# Generating functions and searching automata

Author: Fermat

Research Advisors: Erdös and Cauchy

June 2009 - March 2011

# Acknowledgements

First of all, I would like to thank specially **Dr. Juanjo Rué**, PhD in analytical combinatorics working in l'*École Polytechnique*, Paris, and later in *Instituto de Ciencias Matemáticas*, Madrid, for all the time, dedication and effort he put in helping me with the research, giving the initial orientation, teaching me the great world of generating functions and solving my doubts with great quickness and interest. Without him, this work would certainly not be the same.

I would also like to give special thank to **Mrs. Laura Morera** for being my research adviser, presenting me Mr. Rué and Mr. Giménez, all the corrections she did and the opinions she gave about the act of researching. For a novice in research like me, her expert judgment and advices were of great importance.

I would also like to thank **Dr. Esther Silberstein**, for the great interest she manifested for my research those last two years, and specially for the two afternoons she shared with me, discussing it. At the same time, I would like to thank **Dr. Pere Cairó** for spending one afternoon with me solving my doubts about genetics.

I would also like to thank the Program ***Joves i Ciència*** and specially its director (and my teacher ), **Dr. Maria Calsamíglia** that influenced this work indirectly by introducing me to a first experience with scientific research and giving me some useful advise on the matter. Moreover, I would also like to thank the ***Ross Mathematics Program*** held at Ohio State University, which I attended the summer of 2010. This program, through classes done by **Dr. Daniel Shapiro**, whom I would also like to thank, and the work of two intensive months in math gave me a lot of insight in mathematics, and also shaped my idea of how to think and prove. Specially, I would like to thank **Zev Rosengarten**, my math counselor there and student at Princeton University, for spending literally more than a hundred hours going over math problems with me. All this style and learning received at *Joves i Ciència* and *Ross* are really reflected into this research.

I would also like to thank **Lander Ramos** for his help in concrete problems in the computer science part.**Mr. Joan Alemany** had also a great influence in this study with his advice in some mathematical and computer science points and the presentation of the study. I would like to thank him for that and the interested he showed for my research those months. **Mrs. Isabel Corominas** was also of great help with her advise on the presentation of the paper and the use of LATEX and I am very grateful for her help.

I would also like to thank **Dr. Omer Giménez**, first professor at the *Universitat Politècnica de Catalunya* and now working as a programmer at Google, for showing me an initial presentation of the idea of the orientation of the research, along with Dr. Rué.

I would also like to thank **Lluís Vena**, PhD student at *University of Toronto* and **Dr. Txema Tamayo** ,working at the UPC and the University of Toulouse, for attending a small presentation I did of my research, comment the research and the presentation itself and ask some questions about it, this winter. Moreover, I would like to thank both **Jie Luan** and other friends for sharing some hours with me talking about my work since explaining it and getting questions were a perfect way to see where it could be improved. At the same time, I would also like to thank my family to help me in multiple occasions when I was writing the study, making the work much easier and comfortable. In particular I would like to thank **my father** for the conversations he had with me about it and for going over my work.

Moreover, I would like to thank both ***Aula Escola Europea*** and the ***Universitat Politècnica de Catalunya*** for providing the place to do many of the necessary meeting for the realization of this research.

I would also like to thank **Núria Carreras**, former student at *Aula Escola Europea*, for giving me her research work, answering the doubts I had about it and searching some more information for me. Her help proved very useful at the last part of the research.

Finally I would like to thank the ***Wolfram Alpha project*** [18], an expansion of

*Mathematica*, a professional website that helped me carry out some tedious calculations such as Taylor expansions or polynomial factorization. I am grateful to say that I was able to help this project by correcting some errors I found while I did the study.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Introduction

*A mind once stretched by a new idea never regains its original dimension.* **Oliver Wendell Holmes**

It all started at the end of *2n d'ESO* (8th grade) when I first read the research paper done by Oriol Lozano: *Nombres primers, anàlisi de la complexitat* [11]. Although I could not understand most of it I truly enjoyed his study, not only because I was also fascinated by prime numbers but also because it was the symbol of what I considered a good research. Since then, it was my model: I read it some more times, understanding it better as my mathematical background increased; I even wrote a lot of prime searching algorithms at the end of *ESO* and I finally completely understood his paper last summer.

Partly by chance, I was offered a research with many things in common at the end of *4t d'ESO* (10th grade) combining mathematics and computer science. At the time I was undecided between doing a research about primes or about genetics, but the idea of that research seemed also very interesting, which is why I accepted. First of all, it was an interdisciplinary work, in the sense that it involved both computer science and mathematics; both topics of my interest. Moreover, it included a high connection between them, interlacing both disciplines to achieve a common goal.

This duality between computer science and mathematics was also reflected in the combination of both theory and practice, which characterizes my way of understanding science and, as the reader will later see, this work. In particular, starting from the abstraction of generating functions we will proceed to the more application-oriented algorithms to finish with an application in genetics.

Furthermore, mathematics provided me the possibility of starting from the very beginning. If one does research in real systems such as biology or economics, several questions will arise without never getting to the base of the system due to the number and complexity of the elements in those systems. In mathematics, instead, if something can be proved, one can always go back and understand it from the axioms. In this sense, I wanted to understand the proofs of my work to the least detail and mathematics was the only discipline that enabled me to do that. For this reason, some references to the basic results are done during the work, since I have proved it all from there, although it is not the aim of this work.

Last, but not least, I also wanted it to be an intellectual challenge and the difficulty of understanding and manipulating generating functions was a very good option. Moreover, the possibility of building my own algorithms was a great opportunity to develop an intellectual creativity.

**Questions and objectives.** The main questions and objectives of this work arise from the following observation. English, Catalan, binary system found in computers, the genetic code and even music share one thing in common: they all have an alphabet. My main objective was to analyze words from a mathematical point of view as well as from the algorithmic perspective. Given a finite and uniform alphabet ( meaning that each possible sign appears with equal probability ); although at first it may not seem so, some patterns have a greater probability of appearing than others, depending on its own symmetry. Take, for example, the pattern 11 and 10 and binary words of length 4:

$$0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111$$

Words with 11: $0011, 0110, 0111, 1011, 1100, 1101, 1110, 1111$. Thus, it appears in 8 words and 12 times in total, for example in 1111 it appears 3 times. Words with 10 are the following ones:

$$0010, 0100, 0101, 0110, 1000, 1001, 1010, 1011, 1100, 1101, 1110.$$

Thus, it appears in 11 words and 12 times in total.

It is not difficult to see that the results for 00 should be the same as 11, because sign '0' and '1' are essentially the same. Similarly, 10 and 01 should have the same results as well. Therefore the only thing that influences the probability of a word of appearing in a text only depends on the geometry/symmetry of this word, as we will later see in this introductory chapter.

In general I was interested in one main question:

**Research Question 1: Given a finite alphabet and a finite pattern in this alphabet, how can we find how many words of a given size contain this pattern?**

However, I was not only interested in this question, but also in finding if this geometry affected the way we searched for patterns in big texts. I already knew a very simple algorithm to find patterns in texts, but finding faster and more interesting algorithms became another objective in my research.

Moreover, an interdisciplinary research can be very helpful and deeper than doing it in only one part of science; as it was the case in this particular work. Both parts are build one over the other, as information in one part created insightful questions for the other part and helped solving some others in this other part.

We could generalize the objective of this second part in another question:

**Research Question 2: How can we build an efficient search algorithm using the information we have about the symmetry of the pattern we are looking for?**

As in every research, some other questions arose when trying to answer these very big questions; some of them were necessary to solve these initial questions, some of them were not, but still were worth of notice and solving.

A very big difficulty of a mathematical paper is that to do *new* research one has to have a very deep knowledge of the theme his investigating. For a high school student this implies the necessity of learning a lot before starting its research. However, **answering my own questions, proving the theorems I encountered and trying out different examples in every theme of the student** was a research in itself. In fact, all but one theorems were proved by the author entirely, which represents a huge proportion of the theorems in the work.

It was, in fact, very interesting to see how questions I asked myself and solved them were found later in other papers or books such as finding an algorithm to determine if a text contains either of two patterns or finding the average number of steps of the naive algorithm, that will later be analyzed. Another important example would be the final application, as I thought genetics could be the perfect application with its big texts and small alphabets. However, my initial idea had already been done, which at the same time made me sad but also showed that analyzing genetics was a good idea.

Apart from those questions and the multiple others that would arise I also wanted **to apply the knowledge of those two questions into a more practical application in genetics**, but I would want to remark that this became an extra part of my research and thus it does not pretend to give clear and proved results but a taste of what could be done. At the same time, **the development of an automatic solver became one of the main objectives**, even without realizing its importance beforehand.

Finally, it was also of great importance for me **to make an authentic experience out of that research**, as it will now be explained.

**Personal experience** Instead of starting at the end of the first trimester of *1r de Batxillerat* (11th grade) I started working at the end of *4t d'ESO* (10th), providing me a much longer, and needed, span to work. Learning generating functions had a lot of difficulties, and I spent months only to understand the subject, meeting regularly with Dr. Juanjo Rué, a researcher who introduced me to the topic and guided my learning.

When I was in *1r de Batxillerat* I did two other research studies for the program *Joves i Ciència*, one of them about fractals and the other one was a simulation of a galaxy. After finishing the researches for that program, I considered using one of them as a good solid base of my work for the school; but, I decided not to, for the experience of doing it. Although they were done in very different topics, the difficulties and experiences I had with both were very useful to learn how to handle this project.

The summer before *2n de Batxillerat*, I was accepted in an intense math program in *Ohio State University* for two months, learning a whole new vision of math, proof and the art of thought in general. More precisely, it also helped me greatly in some proofs that appear in Chapters 3 and 7.

I spent only 1 week in Spain before going back to Boston for a school exchange. This experience also helped me indirectly, with the *Abstract Algebra* class and specially by improving my English, which will later be treated. After going back from Boston I worked intensively at the end of the first trimester which provided me the opportunity of taking a break during winter vacation and concentrate in some sections which were more creative at the start of the second trimester.

As I mentioned, doing the document in English was an important part of it; my motivations to do so were the challenge it supposed, the learning I could get from it, and the internationality and flexibility it gave to the transcript since nowadays almost all research is done in English. Looking backwards, although I have had some difficulties with the language, it was a perfect opportunity to practise it.

For this research I had to learn another language, LATEX, a special text editor used by many professional scientists and mathematicians around the world. Although it is a quite complex and non-intuitive language compared to Word and Open Office (my first two tries), after the first two months of learning, it was a good experience. It gave a perfect editor of mathematical notation (specially important in this work), a neat presentation and a good system of citations, references and structure which I hope the reader will appreciate. Related to this, I really appreciated the possibility of working closely with a researcher in mathematics, Dr. Juanjo Rué, which provided me an insight of how mathematical research is.

On the other hand, the computer science part and specially the application in biology were done without any supervisor. In this case, the reader should take the biological application more like an overview of what could be an actual research, than a serious research itself; adding the bigger picture that I wanted to include to this study.

Finally, those last months I have discussed my research with different people, since I wanted to improve the comprehensibility of the study by learning the points that were more difficult to understand. I would like to highlight a presentation I gave in front of 4 mathematicians/engineers, with my two tutors Mrs. Laura Morera and Dr. Juanjo Rué and Lluís Vena and Txema Tamayo, this last winter and a later talk with Ms. Esther Silberstein; both being very special experiences. They were a perfect moment to see the possible doubts that the research could have in understanding. They were also a good practice for a presentation of the study and an opportunity for me to get my ideas clearer, as one always understands better things when it has to explain them.

All in all, those different things made my research more than a simple work becoming a personal experience through almost two years.

# Chapter 1

# *Basic* combinatoric tools

*A key to understanding the complex and unknown is a deep understanding of the simple and familiar.* **Burger and Starbird**

## 1.1 Description of combinatorics

Before starting our work we should define the field we are working on, combinatorics: the British Encyclopedia [20] defines it as "*Branch of mathematics concerned with the selection, arrangement, and combination of objects chosen from a finite set*". More precisely we are going to focus in fields like enumerative combinatorics (counting the structures of a given kind and size) and combinatorial designs: constructing structures that meet those criterion. Finally we will be using the strong connection there is between combinatorics and computer science in analysis of algorithms to do the second part of the study. Thus, it is very important to have a solid combinatoric base. Combinatorics is a very broad subject, from basic formulas every high school student knows to very powerful and complex tools such as generating functions. However, we cannot really understand the latter without the first, so it is very important to start with the most basic things.

In this chapter, we will start by analyzing simple formulas and providing an explanatory proof and an example for each. After this, we will consider linear recurrences by showing how to obtain a formula from them and also providing some examples.

## 1.2 Basic combinatoric operations

### 1.2.1 Permutations with repetitions

Imagine you throw a dice $n$ times, how many possibilities are there? If you throw it only 1 time there are only 6 possibilities: 1,2,...,6. If you throw it 2 times the possibilities are: $11, 12, ..., 16, 21, 22, ..., ...26, ...$ etc until 66; therefore there are 36 possibilities because we have 6 possibilities for the first number and 6 possibilities for the second. Note that repetitions are allowed since you can have two consecutive 1's when you throw a dice. Then for 3 throws you will have 216 possibilities and in general for '$n$' throws you will have $6^n$ different possibilities. Note that here, two sequences A and B are different if and only if there exists an $i$ such that $A_i \neq B_i$.

Another example could be the number of ways to make 'n' steps in a binary tree: for each step you can either go to the left or to the right, therefore you end up with $2^n$ possibilities. Finally consider decimal numbers of n digits. For each digit there are 10 possibilities thus having $10^n$ possibilities for a $n$-digit number.

In general this will mean: $A \times A \times \times \overset{n}{\times} \times A$, thus the size will be $|A|^n$.

### 1.2.2 Permutations without repetitions

Now suppose you have $n$ different people in a lane and you want to know how many ways there are of ordering them. For the first position in the lane you have $n$ options, for the second you have $n-1$ options because you cannot put a person in both the first and second place. For the third place you have $n-2$ options and so on until for the last place you will have only one option. In general a permutation without repetition will be of the form: $P(A) = A \times (A \backslash \{a_1\}) \times (A \backslash \{a_1, a_2\}) \times \overset{n}{\times} \times \times (\{a_n\})$.

Thus, in general in a permutation without repetition of n elements there are exactly the following ways to do it:

$$|P(A)| = n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1 = n!.$$

Remember that the notation $n!$ is defined as:

$$n! = n \cdot (n-1)! \text{ for } n > 0, \text{ with } 0! = 1.$$

### 1.2.3 Combinations without repetitions with order

Consider the different possibilities of the olympic medals in a race of 8 runners. It is clear that this time order matters, it is not the same to be the gold medal than to be the silver medal. Again we can use 'the box technique' to find the formula in a simple way. For the gold medal we have 8 options, for the silver medal we have 7 options because a runner cannot win two different medals in a race, and the bronze medal has 6 options, therefore the total number of options is: $8 \cdot 7 \cdot 6 = 336$. The general form of a combination without repetitions and with order is:

$$C(A) = A \times (A \backslash \{a_1\}) \times (A \backslash \{a_1, a_2\}) \times \overset{n}{\times} \times \times (A \backslash \{a_1, a_2, \dots, a_m\})$$

In general for $n$ runners and $m$ medals the number of ways is:

$$|C(A)| = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-m+1) = \frac{n!}{(n-m)!}.$$

Although this $\frac{n!}{m!}$ may not seem very clear we can see it in a mathematical way as the following. Consider $n!$ and $m!$, then: $n! = n \cdot (n-1) \cdot \dots \cdot (n-m+1) \cdot \cdot (n-m) \cdot (n-m-1) \cdot \dots \cdot 2 \cdot 1$ and $(n-m)! = (n-m) \cdot (n-m-1) \cdot \dots \cdot 2 \cdot 1$.

