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.

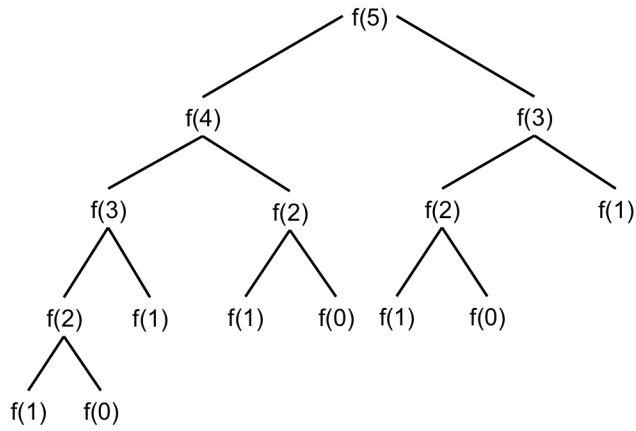

Figure 2.1: Fibonacci Recursion Tree

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.

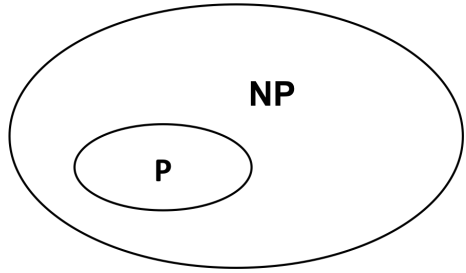

Figure 2.2: P and NP problems

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.

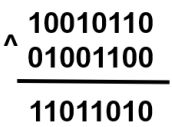

Figure 2.5: Example XOR operator

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
```