    M1[num]++;
  } else {
    M2[num]++;
  }
}
```

The function `generatePrimes` stores in vector `P` all prime num-

bers up to `MAX`. Those prime numbers will be used to find the prime factors of any number up to `MAX*MAX`.

```
void generatePrimes() {
  P.push_back(2);
  P.push_back(3);

  for (long long num = 5; num < MAX; num += 2) {
    for (long long i = 0; i < P.size(); i++) {
      if (num % P[i] == 0) {
        break;
      } else if (num < P[i] * P[i]) {
        P.push_back(num);
        break;
      }
    }
  }
}
```

F.5 Last Non Zero Digit in n!

Given a number n we must find the last non zero digit in $n!$, and because of n can be very large is impossible to compute the value of $n!$. The trick for this problem is to realize that the numbers that generates 0's are the product of the numbers 2 and 5. Thus, we have to get rid of those products in order to find the last non zero digit in $n!$. For example:

$$\begin{aligned} 20! &= 2432902008176640000 \\ &= 1 \times 2 \times 3 \times \cdots \times 20 \\ &= 2^{18} \times 3^8 \times 5^4 \times 7^2 \times 11 \times 13 \times 17 \times 19 \end{aligned}$$

For this case we have 18 2's and 4 5's, so we need to remove four 2's and four 5's, resulting in

$$2^{14} \times 3^8 \times 7^2 \times 11 \times 13 \times 17 \times 19 = 243290200817664,$$

which is 20! without trailing zeros, so we only need to keep track of this product modulus 10. The program in F.4 reads a number n and prints the last non zero digit in $n!$.

Time Complexity: $O(n \log n)$
Input:
A number n ($n \geq 0$).

Output:

One number indicating the last non-zero digit in $n!$.

Listing F.4: Last non zero digit in $n!$

```
#include <cstdio>
using namespace std;

int main() {
  long n, num, f;
  long nFives, nTwos;

  scanf("%ld", &n);

  nFives = 0;
  for (long i = 5; i <= n; i *= 5) {
    nFives += n / i;
  }

  nTwos = 0;
  f = 1;
  for (long i = 1; i <= n; i++) {
    num = i;
    while (num % 5 == 0) {
      num /= 5;
    }

    while (num % 2 == 0 && nTwos < nFives) {
      num /= 2;
      nTwos++;
    }

    f = (f * num) % 10;
  }

  printf("%5ld -> %ld\n", n, f);
  return 0;
}
```

Bibliography

[1] Ronals L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

[2] G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146(2):263–266, 1962.

[3] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software—Practice and Experience*, 24(3):327–336, 1994.

[4] René de la Briandais. File searching using variable length keys. *Proc. Western J. Computer Conf*, pages 295–298, 1959.

[5] Robert Sedgewick. *Algorithms in C++*. Pearson Addison-Wesley, 1 edition, 1995.

[6] timmac. Topcoder. data structures. https://www.topcoder.com/community/competitive-programming/tutorials/data-structures/. Accessed: 2021-01-15.

[7] C.A.R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM.*, 4(7):321–322, 1961.

[8] J.W.J. William. Algorithm 232 (heapsort). *Communications of the ACM*, 7(6):347–348, 1964.

[9] Donald E. Knuth. *The Art of Computer Programming. Sorting and Searching*, volume 3. Addison-Wesley, 2 edition, 1998.

[10] timmac. Topcoder. sorting. https://www.topcoder.com/community/data-science/data-science-tutorials/sorting/. Accessed: 2017-08-01.

[11] danielp. Topcoder. range minimum query and lowest common ancestor. https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/. Accessed: 2017-07-27.

[12] R.C. Chang R.C.T. Lee, S.S. Tseng and Y.T. Tsai. *Introduction to the Design and Analysis of Algorithms.* McGraw Hill, 2007.

[13] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[14] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[16] Jr. Lestor R. Ford and D. R. Fulkerson. Flows in networks. *Princeton University Press*, 1962.

[17] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[18] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM.*, 9(1):11–12, 1962.

[19] P. E. Hart, N. J. Nilsson, and B. Rapahel. A formal basis for the heuristic, determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 4(2):100–107, 1968.

[20] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[21] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[22] Jr. Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[23] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[24] gladius. Introduction to graphs and their data structures. https://www.topcoder.com/community/data-science/data-science-tutorials/introduction-to-graphs-and-their-data-structures-section-1/. Accessed: 2017-08-02.

[25] A. M. Andrew. Another efficient for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[26] Ronald. L. Graham. An efficient algorithm for determining the convexh hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[27] Charles H. Lehmann. *Analytic Geometry*. Limusa, 1999.

[28] Gareth A. Jones and J. Mary Jones. *Elementary Number Theory*. Springer, 2005.

[29] Donald E. Knuth. *The Art of Computer Programming. Seminumerical Algorithms*, volume 2. Addison-Wesley, 3 edition, 1998.

[30] Donald E. Knuth. *The Art of Computer Programming. Combinatorial Algorithms*, volume 4A. Addison-Wesley, 1 edition, 2011.

[31] Charles H. Lehmann. *Algebra*. Limusa, 1999.

[32] Donald E. Knuth, Jr. James H. Morris, , and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

Index