Therefore $\frac{n!}{(n-m)!}$ will be the terms that appear in $n!$ but do not appear in $m!$ giving us exactly what we want. We can also use the permutations without repetitions to explain the formula: Consider the $n!$ ways of ordering the runners. However, for our purposes we will not care about the order of the last $(n-m)$ runners and thus each element will be repeated $(n-m)!$ times; thus, we have to divide by $(n-m)!$.

### 1.2.4 Combinations without repetition

A captain has to choose $k$ soldiers from the $n$ soldiers available to make a guard for the night. Obviously we cannot repeat any soldier and the order does not matter. We can use the box system again to show the formula in this case: n options for the $1^{st}$ soldier, $n-1$ for the $2^{nd}$... $n-k+1$ for the $k^{th}$ which we determined to be $\frac{n!}{(n-m)!}$ However now we do not want to repeat equal ways of doing the guard for the night, for example ABC=BCA=CBA. In fact we counted $m!$ every solution (we can determine it with the permutations without repetition we did previously). Hence, the solution is:

$$\frac{n!}{(n-m)!m!} = \binom{n}{m}.$$

We can generalize it to $k$ groups of sizes $a_1, a_2, \dots, a_k$, and thus $\sum_{i=1}^{k} a_i = n$. Given this, then the number of ways to put $n$ elements into those $k$ groups is:

$$\frac{n!}{a_1! a_2! a_3! \cdots a_k!}.$$

Those type of numbers are called multinomials. Note that, although it may not seem so, they must all be integers because they are counting things.

### 1.2.5 Combinations with repetition

Imagine you have to choose the flavor of the balls of your giant ice cream. There are $k$ flavors and $n$ balls in your ice cream. It does not matter much the order in which they put them in your ice cream as long as you have them. This example is probably conceptually the hardest and we cannot use the box technique since the number of options in each box is dependent on what it has been put in the previous boxes. However we can use another approach which we will call 'points and sticks' algorithm[7]:

1. Draw $n + k - 1$ points.
2. Convert $k - 1$ points in sticks.
3. You will have $n$ points separated in k groups by $k - 1$ sticks.

For example consider 8 balls of 3 flavors: banana, chocolate and vanilla:

1. Draw 12 points
2. Convert 2 points for sticks
3. the number of points before the first stick is the number of banana balls, between the 2 sticks is the number of chocolate balls and after the $2^{nd}$ stick is the number of vanilla balls.

One can now identify the situation: we have to choose the k points we want to convert into sticks from the $n + k - 1$ points without order, the example explained just above. The formula is then:

$$\binom{n+k-1}{k}.$$

### 1.2.6 Some theorems following from the formulas

Using the definitions we showed we can prove some very useful and interesting theorems.

**Theorem 1.2.1**

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n. \tag{1.1}$$

**Proof:** Every $\binom{n}{k}$ counts the number of ways to take k elements out of the n elements. If we do it for every k between 0 and k we are analyzing any subset of the n elements since every subset will consist in taking k elements for $0 \leqslant k \leqslant n$. At the same time $2^n$ also counts the number of possible subsets of a set of $n$ elements since for each element we have 2 possibilities, putting it in the subset or not. Therefore, since they count the same thing, they must be equal. □

**Theorem 1.2.2** ***Binomial theorem:***

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}.$$

**Proof:** First note that $(a+b)^n$ is equivalent to $(a+b)(a+b) \overset{n}{\cdots} (a+b)$. We will obtain each coefficient by multiplying $a$ or $b$ from the first parenthesis, $a$ or $b$ for the second one, etc. Thus, the coefficient of a plus the coefficient of b always have to add up to $n$. Moreover, the number of coefficient $a^i b^{n-i}$ should be equal to the number of ways to choose $i$ elements from $n$ elements since you are simply choosing $i$ parenthesis from the $n$ possible.

□

This provides another very simple proof of Theorem 1.2.1:

**Proof:** If one takes $(1+1)^n$ by the binomial theorem we have:

$$(1+1)^n = \sum_{i=0}^{n} \binom{n}{i} 1^i 1^{n-i}.$$

We can now eliminate the 1, getting:

$$\sum_{i=0}^{n} \binom{n}{i}$$

and knowing that $(1+1)^n = 2^n$ we get:

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n.$$

□

Again, using binomials we can prove that

$$\sum_{k=0}^{n} \binom{2n}{2k} = \sum_{k=0}^{n-1} \binom{2n}{2k+1}$$

and

$$\sum_{k=0}^{n} \binom{2n}{2k} = 2^{2n-1}$$

as we find at the end section 9.2. Those are maybe more a little bit more interesting results, but not so well known; however they will also later be used for the study of algorithm efficiency.

## 1.3 Recurrences

### 1.3.1 Introduction to recurrences

However, there are a lot of problems that are not so standard and it is very difficult, if not impossible, to find a formula based on those general formulas showed previously. A very useful tool are recurrences. Recurrences work very well in cases in which knowing small cases can help solving bigger cases; for example, when the value of a formula for $n-1$ and $n-2$ determine the value for $n$. A recurrence is then something based in itself, $n!$ is a good example of that. It is very important to note, however, that they also must have a base case in order for us to get a solution.

Moreover, a lot of times we may know what the recurrence is, but we will not be able to find it and that is where generating functions will come along.

Let us start with a very simple problem: how many different words of length $n$ over the alphabet $A = \{0, 1\}$?

Let this number be $B_n$. We can easily see that if we have a word of length $n-1$ we could form two different words by adding either 0 or 1 at the end of the word. Moreover, these two words will be different from others as they will have a different beginning. This brings the first recurrence relation:

$$B_n = 2 \cdot B_{n-1}.$$

However, we seek an explicit formula for each $n$. To solve recurrences we find the characteristic polynomial: first we pass all the terms to one side of the equation:

$$B_n - 2 \cdot B_{n-1} = 0.$$

We then change each term $B_n$ by $x^n$, giving:

$$x^n - 2 \cdot x^{n-1} = 0$$

We divide by the lowest term and solve for x, obtaining that $x = 2$.We then know that the formula will be some power of 2 times a constant. As it is a recurrence of degree one (meaning that we can form a term only knowing the previous one) we will only need one constant. It has to match for every $n$, so any particular case could tell us the constant. Take for example $B_1$:

$$B_1 = 2^1 \cdot C = 2.$$

We then know $C = 1$ so in general:

$$B_n = 2^n \cdot 1 = 2^n.$$

However, there is a much more intuitive solution to find that $B_n = 2^n$:

$B_n = 2{\cdot}B_{n-1} = 4B_{n-2} = 2^3 B_{n-3} = \dots$ From this, we could continue expanding the expression getting always something of the form $2^i B_{n-i}$. However, we know that $B_0 = 1$ because there is only one empty word. Thus $B_n = 2^n B_0 = 2^n$.

For a deeper comprehension it is more useful to set more complex examples.
Consider again words only with 0s and 1s, but add the restriction that no word can have the substring 01 in it. Let us call $U_n$ the number of sequences that satisfies these properties and end with 1 and $Z_n$ the number of sequences that ends with 0. We can easily define two recurrences:
$U_n = U_{n-1}$ we require that the previous letter is a 1, otherwise it would be 01.
$Z_n = Z_{n-1} + U_{n-1}$ with a 0 at the end we do not have such problem.

Then the total number of words of length $n$ is:

$$T_n = U_n + Z_n$$

Now we can look at $U_n = U_{n-1}$ and it is easy to see that $U_n = U_1 = 1$ for every n. Then: $Z_n = Z_{n-1} + 1$, we can easily see that $Z_n = n$ without doing the characteristic polynomial, so $T_n = n + 1$. In fact we can see that the number of words is very restricted, once we put a 0 we are obliged to end the word in a sequence of 0s or otherwise we would create the pattern 01, as an example those are the 6 words of length 5:

$$11111, 11110, 11100, 11000, 10000, 00000$$

Now consider the restriction of the pattern 11: Consider again two recurrences $O_n$ and $U_n$ for endings in 0 and 1 respectively. $Z_n = Z_{n-1} + U_{n-1}$ since there are no restrictions in substrings ending in 0.
$U_n = Z_{n-1}$ since we have the restriction of the patter 11.
Now we observe:

$$Z_n = Z_{n-1} + U_{n-1} = Z_{n-1} + Z_{n-2}$$

$$T_n = U_n + Z_n$$

Substituting we get: $T_n = Z_n + Z_{n-1}$ and we can see $T_n = Z_{n+1}$.

We recognize the Fibonacci recurrence, a very classical one:

$$F_n = F_{n-1} + F_{n-2}$$

We can do the same obtaining the following characteristic polynomial:

$$x^2 - x - 1 = 0$$

with solutions $x = \phi$ and $x = \overline{\phi}$ where $\phi = \frac{1+\sqrt{5}}{2}$ and $\overline{\phi}$ is its conjugate: $\frac{1-\sqrt{5}}{2}$. We then know that the formula for the $n^{th}$ Fibonacci number should be of the form:

$$c_1 \cdot \phi^n + c_2 \cdot \overline{\phi}^n$$

Now we need two equations to solve for two variables, that is why the well known Fibonacci sequence has its two first terms predefined. We define $F_0 = 0$ and $F_1 = 1$ having the following equations:

$$c_1 + c_2 = 0, \text{ and } c_1 \cdot \phi + c_2 \cdot \overline{\phi} = 1$$

which give the solution $c_1 = +\frac{1}{\sqrt{5}}$ and $c_2 = -\frac{1}{\sqrt{5}}$ and finally giving the formula for the $n^{th}$ Fibonacci number:

$$F_n = \frac{\phi^n - \overline{\phi}^n}{\sqrt{5}}.$$

If we observe further, this formula also gives us more inside about the sequence: Knowing that $|\overline{\phi}| \approx |-0.618| < 1$ we can determine:

$$\lim_{n\to\infty} \overline{\phi}^n = 0.$$

which implies

$$\lim_{n\to\infty} F_n \approx \frac{\phi^n}{\sqrt{5}}.$$

And so,

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} \approx \phi^n.$$

That is why we can approximate the golden proportion by dividing two consecutive Fibonacci numbers. Due to its properties, Fibonacci numbers appear a lot in art (such as in Ancient Greece) and nature.

### 1.3.2 The characteristic polynomial

Now we want to make a general solution for types of recurrences similar to the ones we have seen. To do that we will test solutions of the form $A^n$ and see if they can work. *Note: for this part the author inspired its proofs in [7], but adding a personal touch or change to each proof when not completely changing it.* Using the trick of the characteristic polynomial it is very important to prove it. We start with a simple proof applicable to one of the previous cases.

**Theorem 1.3.1** *Given a recurrence of the form $f_n - c \cdot f_{n-1} = 0$ for n>1 the explicit formula is of the form: $f_n = f_0 \cdot c^{n-1}$.*

**Proof:** We will approach it by induction.First we check the case for $n = 1$:

$$f_1 = A \cdot c^{1-1} = a \cdot c^0 = A$$

which is true Now we want to prove that if $f(n)$ satisfies the formula then so does $f(n+1)$.
$f_{n+1} = c \cdot f_n = c(A \cdot c^{n-1}) = c(A \cdot cn - 1) = A \cdot c^n$.
We have then showed that it works for every positive integer. □

We now want to proof another theorem applied more than once before:

**Theorem 1.3.2** *We can get a formula for a recurrence of the form* $a_n - c_1 \cdot a_{n-1} - c_2 \cdot a_{n-2} = 0$

**Proof:** Again we suppose that the solution will be the solution to some sort of polynomial:

$$f_n = x^2 - c_1 x - c_2 = 0$$

Two cases appear: $f_n = (x - \alpha)^2$ or two roots: $f_n = (x - \alpha)(x - \beta)$
**Case 1**: $f_n = (x - \alpha)^2$ We now try to find some simple polynomials that satisfy the recurrence $a_n = A \cdot \alpha^n$.
$A \cdot \alpha^n - A \cdot \alpha^{(n-1)} - A \cdot \alpha^{(n-2)} = A\alpha^{(n-2)}(\alpha^2 - c_1\alpha - c_2)$ and $\alpha^2 - c_1\alpha - c_2$ obviously satisfies the recurrence so it must equal 0 which implies $A\alpha^{(n-2)}(\alpha^2 - c_1\alpha - c_2) = 0$, satisfying the recurrence. We can find another polynomial that satisfies the recurrence, $Bn\alpha^n$:
$Bn\alpha^n - c_1B(n-1)\alpha^{n-1} - c_2B(n-2)\alpha^{n-2} = Bn\alpha^{n-2}(\alpha^2 - c_1\alpha - c_2) + B\alpha^{n-2}(c_1\alpha + 2c_2)$ again we use the fact that $\alpha^2 - c_1\alpha - c_2$ satisfies the recurrence to know that the first term is 0. Now we analyze: $B\alpha^{(n-2)}(c_1\alpha + 2c_2)$.
Since $\alpha^{(n-2)} \neq 0$ because $\alpha \neq 0$ then we need: $c_1\alpha + 2c_2 \doteq 0$
Remember that case 1:$x - c_1x - c_2 = (x - \alpha)^2 = x^2 - 2x\alpha + \alpha^2$. In other words, $c_1 = 2\alpha$ and $c_2 = \alpha^2$.
$c_1\alpha + 2c_2 = (2\alpha)\alpha + 2(\alpha^2) = 2\alpha^2 - 2\alpha^2 = 0$ Thus $Bn\alpha^n$ also satisfies the recurrence.

We now use the theorem that if P and Q satisfy a recurrence so does P+Q to know that $f_n = a'_n + a''_n = (A + Bn)\alpha^n$ satisfies the recurrence.

Based in the last two values, it is clear we only need the first two to define the recurrence, let us put them in our polynomial form.

$$f_1 = (A + B)\alpha$$

$$f_2 = (A + 2B)\alpha^2$$

Doing a little bit of tedious algebra we get:

$$A = \frac{2\alpha f_1 - f_2}{\alpha^2}$$

$$and$$

$$B = \frac{f_2 - \alpha f_1}{\alpha^2}$$

Note that since $\alpha \neq 0$ we have determined a formula for the recurrence of the form: $(A + Bn)\alpha^n$

Before going to the next case a question, that the reader may have already noticed, arises: why cannot we have a formula of the form $A\alpha^n$ or $Bn\alpha^n$ if those polynomials also satisfy the recurrence?

$$f_1 = A\alpha$$

$$f_2 = A\alpha^2$$

As we can see, this would imply $f_2 = f_1\alpha$ which is not always the case.

$$f_1 = B\alpha$$

$$f_2 = 2B\alpha^2$$

This would imply $f_2 = 2f_1\alpha$ which is not always the case either.

**Case 2**: $f_n = (x - \alpha)(x - \beta)$ For the previous case we proved that $A\alpha^n$ satisfied the recurrence without using any particular fact of the roots of the polynomial. $B\beta^n$ must also satisfy the recurrence because it is only a change of variables and then so does their sum: $A\alpha^n + B\beta^n$.

Let us try to get A and B from $f_1, f_2, \alpha$ and $\beta$ as we did in case 1:

$$f_1 = A\alpha + B\beta, f_2 = A\alpha^2 + B\beta^2$$

Again, we can solve for A and B:

$$A = \frac{f_1\beta - f_2}{\alpha(\beta - \alpha)}, B = \frac{f_1\alpha - f_2}{\beta(\alpha - \beta)}$$

We could show that $A\alpha^n$ cannot be a formula for all the recurrences of that form as we did in case 1.

Here we could add an interesting point, as in a lot of mathematical calculations it is sometimes much easier starting with 0 instead of 1.

Let $g_n = f_n + 1$, then $g_0 = f_1$ and $g_1 = f_2$.

$$g_0 = A + B$$

$$g_1 = A\alpha + B\beta$$

which looks much easier to solve but gives a similar result. $\square$

### 1.3.3 Failure of recurrences

Now we have shown a very powerful tool to solve combinatoric problems. We started with some basic formulas and we continued with recurrences for more complicated cases. However, there are some problems that we cannot solve with recurrences. Those need a similar structure for every step, in other words: the step from n to $n + 1$ is the same as the step from $n + 1$ to $n + 2$. In the 11 problem we solved with Fibonacci, the rule is always the same and does not change.

Take now for example a much more complicated case: triangulations. For us, triangulating a polygon means dividing the polygon all into triangles, without interior points, interior polygons different than a triangle or any crossing between lines (which implies that the polygon must be convex).For example the triangulations in Figure 1.3.3 are invalid, while figures in the Figure 1.2 are valid.

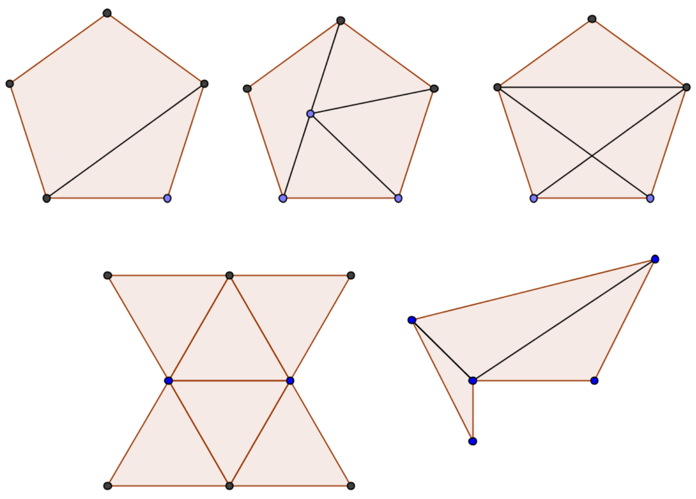


Figure 1.1: Wrong triangulations

As you can see in Figure 1.3 to triangulate $F_n$ we first fix a first edge and then another point. For the point we have in total $n - 2$ options. As the figure illustrates, we can fix a point and get the triangulations we have fixing the triangle. Let $m$ be the position of the point counting from counter-clockwise direction from the left point of our starting edge, then there are $m - 1$ points

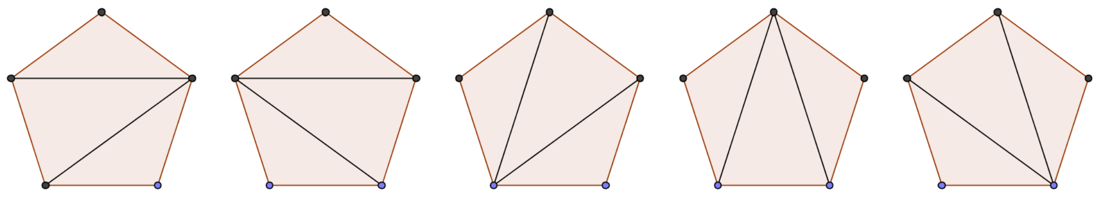


Figure 1.2: Valid Triangulations

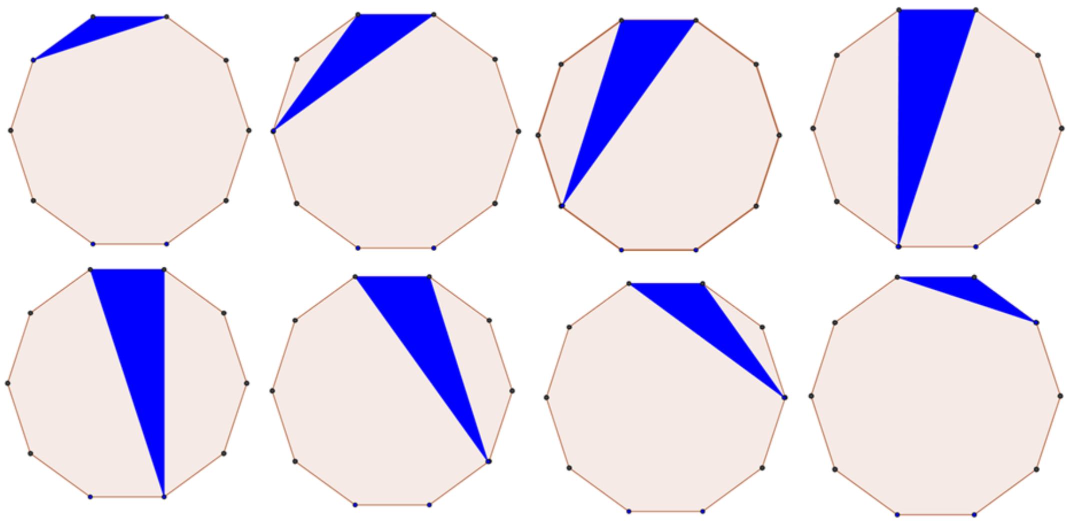


Figure 1.3: Calculating triangulations

between them creating a polygon of size $m+1$. From the clockwise directions they are $n-m-1$ points, which create a polygon of size $n-m+1$. Therefore we have for each point $T_{m+1} \cdot T_{n-m+1}$ triangulations giving the following recurrence:

$$T_n = \sum_{m=1}^{n-2} T_{m+1} \cdot T_{n-m}$$

However, for non-linear recurrences like this one, no closed formula can be found with the methods we explained. A more general and complex structure is needed, generating functions.

# Chapter 2

# Introduction to generating functions

*By relieving the brain of all unnecessary work; a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race.* **Alfred North Whitehead**

Once we showed the failure of *basic* combinatorial tools to solve specially complex combinatorial problems it is necessary to introduce a new mathematical object. Although this object will be used mainly for combinatorial purposes it has to be previously well defined and with certain properties proved. Those definitions and properties will be shown and proved in the first part of the chapter while in a second part of the chapter we will prove some famous theorems of calculus or trigonometry using generating functions to both show their utility and how they work.

*Note: All the theorems shown in the chapter are entirely done by the author. Moreover, more than once we use properties of the integers which, although obvious, have also been proved by the author outside this work.*

## 2.1 Definitions

First of all we define what a ring is.

**Definition 2.1.1** *A* ring *is a set $R$ equipped with two binary operations $+\colon R \times R \to R$ and $\cdot\colon$*

*$R \times R \to R$ we call addition and multiplication. This set and two operations have to satisfy the ring axioms:*

*(R,+) is required to be an abelian group under addition:*

1. *Closure under addition*
2. *Associativity of addition*
3. *Existence of additive identity*
4. *Existence of additive inverse*
5. *Commutativity of addition*

*(R,·) is required to be a monoid under under multiplication:*

1. *Closure under multiplication*
2. *Associativity of multiplication*
3. *Existence of multiplicative identity*

*The distributive laws:*

1. *For all a,b and c in R: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.*
2. *For all a,b and c in R: $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$.*

A **generating function** $F(x)$ is a formal power series whose coefficients give the sequence $a_0, a_1, \ldots$. As it is added in [3] a generating function can often be thought of as a (possible infinite) polynomial whose coefficients count structures that are encoded by the exponents of the variables.

Here formal means that we do not substitute the indeterminate for any values, the important thing are the coefficients.

For example $F(x) = \frac{1}{1-2x} = 1 + 2x + 4x^2 + \cdots + 2^n x^n + \ldots$ would count the binary words of size $n$ because the coefficient of degree $n$ is $2^n$. Another example of generating function would be $G(x) = \frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + 4x^3 + 5x^4 + \cdots + nx^n + \ldots$ which could count the numbers between 1 and $n$, both included.

To work with generating functions we first have to define the normal operations we find in other mathematical objects such as integers or polynomials. We define the sum of two generating functions similar to the sum of two polynomials:

**Definition 2.1.2** *Sum of two generating functions:*

$$A(x) + B(x) = \sum_{n=0}^{\infty}(a_n + b_n)x^n$$

Similarly we can define the substraction.

**Definition 2.1.3** *Substraction of two generating functions:*

$$A(x) - B(x) = \sum_{n=0}^{\infty}(a_n - b_n)x^n$$

We can also define one of the basic operations: multiplication.

**Definition 2.1.4** *Product of two generating functions:*

$$A(x) \cdot B(x) = \sum_{m=0}^{\infty} a_m x^m \sum_{k=0}^{\infty} b_k x^k = \sum_{n=0}^{\infty}\sum_{k=0}^{n} a_k b_{(n-k)} x^n.$$

This means multiplying all possible coefficients that will make the power $n$. For example for coefficient 0 only $a_0b_0$ is possible; for coefficient 1 $a_0b_1$ and $a_1b_0$ and for coefficient $n = 3$: $a_3b_0$, $a_2b1$, $a_1b2$, $a_0b_3$, and so on.

Once we have defined those basic operations in this new mathematical object it is very useful to prove some things about them.

## 2.2 Basic properties of sum and product of generating functions

### 2.2.1 Basic operations with generating functions

We now want to prove some useful properties of sums of generating functions. In particular it is very useful that like other mathematical objects as integers or polynomials, generating functions form also a ring with addition and multiplication.

**Theorem 2.2.1** ***Generating functions are closed under multiplication and addition:***

**Proof:** If we take the definitions of addition and multiplication of generating functions we can see that their output just defines another infinite series of powers, stating what each coefficient will be. Therefore it is a generating function and they are closed under both addition and multiplication. $\square$

**Theorem 2.2.2** ***Generating functions are commutative with the sum:*** $A(x) + B(x) = B(x) + A(x)$***.***

**Proof:** $A(x) + B(x) = \sum_{n=0}^{\infty}(a_n + b_n)x^n$. Now we use the commutativity of integers

$$\sum(a_n + b_n)x^n = \sum_{n=0}^{\infty}(b_n + a_n)x^n = B(x) + A(x).$$

$\square$

**Theorem 2.2.3** ***Generating functions are associative with respect to the sum:*** $(A(x) + B(x)) + C(x) = A(x) + (B(x) + C(x))$.

**Proof:**

$$(A(x) + B(x)) + C(x) = \sum_{n=0}^{\infty}((a_n + b_n) + c_n)x^n.$$

Now we apply associativity of integers

$$\sum_{n=0}^{\infty}(a_n + (b_n + c_n))x^n = A(x) + (B(x) + C(x)).$$

$\square$

We could also prove that multiplication of generating functions is both associative and commutative in a very similar way.

Finally, apart from identities and inverses, which we will prove in the next subsection, we have to prove the distributive law.

**Theorem 2.2.4** ***Given Generating Functions*** $A(x), B(x), C(x)$***:*** $A(x) \cdot (B(x) + C(x)) = A(x) \cdot B(x) + A(x) \cdot C(x) = (B(x) + C(x)) \cdot A(x)$

**Proof:**

$$A(x) \cdot (B(x) + C(x))$$

$$\sum_{n=0}^{\infty}(a_n)x^n \cdot \left(\sum_{n=0}^{\infty}(b_n) + \sum_{n=0}^{\infty}(c_n)\right).$$

Now we apply the summation we defined before:

$$\sum_{n=0}^{\infty}(a_n) \cdot \sum_{n=0}^{\infty}(b_n + c_n).$$

Now we apply the multiplication we defined before:

$$\sum_{n=0}^{\infty}\sum_{k=0}^{n} a_k(b_{(n-k)} + c_{(n-k)}).$$

We apply distributive property of integers:

$$\sum_{n=0}^{\infty}\sum_{k=0}^{n} a_k b_{n-k} + a_k c_{n-k}$$

$$\sum_{n=0}^{\infty}\sum_{k=0}^{n} a_k b_{n-k} + \sum_{n=0}^{\infty}\sum_{k=0}^{n} a_k c_{n-k}$$

$$A(x) \cdot B(x) + A(x) \cdot C(x).$$

Finally, since we have proven commutativity of generating functions with multiplication we know: $(B + C) \cdot A = A \cdot (B + C) = A \cdot B + A \cdot C$. □

### 2.2.2 Identities and inverses

A very important part of a lot of mathematical fields, as group theory or number theory, are identities/units. For example we know that the identity of addition is 0 and the identity of multiplication of integers is 1. For instance, we will use the fact that they are both identities in addition and multiplication respectively to find the identity generating function. First we want to find I(x) such that:

$$A(x) + I(x) = A(x) = I(x) + A(x).$$

**Theorem 2.2.5** ***There exists an identity for the addition of generating functions.***

**Proof:**

$$\sum_{n=0}^{\infty}(a_n + i_n)x^n = \sum_{n=0}^{\infty}(a_n)x^n.$$

So we need $a_n + i_n = a_n - i_n = a_n$ for every $n$. Since they are all integers we can solve and get $i_n = 0$ for every $n$. Which is equivalent to say:

$$I(x) = \sum_{n=0}^{\infty}(0)x^n = 0.$$

□

**Theorem 2.2.6** ***For every generating function*** $v(x)$ ***there exists another generating function*** $v'(x)$ ***such that*** $v(x) + v'(x) = 0$

**Proof:** If we take the negative ( the common name for additive inverse ) of each coefficient of $v$ we get:

$$\sum_{n=0}^{\infty}(v_n + v'_n)x^n = \sum_{n=0}^{\infty}(v_n - v_n)x^n = \sum_{n=0}^{\infty} 0x^n = 0.$$

□

For the product we need an $I(x)$ such that:

$$A(x)I(x) = I(x)A(x) = A(x).$$

**Theorem 2.2.7** ***There exists an identity for the multiplication of generating functions.***

**Proof:**

$$\begin{aligned} & I(x)A(x) \\ &= \sum_{m=0}^{\infty}\sum_{k=0}^{m} i_k a_{(m-k)} x^m \\ &= \sum_{m=0}^{\infty}(i_0 a_m + \sum_{k=1}^{m} i_k a_{m-k}). \end{aligned}$$

We need $i_0$ to be 1 and $\sum_{k=1}^{m} i_k a_{(m-k)}$ to be 0 and therefore all $i_r$ for $r > 0$ should be 0. With this we get that the identity for the multiplication is:

$$I(x) = 1 + \sum_{m=1}^{\infty} 0x^m = 1.$$

□

Now, although it is not a condition for being a ring, we can look for the multiplicative inverse, if it exists, of a generic generating function $A(x)$.

Find B(x) such that $A(x)B(x) = 1$.

$$\sum_{n=0}^{\infty}\sum_{k=0}^{n} a_k b_{n-k} x^n$$

We start with the term of degree 0, since it can come only from multiplying degrees that add to 0: $a_0 \cdot b_0 = 1$.

$$b_0 = \frac{1}{a_0},$$

which gets us a condition for a generating function to have an inverse: $a_0 \neq 0$.

We continue with the term of degree 1: $a_0 b_1 + a_1 b_0 = 0$. We substitute $b_0$: $a_0 b_1 + \frac{a_1}{a_0} = 0$. Now we isolate $b_1$:

$$b_1 = \frac{-a_1}{(a_0)^2}$$

Again we have the condition $a_0 \neq 0$. We do the term of degree 2: $a_0 b_2 + a_1 b_1 + a_2 b_0 = 0$.

We substitute $b_0$ and $b_1$: $a_0 b_2 + \frac{-(a_1)^2}{(a_0)^2} + \frac{a_2}{a_0}$ and now we isolate $b_2$:

$$b_2 = \frac{-a_2}{a_0^2} - \frac{a_1}{a_0^3}.$$

Similarly we could continue to get the following coefficients as we prove formally.

**Theorem 2.2.8** ***Every generating function A has a defined inverse provided*** $a_0 \neq 0$

**Proof:** We prove it by induction proving we can get the value of each coefficient. Base case: $b_0 = \frac{1}{a_0}$ which is well defined when $a_0 \neq 0$.
Now let us assume we know every coefficient of $B$ up to $n-1$, let us prove we can get the $n+1$ coefficient.
$\sum_{k=0}^{n} a_k b_{n-k} = 0$ for n>0 ( definition of the multiplicative identity ).
$a_0 b_n + \sum_{k=1}^{n} a_k b_{n-k} = 0$
$b_n = \frac{-\sum_{k=1}^{n} a_k b_{n-k}}{a_0}$.
Now, provided that $a_0 \neq 0$, since we know every coefficient of $a$ and every coefficient of $b$ between 0 and $n$ we can calculate $b_n$.
Therefore, we can calculate every coefficient of B and the inverse of every generating function with $a_0 \neq 0$ is well defined. □

### 2.2.3 Derivatives and integrations

Two other important operations with generating functions are the derivative and integration; generating functions are roughly speaking as an infinite polynomial so the derivative and the integration are very simple and easily generalizable.

**Definition 2.2.9** *Derivative of a generating function $A(x)$:*
$\frac{d}{dx}\sum_{n=0}^{\infty} a_n x^n = \sum_{n=0}^{\infty} n a_n \cdot x^{n-1}$

We use the notation $\frac{d^n}{dx^n}$ for the $n^{th}$ derivative in respect to x.

**Definition 2.2.10** $\int \sum_{n=0}^{\infty} a_n x^n dx = C + \sum_{n=0}^{\infty} \frac{a}{n+1} x^{n+1}$ *where C is an arbitrary constant. Note that the integration of a generating function, as with a normal function, is defined except for the constant term.*

With derivatives we must prove that some important rules we apply in calculus are valid with generating functions too.

**Theorem 2.2.11** ***Leibnitz rule for differentiation:*** $\frac{d}{dx}(A(x)B(x)) = A'(x)B(x) + A(x)B'(x)$**.**

**Proof:** First we evaluate the left side of the equation.

$$A(x)B(x) = \sum_{m=0}^{\infty}\sum_{k=0}^{m}(a_k \cdot_{m-k} x^m)$$

$$(A(x)B(x))' = \sum_{m=0}^{\infty}\sum_{k=0}^{m} m(a_m \cdot b_{m-k})x^{m-1}.$$

Now we analyze the right side. We first have that:

$$A'(x)B(x) = \sum_{m=0}^{\infty}\sum_{k=0}^{m}(k \cdot a_k \cdot b_{m-k})x^{m-1}$$

$$A(x)B'(x) = \sum_{m=0}^{\infty}\sum_{k=0}^{m}(a_k \cdot (m-k) \cdot b_{m-k})x^{m-1}$$

Therefore,

$$A'(x)B(x) + A(x)B'(x) = \sum_{m=0}^{\infty}\sum_{k=0}^{m}(k \cdot a_k \cdot b_{m-k})x^{m-1} + \sum_{m=0}^{\infty}\sum_{k=0}^{m}(a_k \cdot (m-k) \cdot b_{m-k})x^{m-1} =$$

$$\sum_{m=0}^{\infty}\sum_{k=0}^{m}(k \cdot a_k \cdot b_{m-k}) + ((m-k) \cdot a_k \cdot b_{m-k})x^{m-1} =$$

$$\sum_{m=0}^{\infty}\sum_{k=0}^{m}(a_k \cdot b_{m-k}) \cdot (k + (m-k))x^{m-1} =$$

$$\sum_{m=0}^{\infty}\sum_{k=0}^{m} m \cdot (a_k \cdot b_{m-k})x^{m-1}.$$

which if we recall it is exactly the same as the result of the left part. □

**Theorem 2.2.12** $\frac{d^n}{dx^n}A(x)$ ***if and only if*** $A(x)$ ***is a polynomial of degree*** $\leqslant n$

**Proof:** Let us prove the left sense first:
Let $A(x)$ be $c_0 + c_1x + c_2x^2 + c_3x^3 + ...c_nx^n$, then

$$\frac{d^n}{dx^n}A(x) = \sum_{d=0}^{n} c_d \prod_{i=0}^{n}(d-i)x^{(d-n)}.$$

Now, since $n \geqslant d$ for all $d$'s, there exists an $i$ for every $d$ such that $(d - i) = 0$. This implies that the whole product must be zero and thus the summation of the products must be also 0.
Now let us prove the right sense:

$$\int \frac{dx^n}{d}0 = \sum_{d=0}^{n} \frac{c_d x^d}{d!},$$

which as it shown, it has only degree at most $n$ (depending on what $c_i \neq 0$). □

## 2.3 Compositions

Another binary operation one can do with generating functions is, as with normal functions, compose them. This means that the result of the first one is the input for the next one:
$A(B(x))$ would mean: take x and do $B(x)$ and the result plug it into $A(x)$.
There are two specially important questions about compositions and generating functions that have to be solved.
1. When can we calculate the $n^{th}$ coefficient of a composition, for a fixed $n$, in a finite amount of calculations?
2. For what $A$ there exists a $B$ such that $A(B(x)) = B(A(x)) = I(x) = x$?

Note that the identity function for the composition is simple $I(x) = x$ since it returns its input:

$$I(A(x)) = A(I(x)) = A(x).$$

Let us start solving the first question:

$$C(x) = A(B(x)) = \sum_{n=0}^{\infty} a_n \left( \sum_{m=0}^{\infty} b_m x^m \right)^n$$

We want that given the coefficients of $A(x)$ and $B(x)$ to get all the coefficients of $C(x)$. Starting with the constant coefficient:

$$c_0 = \sum_{n=0}^{\infty} a_n (b_0).$$

Assuming A has an infinite amount of coefficients this would imply $b_0 = 0$ if we want a finite number of calculations. Since the converse, $B(A(x))$, has to satisfy the same properties $a_0 = 0$ also. Let us continue with the degree one coefficient.

$$c_1 = \sum_{n=1}^{\infty} a_n b_0^{n-1} b_1.$$

$b_0 = 0 \Rightarrow (b_0)^n = 0$ for all $n > 0$, thus we have $c_1 = a_1 b_1$ which can be always found. Let us continue with the degree two coefficient.

$$c_2 = \sum_{n=1}^{\infty} a_n b_0^{n-1} b_2 + \sum_{n=1}^{\infty} a_n b_0^{n-2} b_1^2.$$

Again we use the fact that $b_0^n = 0$ for all $n > 0$ to simplify the expression: $c_2 = a_1 b_2 + a_2 b_1^2$ which can always be found in a finite number of operations.

We can deduce that we can get $c_n$ for all $n$ in a finite number of operations given $a_0 = b_0 = 0$. Without constant coefficients the only possibilities of getting the exponent $n$ is by multiplying terms which degrees add up to $n$ and since those degrees are positive the list has to be finite.

Thus, the only condition for getting $C(x) = A(B(x)) = B(A(x))$ is having $a_0 = b_0 = 0$.

Now let us solve the second one: for what $A(x)$ there exists a $B(x)$ such that $A(B(x)) = B(A(x)) = I(x) = x$?

We can start with the condition of the above question, since we have to calculate the composition in a finite amount of calculations $\Rightarrow a_0 = b_0 = 0$. Now let us try to get the next coefficients of B, starting with the linear coefficient:

$$\sum_{n=0}^{\infty} a_n b_0^{n-1} b_1 = 1,$$

we simplify given $b_0 = 0$ obtaining:

$$a_1 b_1 = 1$$

$$b_1 = \frac{1}{b_1}$$

And thus,$a_1 \neq 0$. From now on, the coefficients have to be all of them 0. We continue with the coefficient of degree 2:

$$\sum_{n=0}^{\infty} n \cdot a_n b_0^{n-1} b_2 = 0 + \sum_{n=0}^{\infty} \binom{n}{2} a_n b_0^{n-2} b_1^2 = 0$$

We use $b_0 = 0$: $a_1 b_2 + a_2 b_1^2 = 0$. We can always get it since we have $b_1$ and all the coefficient of a. Similarly since we know that $b_0 = 0 \Rightarrow b_n$ only depends on $b_j$ for $j < n$ and $a_k$, we can get all the next coefficients of B.

Thus, the only conditions for a generating function to have an inverse is $a_0 = 0$ and $a_1 \neq 0$.

## 2.4 Simple forms

As it has been described generating functions are infinite polynomials, which are very hard to manipulate. However they usually have a much simpler expression, normally as a fraction of polynomials. Those expressions would be much easier to manipulate and therefore have a great importance. Let us show some of them.

### 2.4.1 Powers of $\frac{1}{1-x}$

We want to see the following equality between generating functions:

**Theorem 2.4.1**

$$1 + x + x^2 + x^3 + \ldots = \sum_{n=0}^{\infty} x^n = \frac{1}{(1-x)}.$$

**Proof:** In other words, we want to prove that (1-x) is the multiplicative inverse of $\sum_{n=0}^{\infty} x^n$.

$$(1 + x + x^2 + x^3 + \ldots) \cdot (1 - x) =$$

$$(1 + x + x^2 + x^3 + \ldots) - x(1 + x + x^2 + x^3 + \ldots) =$$

$$1 + x + x^2 + \ldots - x - x^2 - x^3 - \ldots = 1.$$

And consequently the first relation holds. □

Before moving on, we find an application of the multiplication that may prove useful later on.

Suppose you have a generating function $A(x) = \sum_{n=0}^{\infty} a_n x^n$ and you want a new generating function $B(x)$ with $a_0, a_0 + a_1, a_0 + a_1 + a_2, ...$ as coefficients. Then from the definition of multiplication it is quite clear that multiplying a generating function by the constant generating function would satisfy this requirement since we need all the coefficients of the other function to be 1. Thus we have to multiply by $\frac{1}{1-x}$, for example:

$$\frac{1}{1-x}\sum_{n=0}^{\infty} x^n,$$

$$\left(\frac{1}{1-x}\right)^2 = \sum_{n=0}^{\infty} nx^n.$$

We can do a more complex example with powers of 2, to show for example that $\sum_{i=0}^{n} 2^i = 2^n - 1$:

$$(1 + 2x + 4x^2 + 8x^3 + ...)\frac{1}{1-x}$$

As we will later show, this is:

$$\frac{1}{1-2x}\frac{1}{1-x}$$
$$\frac{1}{(1-2x)(1-x)}$$

We can separate it into two fractions, getting:

$$\frac{1}{1-2x} - \frac{1}{1-x} = \sum_{n=0}^{\infty}(2^n - 1)x^n.$$

Then, in general:

$$\frac{1}{1-x}\sum_{n=0}^{\infty} a_n x^n = \sum_{n=0}^{\infty}\left(\sum_{k=0}^{n} a_k\right) x^n.$$

Let us get back to the powers of $\frac{1}{(1-x)^n}$,specifically we want to generalize it and find the generating function for $\frac{1}{(1-x)^n}$.
We first observe that: $(\frac{1}{1-x})' = \frac{1}{(1-x)^2}$.
From this it is pretty logical to observe the following powers:
$\frac{1}{(1-x)^3} = (\frac{1}{1-x})'' \cdot \frac{1}{2}$
$\frac{1}{(1-x)^4} = (\frac{1}{1-x})''' \cdot \frac{1}{2 \cdot 3}$
$\frac{1}{(1-x)^5} = (\frac{1}{1-x})'''' \cdot \frac{1}{2 \cdot 3 \cdot 4}$
In general:
$\frac{1}{(1-x)^n} = (\frac{d^n}{dx^n}\frac{1}{1-x}) \cdot \frac{1}{(n-1)!}$.
Now, let us take a look at how the derivatives affect our basic generating function.

1. $\sum_{n=0}^{\infty} \frac{d}{dx} = \sum_{n=0}^{\infty} n \cdot x^{(n-1)} = 1 + 2x + 3x^2 + 4x^3...$
2. $\sum_{n=0}^{\infty}(\frac{d}{dx})^2 = \sum_{n=0}^{\infty} n \cdot (n-1) \cdot x^{(n-2)} = 2 + 6x + 12x^2 + 20x^3 + ...$

3. $\sum_{n=0}^{\infty}(\frac{d}{dx})^3 = \sum_{n=0}^{\infty} n \cdot (n-1) \cdot (n-2) \cdot x^{(n-3)} = 6 + 24x + 60x^2 + 120x^3 + ...$

We generalize:

$$\sum_{n=0}^{\infty}(\prod_{i=0}^{m}(n-i)) \cdot x^{(n-2)}.$$

But

$$\prod_{i=0}^{m}(n-i) = \frac{n!}{(n-m)!}$$

Therefore:

$$(\frac{d}{dx})^m \sum_{n=0}^{\infty} x^n = \frac{n!}{(n-m)!} \sum_{n=0}^{\infty} x^{(n-m)}.$$

We return to: $\frac{1}{(1-x)^m} = ((\frac{d}{dx})^{(m-1)} \frac{1}{1-x}) \frac{1}{(m-1)!}$, substituting we get:

$$\frac{1}{(1-x)^m} = \frac{1}{(m-1)!} \sum_{n=0}^{\infty} \left( \frac{n!}{(n+1-m)!} \cdot x^{(n+1-m)} \right).$$

Now we can use the definition we gave about binomials in Chapter 1.

$$\frac{1}{(1-x)^m} = \sum_{n=0}^{\infty} \binom{n}{m-1} \cdot x^{n-m+1)}$$

We can also get expressions for generating functions of the form: $1^n x + 2^n x^2 + 3^n x^3 + \ldots$. For example:

$$\frac{x}{(1-x)^2} = x + 2x + 3x^3 + \cdots + nx^n + \ldots$$

or

$$\frac{x(x+1)}{(1-x)^3} = x + 4x^2 + 9x^3 + 16x^4 + \cdots + n^2 x^n + \ldots$$

A deeper analyze of those generating functions can be found in the Appendix.

### 2.4.2 Other important simple forms

The way we proved that $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$ makes one think about how you could create other generating functions, specially after doing the recurrences in the previous chapter.

One can start playing with the different parts of the simple form and observe what happens and make deductions.

First of all, the numerator:1. As we proved before for finding the multiplicative inverse of a generating function the numerator of the simple form must equal the constant term of the generating function times the constant term of the denominator. As a specific case remember that a generating function has an inverse if and only if $a_0 \neq 0$. So for example $\frac{2}{1-x} = 2 + 2x + 2x^2 + ... = \sum_{n=0}^{\infty} 2x^n$ and $\frac{1}{2-x}$ has its first term equal to $\frac{1}{2}$.

On the other hand, the denominator is also very interesting, as it shows you the relation between some coefficients of the generating function. Note for example that for $\frac{1}{1-x}$ to be the inverse of the constant generating function we needed every term multiplied by x to be the next one multiplied by 1. Lets change for example the 1 for another number:

1. $\frac{1}{1-x} = 1 + x + x^2 + ... = \sum_{n=0}^{\infty} x^n$
2. $\frac{1}{2-x} = \frac{1}{2} + \frac{x}{4} + \frac{x^2}{8} + ... = \sum_{n=0}^{\infty} \frac{x^n}{2^{n+1}}$
3. $\frac{1}{3-x} = \frac{1}{3} + \frac{x}{9} + \frac{x^2}{27} + ... = \sum_{n=0}^{\infty} \frac{x^n}{3^{n+1}}$
4. $\frac{1}{c-x} = \frac{1}{c} + \frac{x}{c^2} + \frac{x^2}{c^3} + ... = \sum_{n=0}^{\infty} \frac{x^n}{c^{n+1}}$

Now we can analyze the coefficient of the x term:

1. $\frac{1}{1-x} = 1 + x + x^2 + ... = \sum_{n=0}^{\infty} x^n$
2. $\frac{1}{1-2x} = 1 + 2x + 4x^2 + ... = \sum_{n=0}^{\infty} (2x)^n$
3. $\frac{1}{1-3x} = 1 + 3x + 9x^2 + ... = \sum_{n=0}^{\infty} (3x)^n$
4. $\frac{1}{1-cn} = 1 + cx + c^2x^2 + ... = \sum_{n=0}^{\infty} (cx)^n$

Note that here we also get the special case with $c = -1$: $\frac{1}{1+x} = \sum_{n=0}^{\infty} (-1)^n x^n = 1 - x + x^2 - x^3 + x^4 ....$ If we continue our train of thoughts we can find a deeper relation: the characteristic polynomial. Those last generating functions were the geometric series commented on the first chapter that were recurrences of 1 degree and had characteristic polynomials of the form: $x - c = 0$. With a linear recurrence the generating function that has the sequence as coefficients has a simple form with a the characteristic polynomial of that recurrence as a denominator. Let us put some examples:

Fibonacci sequence:

$$\frac{1}{1-x-x^2} = 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + ...$$

$a_n = 3a_{n-1} - 4a_{n-2}$:

$$\frac{1}{1-3x+4x^2} = 1 + 3x + 5x^2 + 3x^3 - 11x^4 - 45x^5 - 91x^6 - 93x^7 ...$$

**In general:** if we have:

$$G(x) = \frac{n_0 + n_1x + n_2x^2 + \cdots + n_sx^s}{d_0 + d_1x + d_2x^2 + \cdots + d_mx^m}$$

We know that, if $G(x) = \sum_{k=0}^{\infty} g_kx^k$, then:

$$\sum_{k=0}^{\infty} g_kx^k \cdot (d_0 + d_1x + d_2x^2 + \cdots + d_mx^m) = n_0 + n_1x + n_2x^2 + \cdots + n_sx^s.$$

Therefore for $k > s$ we have that the $k^{th}$ coefficient of the numerator has to be 0 and thus:

$$\sum_{j=0}^{j-1} g_j \cdot d_{j-k} = 0$$

However, for $k \leqslant s$ we have:

$$\sum_{j=0}^{j-1} g_j \cdot d_{j-k} = d_0 \cdot g_k$$

Which is very similar to the characteristic polynomial of recurrences. Take for example the generating function:

$$G(x) = \frac{1 + x + 2x^2}{1 - 2x - 4x^2 - 8x^3}$$

Then for $k > 2$ we have:

$$g_k - 2g_{k-1} - 4g_{k-2} - 8g_{k-3} = 0.$$

Thus, its will follow the recurrence $G[k] = 2G[k-1] + 4G[k-2] + 8G[k-3]$ for $k > 2$.
However, for $k \leqslant 2$ we will have to take into account the numerator.

$$g_0 = 1$$

$$g_1 - g_0 = 1 \rightarrow g_1 = 2$$

$$g_2 - 2g_1 - 4g_0 = 2 \rightarrow g_2 = 10$$

which is equivalent to specify the first terms of a recurrence as we did in the previous chapter. This technique relates generating functions with recurrences and it will be very useful for calculating different expressions from generating functions.

## 2.5 Taylor's Theorem

A very important theorem for our purposes is Taylor's theorem (proved in [16, p. 736]) which finds a connection between analysis and our algebraic expressions: the generating functions. Taylor's theorem states the following:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + ... + \frac{d^{(n)}f(a)}{dx^n}(x-a)^n + ...$$

This enables us to approximate the value of a function around a point using polynomials. Concretely Taylor states that the Taylor polynomial of a certain degree $n$ given with his theorem it is the one with that degree that best approximates the function around that point. Also we have that with an infinite degree, a generating function, we obtain the exact value. A concrete kind of Taylor series are Maclaurin series which are Taylor series around $a = 0$.

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + ... + \frac{d^{(n)}f(0)}{dx^n \cdot n!}x^n + ...$$

## 2.6 Examples of generating functions

### 2.6.1 $sin(x)$ and $cos(x)$

Let,

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(1+2n)!} x^{1+2n}$$

$$cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n!} x^{2n}$$

Now that we have both formulas for the functions we can prove some things we use in calculus and other math fields, starting by proving $sin'(x) = cos(x)$ and $cos'(x) = -sin(x)$, which the reader will found in the Appendix.

Looking backwards, we can understand the construction of the series, given $\cos(0) = 1$ and $\sin(0) = 0$. We can get the generating function for $sin(x)$ with the Maclaurin series.

$$sin(x) = sin(0) + sin'(0)x + \frac{sin''(0)x^2}{2} + \frac{sin'''(0)x^3}{3!} + ...$$

Remember that the derivatives of the sine function can be determined doing mod 4.

$$\frac{d^n}{dx^n}\sin(x) = \begin{cases} \sin x & \text{if n mod 4=0} \\ \cos x & \text{if n mod 4=1} \\ -\sin x & \text{if n mod 4=2} \\ -\cos x & \text{if n mod 4=3} \end{cases}$$

This happens because $(-cos(x))' = sin(x)$ so a cycle is formed.
Now, since $sin(0) = 0$ and $cos(0) = 1$ we get the following fuzzy expression:

$$sin(x) = \sum_{n=0}^{\infty} \frac{(n \bmod 2)(-1)^{((n-1) \bmod 4)/2}}{n!} x^n$$

We can do the same for $cos(x)$ with the Maclaurin series.

$$cos(x) = cos(0) + cos'(0)x + \frac{cos''(0)x^2}{2} + \frac{sin'''(0)x^3}{3!} + ...$$

Again the derivatives of the cosine function can be determined doing mod 4.

$$\frac{d^n}{dx^n}\cos(x) = \begin{cases} \cos x & \text{if n mod 4=0} \\ -\sin x & \text{if n mod 4=1} \\ -\cos x & \text{if n mod 4=2} \\ \sin x & \text{if n mod 4=3} \end{cases}$$

Again, this happens because sin'(x)=cos(x) forming the cycle.

Now we use again $sin(0) = 0$ and $cos(0) = 1$ getting the following expression similar in form to the $cos(x)$ function:

$$cos(x) = \sum_{n=0}^{\infty} \frac{((n+1) \bmod 2)(-1)^{(n \bmod 4)/2}}{n!} x^n$$

We can change to get a formula without modulus using the fact that they have infinite coefficients to restate their values:

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(1+2n)!} x^{1+2n}$$

$$cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n!} x^{2n}$$

Apart from Calculus, we can also prove that one of the most basic and useful theorems of geometry also works for our series.

**Theorem 2.6.1** $sin^2(x) + cos^2(x) = 1$

**Proof:** Let us start with $cos^2(x)$:

$$cos^2(x) = \sum_{n=0}^{\infty} \sum_{k=0}^{n} \frac{(-1)^k(-1)^{n-k} x^{2k} x^{2n-2k}}{(2k)!(2n-2k)!} =$$

$$\sum_{n=0}^{\infty} \sum_{k=0}^{n} \frac{(-1)^n x^{2n}}{(2k)!(2n-2k)!}$$

Now we can apply a smart trick, since $\binom{2n}{2k} = \frac{(2n)!}{(2k)!(2n-2k)!}$; then $\frac{1}{(2k)!(2n-2k)!} = \binom{2n}{2k}(2n)!$.

$$\sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \sum_{k=0}^{n} \binom{2n}{2k}$$

Here we can use a Theorem enounced in Chapter 1, but proven in the Appendix.

$$\sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \cdot 2^{2n-1}$$

Now let us do $sin^2(x)$:

$$sin^2(x) = \sum_{n=0}^{\infty} \sum_{k=0}^{n} \frac{(-1)^k(-1)^{n-k} x^{1+2k} x^{1+2n-2k}}{(1+2k)!(1+2n-2k)!}$$

We apply the same trick as before:

$$\sum_{n=0}^{\infty} \frac{(-1)^n x^{(2n+2)}}{(2n+2)!} \sum_{k=0}^{n} \binom{2n+2}{1+2k}$$

$$\sum_{n=0}^{\infty} \frac{(-1)^n x^{(2n+2)}}{(2n+2)!} \cdot 2^{2n+1}$$

Now we have:

$$cos^2(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} \cdot 2^{2n-1}$$

and

$$sin^2(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{(2n+2)}}{(2n+2)!} \cdot 2^{2n+1}$$

Now observe that the term $n+1$ of $cos^2(x)$ is $-\frac{(-1)^n x^{(2n+2)}}{(2n+2)!} \cdot 2^{2n+1}$ which is the same as the term $n$ of $sin^2(x)$ but with the negative sign. Therefore, every $n^{th}$ term of $cos^2(x)$ will be canceled by the term $(n-1)$ of $sin^2(x)$ and only the term 0 of $sin^2(x)$ will remain, which is $\frac{(-1)^0 x^0}{2!} \cdot 2^1 = 1$. Therefore:

$$cos^2(x) + sin^2(x) = 1$$

□

### 2.6.2 $e^x$ and $ln(x)$

Let:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$ln(x) = \sum_{n=1}^{\infty} \frac{-(1-x)^n}{n}$$

which can be rewritten as:

$$ln(1-x) = \sum_{n=1}^{\infty} \frac{-x^n}{n}$$

From this we can prove that $e^x$ is the function that has its derivative equal to itself with $e^0 = 1$.

$$(e^x)' = \sum_{n=1}^{\infty} \frac{nx^{n-1}}{n!} = \sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!}$$

If we rename $n-1$ to be $n$ we get,

$$\sum_{n=0}^{\infty} \frac{x^n}{n!} = e^x$$

As we have done with the $e^x$ function we will find the derivative of $ln(x)$:
$(ln(1-x))' = (\sum_{n=1}^{\infty} \frac{-x^n}{n})' = \sum_{n=0}^{\infty} \frac{-nx^n}{n} = \sum_{n=0}^{\infty} -x^n$. We recognize the negative of the basic generating function.
$= \frac{-1}{1-x}$. It would be also interesting to show that $e^{ln(x)} = ln(e^x) = x$.

**Theorem 2.6.2** *$e^x$ is the inverse of $ln(x)$*

**Proof:** To do it we will use the Chain Rule, proved in [16, p. 203]. We will start proving $e^{ln(1-x)} = 1-x$. To do it easier, note that we have a bijection between x and $1-x$. Ergo, proving it for $1-x$ is the same as proving it for x.

$$e^{ln(1-x)} = 1-x.$$

$$(e^{ln(1-x)})'' = (\frac{e^{ln(1-x)}}{x-1})' = \frac{e^{ln(1-x)}}{(x-1)^2} - \frac{e^{ln(1-x)}}{(x-1)^2} = 0.$$

Now we apply one of the previous lemmas that $\frac{d^n}{dx^n}A(x) = 0 \Rightarrow A(x)$is a polynomial of degree$\leqslant n$. So we can conclude that $e^{ln(1-x)}$ is a polynomial of degree $\leqslant 1$, thus of the form $Ax+B$ where A and B are constants. Now take the first derivative, which we know it is equal to A:

$$\frac{e^{ln(1-x)}}{x-1} = A.$$

$$e^{ln(1-x)} = Ax - A.$$

Thus $B = -A$.
Let us find the constant term of: $e^{ln(1-x)} = \sum_{n=0}^{\infty}(\sum_{m=1}^{\infty}\frac{-x^m}{m})x^n$.
The constant term can only appear with $n=0$ otherwise it would have an $x$. Since $(\sum_{m=1}^{\infty}\frac{-x^m}{m})^0 = 1$ the constant term is 1 so B=1 and A=-1, thus:

$$e^{ln(1-x)} = 1-x$$

Thus, $e^{ln(x)} = x$. Now we will prove the other direction: $ln(e^x) = x$. We can again do the second derivative of $ln(e^x)$:

$$(ln(e^x))' = \frac{1}{e^x}\cdot(e^x)' = \frac{e^x}{e^x} = 1.$$

We can do the second derivative getting:

$$(ln(e^x))'' = (1)' = 0.$$

Thus, from Theorem 2.2.12 we know that $ln(e^x) = x + A$ in which A is the constant term of $ln(e^x)$. We can now substitute by the generating functions we had for both functions:

$$ln(e^x) = \sum_{n=1}^{\infty} - \frac{-(1-\sum_{m=0}^{\infty}\frac{x^m}{m!})^n}{n}$$

For $\sum_{m=0}^{\infty}\frac{x^m}{m!}$ the only constant term will appear when $m=0$ having $\frac{1}{1}=1$, thus we have that the constant part of $ln(e^x)$ is:

$$\sum_{n=1}^{\infty} - \frac{-(1-1)^n}{n} = 0.$$

Therefore $ln(e^x) = x$. Having proven both directions we can now say that $ln(x)$ is the inverse of $e^x$. □

## 2.7 Chapter conclusions

In this chapter we have proved a lot of important things about generating functions such as the fact that they are a ring. Those facts will be used constantly in further chapters, as will be simple forms. Moreover we have also showed some applications of generating functions and examples of how to manipulate generating functions.

In further chapters, however, generating functions will mainly be used for combinatorial purposes. For instance, in the next chapter, will be an introduction to the use of generating functions in combinatorics, which can now be done since the object is correctly defined.

# Chapter 3

# Generating functions for enumerative combinatorics

*In modern mathematics, algebra has become so important that numbers will soon have only symbolic meaning.*

After this introduction we get back to our starting problem:

**How many words of a determined size contain a given pattern P?**

Now that both the introduction to both combinatorics and to generating function are done it is time to merge both of them to solve this problem. We will first define some terms which are necessary for our work and the different equivalences we can find between generating functions and combinatorial operations. Then we will solve different combinatorial problems to show how to use the different combinatorial operations and equivalences and the power generating functions have with respect to the basic combinatorial tools shown in the first chapter.

## 3.1 Definitions

A **combinatorial class** is a pair $(\mathcal{A}, |\Delta|)$ such that $\mathcal{A}$ is a finite or infinite denumerable set and $|\Delta| : \mathcal{A} \to \mathbb{N}$ is a size function such that, for all $n \geqslant 0$, $\mathcal{A}_n = \{\alpha \in \mathcal{A} | |\alpha| = n\}$ is finite. Thus, knowing something about one set in the equivalence class can be useful to the other sets in the class.

Particularly, we are interested in pairs $(A, |\cdot|)$ where $A$ is a set and $|\cdot|$ is a function, which we will call 'size', which goes from the particular $A$ to the positive whole numbers; moreover, these numbers have to be finite. In this case, we will call the combinatoric class *admissible* and write $a_n$ for the number of elements in $A$ with size n.

With this hypothesis we can define an ordinary generating function associated to $(A, |\cdot|)$ with a formal power series:

$$A(x) = \sum_{n=0}^{\infty} a_n x^n.$$

Note that the number of elements in A can be infinite, since the elements can have any positive size, but given a size the number of elements in $A$ with that size must be positive.

We will also write $A_n$ for the subset of elements of $A$ with size n. In particular, the cardinal of $A_n$, $|A_n|$, is equal to $a_n$. Thus,

$$A(x) = \sum_{n=0}^{\infty} |A_n| x^n$$

Take the example of the set $B$ of binary words without any restriction and consider as size the longitude of the word. We show in the first chapter that the number of binary words of length n is $2^n$ because for each position in the word you can choose a 0 or a 1.
$2^n$ is a whole number for n whole; moreover, it is positive and finite; therefore it is an acceptable combinatorial class. We can associate to it the generating function:

$$W(x) = \sum_{n=0}^{\infty} 2^n x^n = \frac{1}{1-2x} = 1 + 2x + 4x^2 + \cdots + 2^n x^n + \dots$$

## 3.2 Multivariable generating functions

Quite often, we may also be interested in studying particular parameters in these combinatoric classes. In a more rigorous form: given a combinatoric class $(A, |\cdot|)$, let $\chi : A \to \mathbb{N}$ a parameter.

To solve this, we use an auxiliary variable to take into account this parameter giving multivariable generating functions (MGF).

$$A(u, x) = \sum_{a \in A} x^{|a|} u^{\chi(a)}.$$

**Theorem 3.2.1** $A(1, x) = A(x)$

**Proof:** $A(1, x) = \sum_{a \in \mathcal{A}} x^{|a|} u^{\chi(a)}$
$= \sum_{a \in A} x^{|a|} 1^{\chi(a)}$

$= \sum_{a \in \mathcal{A}} x^{|a|} \cdot 1$

$= \sum_{a \in \mathcal{A}} x^{|a|} = \mathcal{A}(x).$ □

Take for example the function $W(u, x)$ where u points the number of 1's and x counts the total size of the word. As we will see later in the chapter this function is simply: $\frac{1}{1-ux-x}$. Take $u = 1$ and you get: $\frac{1}{1-2x}$ which we showed to be $W(x)$. In fact, multivariable generating functions will be used later in a similar example.

## 3.3 Symbolic method

The Symbolic method provides a framework to translate combinatorial constructions between combinatorial classes to equations, most of them algebraic between the associated generating functions. In what follows we will present the methodology to apply these ideas by showing in each generic case the translation between the two worlds, the combinatorial one and the algebraic one.

## 3.4 Basic constructions

### Empty class

We simply define a special but important class: $(\emptyset, |\cdot|)$, where we assume $|\emptyset| = 0$, which we will call the *empty* combinatorial class.

### Units

Very often we have to work with units, for example in case of binary words 0 and 1 would be two units of size 1 ( because they increase the size of the word by one ). $\mathcal{E} = (\bullet, |\cdot|)$, where $|\cdot| = 1$. Recall that we may have units of other sizes such as: $|\cdot| = 2$.

### Union

Given two classes $(A, |\Delta|)$ and $(B, |\Delta|)$ with $A \cap B = \emptyset$. We define $C = A \cup B$.
Then the size over $C$ is the one from $A$ and $B$.
It is important to notice that in case $A \cap B \neq \emptyset$ then we can consider a copy $\widetilde{B}$ of B coloring the objects in a different color. And now consider $A \cup \widetilde{B}$ which is $\emptyset$ and then get $C = A \cup \widetilde{B}$.

## Cartesian product

Again, given two classes $(A, |\Delta|)$ and $(B, |\Delta|)$, we define the cartesian product of $A$ and $B$ to be the set $A \times B = \{(a, b) : a \in A, b \in B)\}$.
The size of an element $(a, b)$ of $A \times B$ is $|a| + |b|$.
The combinatoric class that results is called the product from the initial classes.

## Sequence

Now, given a single class $(A, |\cdot|)$ we can consider the set:

$$\mathrm{Seq}\,(A) = \varepsilon \cup A \cup (A \times A) \cup (A \times A \times A) \cup ... = \bigcup_{r=0}^{\infty} A \times \overset{r}{\ldots} \times A.$$

This is probably one of the most important operations and at the same time one of the hardest to understand, so let us explain it carefully with the example of binary words.

In binary words we have two units such as $\circ$ and $\bullet$ both of size 1. Then the binary words are:

1. The empty binary word $(\mathcal{E})$
2. The units $\circ$ and $\bullet$ $(\mathcal{A})$
3. The words of size two given by the cartesian product: $\circ\circ, \circ\bullet, \bullet\circ, \bullet\bullet$, which is $(\mathcal{A} \times \mathcal{A})$.
4. The words of size three you can get with those two units such as $\circ \circ \circ$ or $\bullet \circ \bullet$ which is $(\mathcal{A} \times \mathcal{A} \times \mathcal{A})$
5. etc.

A particular element of Seq(A) it is written as $(a_1, ..., a_k)$ for a particular natural $k$ and its size is $|a_1| + \cdots + |a_k|$. We call this class a sequence generated with its initial class. For restricted sequences of a subgroup r of naturals we define it as :

$$\mathrm{Seq}_R(A) = \bigcup_{r=0}^{\infty} A \times \overset{r \in R}{\ldots} \times A$$

.

## Pointing

Given a combinatorial class $(A, |\cdot|)$ we consider the set

$$A^{\bullet} = \bigcup_{r=0}^{\infty} A_r \times \{\epsilon_1, \ldots, \epsilon_r\},$$

where each of the $\epsilon_i$ has size 0. This combinatorial construction is interpreted as follows: given an element $a \in A$ of size n it can be interpreted as the union of $n$ atoms (connected in a certain way). Then with the previous operation we are pointing one of those atoms giving the pointing of the combinatorial class $\mathcal{A}$.

### Substitution

Given two combinatorial classes $(A, |\cdot|)$ and $(B, |\cdot|)$ we define a new set:

$$A \circ B = \bigcup_{n=0}^{\infty} A_n \times (B \times \overset{n}{\ldots} \times B),$$

called *composition* of combinatorial classes. This construction can be interpreted as follows: given an element $a$ of $A$ of size n, we substitute each of its $n$ atoms for an arbitrary element of $B$. Thus, the size of the object created is the sum of the sizes of the elements we choose of $B$.

## 3.5 Equivalence with generating functions

1. **Units** Recall that we talked about different sizes of units: in general we can say that a unit is $\mathcal{E}$ represented as $x^{|\mathcal{E}|}$. In particular:
   If $|\mathcal{E}| = 0$ then it is represented as 1.
   If $|\mathcal{E}| = 1$ then it is represented as $x^1$.
   if $|\mathcal{E}| = 2$ then it is represented as $x^2$.
   And so on.
2. **Union** This one it is pretty simple, as in almost every field in mathematics union is addition: $A \cup B$ is represented in generating functions as $A(x) + B(x)$. We find the reason in the definition of union ( adding the elements of a particular size ) and addition of generating functions to be equivalent.
3. **Cartesian product** Another classical representation: $A \times B$ is represented as $A(x)B(x)$. To find the reason why it is like this you just have to recall the definition of multiplication given in the first chapter.
4. **Sequence** The sequence $\mathrm{Seq}(\mathcal{A})$ is represented as $\frac{1}{1-A(x)}$. Note that:
   (a) The famous $\frac{1}{1-x}$ is a special case because it signifies the sequence you get with a single unit of size 1, which is one word for every size.
   (b) You cannot have any element of size 0 otherwise there is a double contradiction: there is no polynomial representation and it is not a combinatorial class since you can compose infinitely many elements of size 0 giving an infinite amount of size 0 words.

5. **Pointing** $\mathcal{A}^{\bullet}$ is equivalent to $x\frac{\partial}{\partial x}A(x)$.

   The derivative must be understood as a formal derivative. It is simply a clever way of representation:

   Notice that we need to multiply the term of size 1 by 1, the one of size 2 by 2 and the term of size n by n. The differentiation of $x^n$ is $nx^{n-1}$ so by first differentiating and then multiplying by x we get $nx^n$ in each term.

6. **Substitution** As with union and cartesian product, the definition of composition is the same as substitution so $\mathcal{A} \circ \mathcal{B}$ is represented in generating functions as $A(B(x))$.

## 3.6 Basic examples

### 3.6.1 Natural numbers

**Construction of natural numbers** We can define the natural numbers as the set $1, 2, ...$ with size ($|i| = i$). We can define this set in terms of the combinatoric class $\mathcal{A} = \bullet$ with $|\bullet| = 1$ as $\operatorname{Seq}(\mathcal{A}) = \frac{1}{1-x} = 1 + x + x^2 + x^3 + ....$
In other words, there is a single natural number of every positive whole size.

**Coverage of natural numbers with 1's and 2's** We will do it in two ways.
**1. Order does matter** We define the combinatorial class $\mathcal{A} = \circ, \bullet$ with $|\circ| = 1, |\bullet| = 2$.
In generating functions $\mathcal{A} = x + x^2$.
Now simply take:

$$\operatorname{Seq}(\mathcal{A}) = \frac{1}{1-(x+x^2)} = \frac{1}{1-x-x^2} = 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + ...$$

Note that this is the Fibonacci sequence! The interpretation is the following: each string of signs of $\circ, \bullet$ must end with one of the two. So the structure will follow the fibonacci recurrence: $F_n = F_{n-1} + F_{n-2}$, because we can have $F_n$ by getting a sequence of size $n-2$ and adding a $\bullet$ or having a sequence of size $n-1$ and adding a $\circ$.

**2. Order does not matter** This one is a little bit more tricky because it involves more than one operation. We can simply look for *ordered* string of signs where $\circ$ must always go before $\bullet$. To do that we make two combinatorial classes, the one you get only with $\circ$ and the one you get only with $\bullet$:

$$\operatorname{Seq}(\circ) = \frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$$
$$\operatorname{Seq}(\bullet) = \frac{1}{1-x^2} = 1 + x^2 + x^4 + x^6 + ... + x^{2n} + \dots$$

Now, you can imagine each coverage as all the possible combinations of taking 1 group of all $\circ$ and then 1 group of all $\bullet$ after with their sizes adding the given size. But that is simply the definition of cartesian product!

$$\text{Seq}(\circ) \times \text{Seq}(\bullet) = \frac{1}{1-x}\frac{1}{1-x^2} = \frac{1}{(x-1)^2(x+1)}$$

$$\frac{1}{(x-1)^2(x+1)} = 1 + x + 2x^2 + 2x^3 + 3x^4 + 3x^5 + \dots$$

It is important to notice that this notion of order would have been much more difficult with the previous tools we had of combinatorics and now we have solved the problem in a very simple way.

### 3.6.2 Binary words counting 1's

This problem was showed at the beginning of the chapter and now it is also quite easy to show:

Take the combinatorial class $\mathcal{B} = \{'0',' 1'\}$.

We have two types of sizes, $x$ =the size of the word and $u$ =the number of 1's. Then note that we have two units: '0' which has size $x = 1$ and size $u = 0$.
'1' which has size $x = 1$ and size $u = 1$.
Therefore: $|'0'| = x$ and $|'1'| = u \cdot x$. Now we can simply do the sequence of this:

$$Seq\mathcal{B} = \frac{1}{1-x-ux} = 1 + (u+1)x + (u+1)^2x^2 + (u+1)^3x^3 + ... + (u+1)^n x^n + ...$$

From that we get that for a particular size of the word m and number of 1's equal to n the number of strings with both conditions will be the coefficient with degree of u equal to n in $(u+1)^m x^m$. Now we can use the binomial theorem to know that the coefficient of degree m will be:

$$\binom{m}{n} u^n x^m$$

### 3.6.3 Exchange

**Concrete example** Suppose a machine can only return moneys of 1,2,5 and 20 cents of . In how many ways can it return n cents? Again we have $\mathcal{A} = *, \star, \circ, \bullet$ with $|*| = 1, |\star| = 2, |\circ| = 5, |\bullet| = 20$.

We create all the generating functions as we did with the coverage with 1's and 2's:

$$Seq(*) = \frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$$

$$Seq(\star) = \frac{1}{1-x^2} = 1 + x^2 + x^4 + x^6 + \dots$$

$$Seq(\circ) = \frac{1}{1-x^5} = 1 + x^5 + x^{10} + x^{15} + \dots$$

$$Seq(\bullet) = \frac{1}{1-x^{20}} = 1 + x^{20} + x^{40} + x^{60} + \dots$$

Now we multiply all them, since we want 1 group of each:

$$Seq\ast \times Seq\star \times Seq\circ \times Seq\bullet = \frac{1}{(1-x)(1-x^2)(1-x^5)(1-x^{20})}$$

**Generalization** It is quite simple to see that we can always to the same procedure, so in general:
$\mathcal{A} = \{a_1, a_2, ..., a_n\}$ with $|a_1| = v_1, |a_2| = v_2, ..., |a_n| = v_n$ the generating function associated will be:

$$\prod_{i=0}^{n} \mathrm{Seq}\,(a_i) = \prod_{i=0}^{n} \frac{1}{1-x^i}$$

## 3.7 More complex examples

### 3.7.1 Compositions

This problem is in essence, very similar to the coverage of the naturals with 1's and 2's, but now we can use every positive number. So:

$$\mathcal{A} = \mathbb{N}$$

And the associated generating function is therefore:

$$\frac{x}{1-x} = x + x^2 + x^3 + \dots$$

Note the importance of the x on the numerator. Without it we would have a problem since the term of degree 0 would appear and with it there are infinitely many ways of adding every number since we can always add another 0. Now, as we did last time we can do the sequence of this:

$$\mathrm{Seq}\,(\mathcal{A}) = \frac{1}{1-\frac{x}{1-x}} = \frac{(x-1)x}{2x-1} = x + 2x^2 + 4x^3 + 8x^4 + ...$$

Note that every term is of the form $2^{n-1}x^n$.

We can prove this formula as follows: Imagine a group of n 1's and put the first 1 in a box. For the other 1's you can choose to put it in the same box or start a new box giving a composition which will be the number of 1's in each box.

Since you have $n-1$ places with 2 options each, the total possible combinations are $2^{n-1}$.

We can also have a graphical visualization by imagining n points with $(n-1)$ spaces between them. For each space you can put a bar of separation or not; ending with a list of boxes; again since we can put or not a bar in each space we have $2^{n-1}$ possibilities.

### 3.7.2 Partitions

Partitions is probably one of the hardest problems of number theory. The first approximation was done by Hardy and Ramanujan in 1918:

$$p(n) \sim \frac{exp(\pi\sqrt{2n/3})}{4n\sqrt{3}} \text{ as n} \to \infty$$

Some years after, in 1937, a better expression was found by Hans Rademacher using Ford circles, Farey sequences, modular symmetry and the Dedekind eta function. As you can see the formula is quite dense:

$$p(n) = \frac{1}{\pi\sqrt{2}} \sum_{k=1}^{\infty} \sum_{0 \le m < k; (m,k)=1} e^{\pi i [s(m,k) - 2nm/k]} \sqrt{k} \frac{d}{dn} \left( \frac{sinh\left(\frac{\pi}{k}\sqrt{\frac{2}{3}(n-\frac{1}{24})}\right)}{\sqrt{n-\frac{1}{24}}} \right)$$

where $s(m,n)$ stands for the Dedekind sum.

Compared with this complexity we have the approach with generating functions which is simply the problem of the exchange with $\mathcal{A} = \mathbb{N}$ so:

$$P(x) = \prod_{n=1}^{\infty} \frac{1}{1-x^n}$$

With every $n^{th}$ coefficient giving the number of compositions of n. This is a very clear example of the power of generating functions in the sense that they make complex things simple.

### 3.7.3 Triangulations of a polygon

Introducing combinatoric tools we put triangulations as an example of a problem we could not solve using just recurrences or combinatorial formulas. However, with generating functions this problem is now solvable, in two different approaches: Looking at triangles we can take an edge in the n-gon and call it your 'root' edge. This edge has to have some triangle associated to it; to complete this triangle a vertex from the rest has to be chosen, as Figure 3.7.3 shows.

This creates both a triangle and two other polygons that will be 'stuck' to the triangle on its other two edges. Then, we only have to triangulate those two other polygons. Therefore all the triangulations having this triangle are the possible triangulations of the first polygon with all the possible triangulations of the second polygon, as shown in Figure 3.7.3.

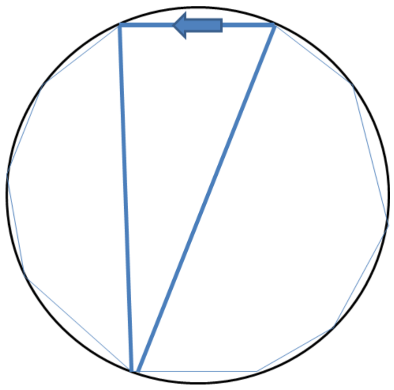


Figure 3.1: Choosing a root edge in the polygon

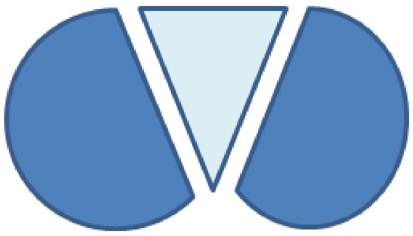


Figure 3.2: Symbolic structure of a triangulation

Therefore, with the symbolic method, let A be the generating function associated with triangulations then:

$$\mathcal{A} =' |' \cup \mathcal{A} \times \bigtriangledown \times \mathcal{A}$$

Note that is the element of size one, since we have added a triangle and that the unit is an edge, since this is the base case when in one of the two sides there is not a polygon, then an edge is stuck. Then we can easily translate this to generating functions:

$$A = 1 + AxA = 1 + A^2x$$

Which is equivalent for:

$$xA^2 - A + 1 = 0$$

Now we can use the quadratic equation which gives us two results:
$A = \frac{1+\sqrt{1-4x}}{2x}$ or $A = \frac{1-\sqrt{1-4x}}{2x}$. However $\frac{1+\sqrt{1-4x}}{2x} = \frac{1}{x} - 1 - x - 2x^2 - 5x^3 - ...$ so this cannot be the solution since it must have integer positive coefficient and exponents. On the other hand $A = \frac{1-\sqrt{x^2+4x}}{2x} = 1 + x + 2x^2 + 5x^3 + 14x^4 + ...$ so this must be the solution. Note that these numbers are the famous Catalan numbers of the form: $\frac{1}{n+1}\binom{2n}{n}$.

### 3.7.4 Decompositions of a polygon in quadrilaterals

As we did with triangulations of a polygon, we can use the Symbolic Method to set up an equation for the generating function. This time, however we have three different polygons attached to the quadrilateral, as Figure 3.7.4 shows.

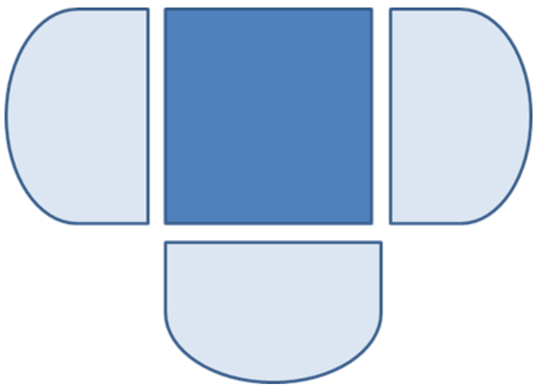


Figure 3.3: Symbolic structure of a Decomposition in quadrilaterals

Therefore we get the following equation in symbolic method:

$$\mathcal{B} =' |' \cup \mathcal{B} \times \mathcal{B} \times \square \times \mathcal{B}.$$

Which, translated to generating functions is:

$$\mathcal{B} = 1 + \mathcal{B}^3 x,$$

since, again, the edge is the unit and the square the element of size 1. Again, we can put it as: $\mathcal{B}^3 x - \mathcal{B} + 1 = 0$ and get the following solutions using the cubic equation:

$$A_1 = \frac{\sqrt[3]{\sqrt{3}\sqrt{27x^4 - 4x^3} - 9x^2}}{\sqrt[3]{2}3^{2/3}x} + \frac{\sqrt[3]{\frac{2}{3}}}{\sqrt[3]{\sqrt{3}\sqrt{27x^4 - 4x^3} - 9x^2}},$$

$$A_2 = -\frac{1 + \imath\sqrt{3}}{2^{2/3}\sqrt[3]{3}\sqrt[3]{\sqrt{3}\sqrt{27x^4 - 4x^3} - 9x^2}} - \frac{(1 - \imath\sqrt{3})\sqrt[3]{\sqrt{3}\sqrt{27x^4 - 4x^3} - 9x^2}}{2\sqrt[3]{2}3^{2/3}x},$$

$$A_3 = -\frac{(1 + \imath\sqrt{3}\sqrt[3]{\sqrt{3}\sqrt{27x^4 - 4x^3} - 9x^2}}{\sqrt[3]{2}3^{2/3}x} - \frac{1 - \imath\sqrt{3}}{2^{2/3}\sqrt[3]{3}\sqrt[3]{\sqrt{3}\sqrt{27x^4 - 4x^3} - 9x^2}}.$$

However, again, only the third one has integer coefficients, therefore $A = A_3 = 1 + x + 3x^2 + 12x^3 + 55x^4 + 273x^5 + \ldots.$

### 3.7.5 Dissections of a n-gon

Finally, the objective is to find the general case: dissections. Dissections are divisions of a convex n-gon, without any crossings inside the polygon and no condition about the final polygons into which the initial polygon is divided.

However, it is impossible to count the number of dissections in term of the number of final polygons, as we did with quadrilaterals, but we must do it in function of the vertices. Generating functions must count finite sets and the set of dissections in 1,2 or any other positive integer polygons is infinite. Take for example dissections into 1 polygon, which is not dissecting anything. Then, there are infinitely many dissections of this form since we can increase the number of vertices indefinitely and each different n-gon would be by itself a dissection of degree one, giving infinitely many solutions.Therefore, we have to do it as a function of the number of vertices.

We can follow the same procedure we did with triangulations knowing that introducing a m-gon into the n-gon will create m other polygons and will decrease the number of available vertices by $m-2$. The symbolic method representation would then be:

$$A = | \bigcup_{m=2}^{\infty} \frac{A^m}{\bullet^{m-1}}.$$

Translated into generating functions:

$$A = x^2 + \frac{A^2}{x} + \frac{A^3}{x^2} + \frac{A^4}{x^3} + ...$$

Now, we can do a variation of the classic $\frac{1}{1-x}$ by doing:

$$\frac{A}{1-\frac{A}{x}} = A + \frac{A^2}{x} + \frac{A^3}{x^2} + \frac{A^4}{x^3} + ...$$

Therefore,

$$A = \frac{A}{1-\frac{A}{x}} - A + x^2.$$

We can solve again for A using the quadratic equation giving $A = \frac{x^2-x\sqrt{x^2-6x+1}+x}{4}$ or $A = \frac{x^2+x\sqrt{x^2-6x+1}+x}{4}$. It turns out that the one with positive coefficients is $A = \frac{x^2-x\sqrt{x^2-6x+1}+x}{4} = x^2+x^3+3x^4+11x^5+45x^6+197x^7+...$

## 3.8 Chapter remarks

With this, we have shown the power of generating functions and the symbolic method by solving some very hard problems that were unapproachable using recurrences. However, there is still no clue about whether the symbolic method could be used to get generating functions for words, and how to do it.

Chapter 4

# Introduction to string matching algorithms

*If 90% of the ideas you generate aren't absolutely worthless, then you're not generating enough ideas.* **Martin Artin**

## 4.1 Connection with generating functions

In the first part of our research we have developed a series of mathematical tools to solve combinatorial problems, from the simplest ones using formulas to some very complex ones as dissection of polygons or partitions. After this, we moved on to the heart of the study: sequences; we used generating functions to analyze and count mathematically how many sequences of that kind or that other should appear.

Now, we use computer science to look at the same problem in a different perspective, probably more practical and as useful. It is not a supplementary part, but complementary, as both parts will help one another and at the same time help getting to deeper conclusions. For example, in chapter 6 , we will see a very deep conceptual connection between generating functions and automatons of search. Another example could be the last chapter, where we apply some programs to big data sets; if math predicts one thing and the result of the program tells us another, something interesting could be found.

## 4.2 Concept of string matching

As we find it in *Introduction to Algorithms*([2]):
The ***string-matching problem*** is the following. Assume that the text is an array $T[1...n]$ of length $n$ and that the pattern is an array $P[1...m]$ of length $m \leq n$. We further assume that the elements of $P$ and $T$ are characters drawn from a finite alphabet $\Sigma$. Some examples of alphabets could be the binary alphabet: $\Sigma = \{0, 1\}$, the Latin alphabet: $\Sigma = a, b...z$ or the genetic code: $\Sigma = A, C, T, G$. Those three alphabets are the ones we will use more often in this second part of the work.

We say that pattern $P$ occurs with shift s in text T ( or, equivalently, that pattern P occurs beginning at position $s + 1$ in text T) if $0 \leq s \leq n - m$ and $T[s + 1...s + m] = P[1..m]$ (that is $T[s + j] = P[j]$, for $1 \leq j \leq m$).

If P occurs with shift $s$ in $T$, then we call $s$ a valid shift; otherwise, we call s an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T. Although sometimes the problem will be reduced to know if there are such shifts of pattern P in T.

The string matching problem is very important in present computer science and probably will become increasingly important as data becomes more and more plentiful. With huge amounts of data we need computers to search for us in them, as it would take thousands of years for us to find something. However, using computers does not guarantee fast results in searches as there has been an exponential growth of data production in the last few decades and this tendency it is likely to continue.

Take the case of Google and the internet: for the last two decades people from all over the world have been adding content to the net and it would be materially impossible for anyone to read all the content in the net. Here is where Google comes, finding exactly what you want. Take the example of text-editing programs you use everyday, or the program you use to look in your own computer; without string matching algorithms search would mean so much of waiting time that the search itself would become useless.

## 4.3 Notation and terminology used

Let $\Sigma^\star$ denote the set of all finite-length strings formed using characters from the alphabet $\Sigma$. The zero-length **empty string** will be called $\varepsilon$ and it also belongs to $\Sigma^\star$. As with generating functions

the size of a string is denoted as $|s|$. The *concatenation* of $x$ and $y$ consists of the characters from x followed by the characters from y.
**prefix**: w is a prefix of string x when $x = wy$ for some string $y \in \Sigma^\star$. We denote it as: $w \sqsubset x$
**suffix**: w is a prefix of string x when $x = yw$ for some string $y \in \Sigma^\star$. We denote it as: $w \sqsupset x$
From these definitions we know that $\varepsilon$ is both a prefix and a suffix of every string. Also note that $\sqsubset$ and $\sqsupset$ are transitive relations. $A \sqsubset B \wedge B \sqsubset C \Rightarrow A \sqsubset C$ and $A \sqsupset B \wedge B \sqsupset C \Rightarrow A \sqsupset C$.

## 4.4 Efficiency and asymptotic notation

*Note: this section is a brief overview of the concept of efficiency in Computer Science. Thus, it is only recommended for readers without any experience in this field.*

### 4.4.1 Concept of Efficiency

Programs have a lot of aspects that have to be taken into account, from time consumption to interaction with other devices or portability. The algorithms at Google might be very fast, but they have to be equally liable and be able to function in mobile phones, computers, Mac's equally well and also easy to make use of them. However, in this case we will only be interested in usage of resources ( primarily time and memory ) and liability since the programs are for our own use.

We can assume liability since there is not much theory about it, just simply assuring that the program does not fail even if it is by a fault of the user.

Usually we are more interested in time consumption. Why? Because computers have a bounded capacity. If computers were infinitely fast, we would not care much about *efficient* algorithms since all would take the same time. However, we *are* bounded by the capacity of the computers, so we have to care about time. Normally, memory is not such a problem and it is usually easier to handle as well.

Although memory might not seem a problem since we all have gigabytes or even terabyte in our computers, we have to remember that it is very useful to have all the data used by our computer in the RAM memory ( a fast access part of the computer ) which is usually bound by less than 10 GB in personal laptops right now. Using more memory than the one in the RAM would imply using the one in the hard drive, much slower to use, a fact that time efficiency does not take into account.

### 4.4.2 Asymptotic notation

To know the running time of an algorithm you have to take into account the speed of the hardware and software, but also your own algorithm. In fact, algorithms can (and usually do) play a crucial part in the running time.

Let us for example compare a quadratic algorithm with a linear algorithm, both with respect to n. Even if the computer of the quadratic algorithm is 100 times faster than the linear algorithm for just an input of $n = 10^6$:

$$\frac{(10^6)^2 instructions}{10^9 instructions/second} = 1000 seconds.$$

$$\frac{10^6 instructions}{10^7 instructions/second} = 0.1 seconds.$$

Moreover, we can just look at the asymptotic behavior without looking at all the coefficients in the algorithm function and just taking the leading term, without looking at the constants. For example $2x^3 + x^2$ is $O(x^3)$, the same category as $10000x^3 + x$ or simply $x^3$. This might sound confusing but it is a fairly good approximation in most cases.

Figure 4.1 shows the difference between the different degrees of polynomials versus the effect that constants may have. As one can see, constants do not count much since asymptotically slower algorithms will always end up taking more time, which is the example of constant time: 1,000,000 and $1 \cdot x^3$. String-matching algorithms are quite similar in asymptotic behavior, so to be more

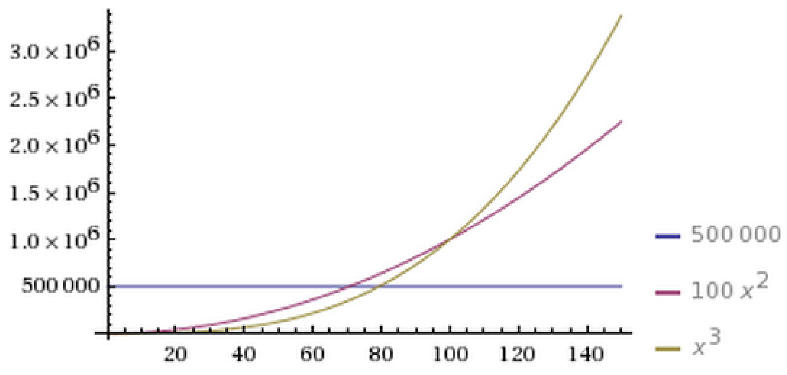


Figure 4.1: Comparison of $10^6, 100x^2, x^3$

precise we will include 4 types of notations.

**Best case:** the complexity in case it is the best case, or almost like it. Hardly used. For example having to sort an almost sorted list.

**Average case:** sometimes the best and worst case are very different, in order to resolve this

problem we can look at the average performance. Example: Monte Carlo ( randomized ) algorithms
**Worst case:** a very used notation, to put an upper-bound to the algorithm. It might be useful to assure that an algorithm will be finished in a certain amount of time.
Although we will not use most of them, to have a global perspective we divide the algorithms mainly in those categories:

$$\lg n < \sqrt{n} < n < n \lg n < n^2 < n^3 < ... < 2^n < n!.$$

Moreover algorithms can depend in more than one variable. In fact, some of the algorithms presented in this work will. However, in terms of notation it does not change much from *monovariable* algorithms, having only to take the combination required such as $n \cdot m$, $lgmn$, $m^n$, etc.

It is also important to notice the big difference between the polynomial algorithms and the non-polynomial ( exponential ) algorithms. We call NP-complete problems, the ones for which we do not have a polynomial solution. One of the most important problems in Computer Science and math is the P-NP problem, to know if all problems have a polynomial solution or not; a very important issue since NP problems do not have a practical solution since for very small cases it will take for centuries to compute the result. An example of that is the Traveling Salesman Problem, where you have to find the path to visit a list of cities and return to the start in the least amount of time.

Fortunately, as we will see, the string-matching problem is a problem with polynomial solution. Note that we can use the same notation for memory efficiency as for time efficiency, so essentially we have all the notation required to study the algorithms presented later.

## 4.5 The naive string-matching algorithm

### 4.5.1 Description of the algorithm

The term *naive* probably refers to the immediateness or facility with which it comes to mind when thinking about how to find a substring in a bigger string or the perfect logic it uses, just coming straight from the definition.

We want to find all valid shifts that checks $P[1...m] = T[s+1...s+m]$. The pseudo algorithm is the following:

Essentially, we look at every shift and check if it is valid or not. However, we have to take into account that looking if the shift is valid is itself linear so the pseudo code should look more like:

**Algorithm 1** Naive String Matcher

```
Require: pattern P, text T
  $n \leftarrow$ length[T]
  $m \leftarrow$ length[P]
  for $s = 0$ to $n - m$ do
    if P[1..m]=T[s+1..s+m] then
      print ”Pattern occurs with shift” s
    end if
  end for
```

**Algorithm 2** Improved Naive String Matcher

```
Require: pattern P, text T
  $n \leftarrow$ length[T]
  $m \leftarrow$ length[P]
  for $s = 0$ to $n - m$ do
    $b \leftarrow$ true
    for $k = 1$ to $m$ and while b is true do
      if $P[k] \neq T[s + k]$ then
        $b \leftarrow$ false
      end if
    end for
    if b then
      print ”Pattern occurs with shift” s
    end if
  end for
```

### 4.5.2 Complexity

The complexity of this algorithm is $\Theta(n-m+1)$ times a *for* with complexity $\Theta(m)$ so a complexity $\Theta((n-m+1)\Theta(m)) = \Theta(nm-m^2)$, which is quite slow. Since its bounded in the worst case to $m = \frac{n}{2}$ it will give $\Theta(\frac{n}{2}n - \frac{n^2}{4}) = O(n^2)$.

However we can make the algorithm much faster by making the loop stop when we already know a particular shift is invalid.
Then, the complexity in the worst case is still the same: $\Theta(mn - m^2)$, for example for a pattern in which all shifts are valid. So in general we can say that the algorithm with this small, but important, improvement runs in $O(mn)$ instead of $\Theta(mn)$.
In the best case, the complexity decreases substantially to $\Theta(n)$, for example a string in which all the characters are different to the first character in the substring we are looking for.

Moreover, there is also a great improvement in the average case:
For example, consider a random word with the latin alphabet ( 26 caracters ), the probability that it ends in the first letter is $\frac{25}{26}$ which is more than a 96% of cases. This gives as very low average cases as we can see in table of figure 4.2: We can perform a deeper analyze of those tables and ask some questions about them. First, we observe that the probability of ending at the $i^{th}$ character in an alphabet of length L. This is equal to the probability of not ending in any of the steps before i and that the strings differ in the $i^{th}$ character. The probability that two characters coincide in an alphabet with L characters is $\frac{1}{L}$, the probability that they do not is therefore: $\frac{L-1}{L}$. Therefore the probability of ending in the $i^{th}$ character in an alphabet of length L is:

$$P = \frac{L-1}{L} \cdot (\frac{1}{L})^{i-1} = \frac{L-1}{L^i} \approx L^{-(i-1)}.$$

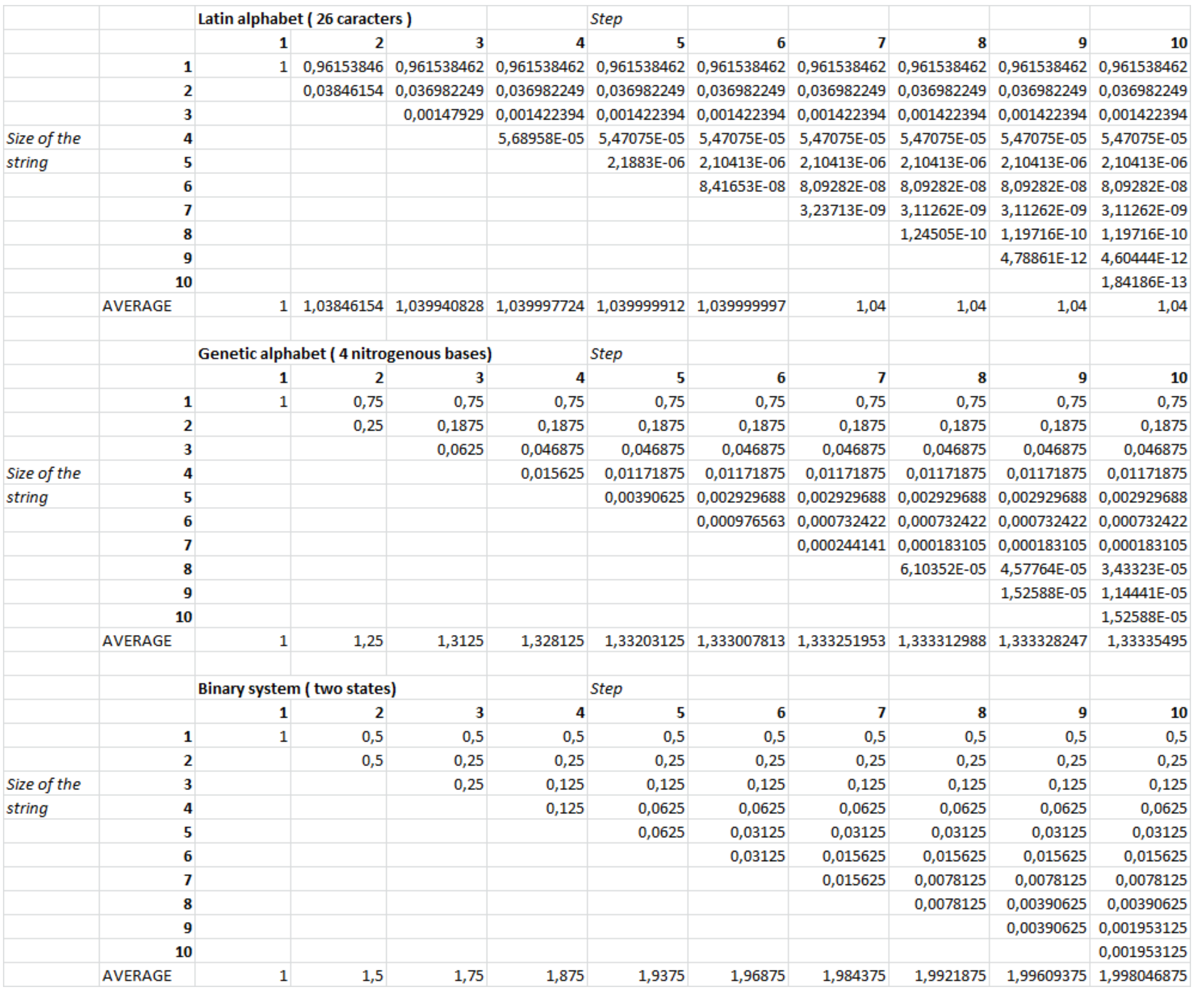
It would also be interesting to know the average number of steps depending on n, the length of the string, and L, the length of the alphabet. We have then to do a weighted average, the turn $i$·probability of finishing at turn i. For every turn $i < n$ the probability would be the same as stated before but the probability of finishing at the $n^{th}$ turn is the probability of not finishing in any of the other terms. Therefore the total average is:

$$\sum_{i=0}^{n-1} i \cdot \frac{L-1}{L^i} + L^{-n}.$$

Now we want to get a closed expression. First we want to know the value of the sum. For that we will simplify the expression taking $u = \frac{1}{L}$ and factoring out $(L-1)$ getting:

$$(L-1) \cdot \sum_{i=0}^{n-1} i \cdot u^i.$$

Now we prove a little Lemma:



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Latin alphabet ( 26 caracters )** | | | *Step* | | | | | | |
| | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| | **1** | 1 | 0,96153846 | 0,961538462 | 0,961538462 | 0,961538462 | 0,961538462 | 0,961538462 | 0,961538462 | 0,961538462 | 0,961538462 |
| | **2** | | 0,03846154 | 0,036982249 | 0,036982249 | 0,036982249 | 0,036982249 | 0,036982249 | 0,036982249 | 0,036982249 | 0,036982249 |
| | **3** | | | 0,00147929 | 0,001422394 | 0,001422394 | 0,001422394 | 0,001422394 | 0,001422394 | 0,001422394 | 0,001422394 |
| *Size of the* | **4** | | | | 5,68958E-05 | 5,47075E-05 | 5,47075E-05 | 5,47075E-05 | 5,47075E-05 | 5,47075E-05 | 5,47075E-05 |
| *string* | **5** | | | | | 2,1883E-06 | 2,10413E-06 | 2,10413E-06 | 2,10413E-06 | 2,10413E-06 | 2,10413E-06 |
| | **6** | | | | | | 8,41653E-08 | 8,09282E-08 | 8,09282E-08 | 8,09282E-08 | 8,09282E-08 |
| | **7** | | | | | | | 3,23713E-09 | 3,11262E-09 | 3,11262E-09 | 3,11262E-09 |
| | **8** | | | | | | | | 1,24505E-10 | 1,19716E-10 | 1,19716E-10 |
| | **9** | | | | | | | | | 4,78861E-12 | 4,60444E-12 |
| | **10** | | | | | | | | | | 1,84186E-13 |
| | AVERAGE | 1 | 1,03846154 | 1,039940828 | 1,039997724 | 1,039999912 | 1,039999997 | 1,04 | 1,04 | 1,04 | 1,04 |
| | | | | | | | | | | | |
| | | **Genetic alphabet ( 4 nitrogenous bases)** | | | *Step* | | | | | | |
| | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| | **1** | 1 | 0,75 | 0,75 | 0,75 | 0,75 | 0,75 | 0,75 | 0,75 | 0,75 | 0,75 |
| | **2** | | 0,25 | 0,1875 | 0,1875 | 0,1875 | 0,1875 | 0,1875 | 0,1875 | 0,1875 | 0,1875 |
| | **3** | | | 0,0625 | 0,046875 | 0,046875 | 0,046875 | 0,046875 | 0,046875 | 0,046875 | 0,046875 |
| *Size of the* | **4** | | | | 0,015625 | 0,01171875 | 0,01171875 | 0,01171875 | 0,01171875 | 0,01171875 | 0,01171875 |
| *string* | **5** | | | | | 0,00390625 | 0,002929688 | 0,002929688 | 0,002929688 | 0,002929688 | 0,002929688 |
| | **6** | | | | | | 0,000976563 | 0,000732422 | 0,000732422 | 0,000732422 | 0,000732422 |
| | **7** | | | | | | | 0,000244141 | 0,000183105 | 0,000183105 | 0,000183105 |
| | **8** | | | | | | | | 6,10352E-05 | 4,57764E-05 | 3,43323E-05 |
| | **9** | | | | | | | | | 1,52588E-05 | 1,14441E-05 |
| | **10** | | | | | | | | | | 1,52588E-05 |
| | AVERAGE | 1 | 1,25 | 1,3125 | 1,328125 | 1,33203125 | 1,333007813 | 1,333251953 | 1,333312988 | 1,333328247 | 1,33335495 |
| | | | | | | | | | | | |
| | | **Binary system ( two states)** | | | *Step* | | | | | | |
| | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| | **1** | 1 | 0,5 | 0,5 | 0,5 | 0,5 | 0,5 | 0,5 | 0,5 | 0,5 | 0,5 |
| | **2** | | 0,5 | 0,25 | 0,25 | 0,25 | 0,25 | 0,25 | 0,25 | 0,25 | 0,25 |
| *Size of the* | **3** | | | 0,25 | 0,125 | 0,125 | 0,125 | 0,125 | 0,125 | 0,125 | 0,125 |
| *string* | **4** | | | | 0,125 | 0,0625 | 0,0625 | 0,0625 | 0,0625 | 0,0625 | 0,0625 |
| | **5** | | | | | 0,0625 | 0,03125 | 0,03125 | 0,03125 | 0,03125 | 0,03125 |
| | **6** | | | | | | 0,03125 | 0,015625 | 0,015625 | 0,015625 | 0,015625 |
| | **7** | | | | | | | 0,015625 | 0,0078125 | 0,0078125 | 0,0078125 |
| | **8** | | | | | | | | 0,0078125 | 0,00390625 | 0,00390625 |
| | **9** | | | | | | | | | 0,00390625 | 0,001953125 |
| | **10** | | | | | | | | | | 0,001953125 |
| | AVERAGE | 1 | 1,5 | 1,75 | 1,875 | 1,9375 | 1,96875 | 1,984375 | 1,9921875 | 1,99609375 | 1,998046875 |

Figure 4.2: Probability of ending in each step in the naive algorithm

**Lemma 4.5.1**

$$1 + u + u^2 + ... + u^{(n-2)} + u^{(n-1)} = \frac{1-u^n}{1-u}.$$

**Proof:**

$$1 + u + u^2 + u^3 + u^4 + ... = \frac{1}{1-u}.\text{proved in Chapter 2}$$

If we multiply by $u^n$ we get $u^n + u^{(n+1)} + u^{(n+2)} + ... = \frac{u^n}{1-u}$. Now we subtract the $2^{nd}$ equality from the $1^{st}$ getting:

$$1 + u + u^2 + ... + u^{(n-2)} + u^{(n-1)} = \frac{1-u^n}{1-u}.$$

□

We can continue to develop our sum:

$$\sum_{i=0}^{n-1} u^i = 1 + u + u^2 + u^3 + u^4 + ... + u^{(n-1)} = \frac{1-u^n}{1-u}.$$

Recalling our experience with generating functions we can now take the derivative and multiply by $u$, getting:

$$1u + 2u^2 + 3u^3 + 4u^4 + ... + (n-1)u^{(n-1)} = \frac{u + n \cdot u^{(n+1)} - u^n - n \cdot u^{(n-1)}}{(1-u)^2}$$

We now substitute $u = 1/L$ and multiply by (L-1) getting:

$$\frac{L}{L-1} - \frac{L^{(1-n)} \cdot (Ln - n + 1)}{L-1}$$

This rearrangement is not a coincidence showing the evolution we see in graph of figure 4.3. Fixing L constant the first term is clearly constant and the second one goes to 0 because exponential grows much faster than linear. Note also that since L>0 the second term of the formula will always be positive and therefore we can fix the upperbound at $\frac{L}{L-1}$ as the graph in Figure 4.3 also points out.

This means for example that the average case of a binary word will never be greater than 2, in the genetic code it will never be greater than $\frac{4}{3}$ etc. This is a pretty amazing result because one would expect the average number of steps to be very large when the words have infinitely many characters.

Therefore the average case is:$\Theta(\frac{|\Sigma|}{|\Sigma|-1}n) \rightarrow O(2n) = O(n)$.

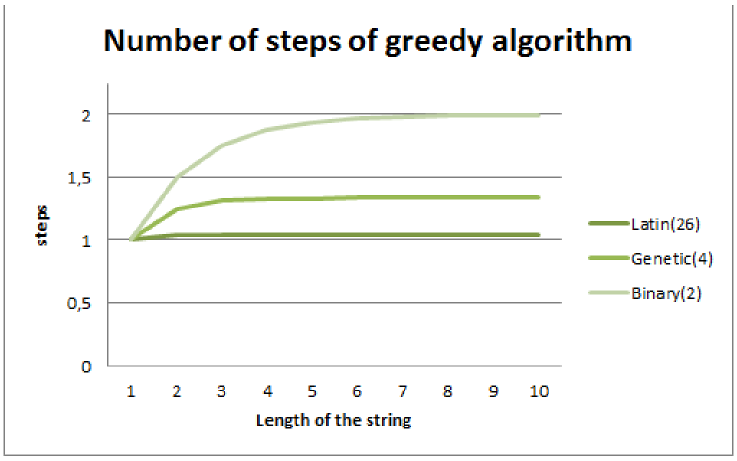


Figure 4.3: Average number of steps of the naive algorithm

With this we have seen that the naive algorithm works amazingly well in the average case, but at the same time runs very slow in the worst case. Let us make an example of that:

### 4.5.3 Examples and particular cases

**Running the algorithm with pattern $P = 0001$ and text $T = 000010001010001$.**

1. With shift 0 we make 4 comparisons to determine that the pattern is invalid (1 vs 0).
2. With shift 1 we make 4 comparisons to determine that the pattern is valid.
3. With shift 2 we make 3 comparisons to determine that the pattern is invalid (0 vs 1).
4. With shift 3 we make 2 comparisons to determine that the pattern is invalid (0 vs 1).
5. With shift 4 we make 1 comparison to determine that the pattern is invalid (0 vs 1).
6. With shift 5 we make 4 comparisons to determine that the pattern is valid.
7. With shift 6 we make 3 comparisons to determine that the pattern is invalid (0 vs 1).
8. With shift 7 we make 2 comparisons to determine that the pattern is invalid (0 vs 1).
9. With shift 8 we make 1 comparison to determine that the pattern is invalid (0 vs 1).
10. With shift 9 we make 2 comparisons to determine that the pattern is invalid (0 vs 1).
11. With shift 10 we make 1 comparison to determine that the pattern is invalid (0 vs 1).
12. With shift 11 we make 4 comparisons to determine that the pattern is valid.

In total we have made 31 comparisons.

**Supposing that all characters in the pattern *P* are different we can accelerate the naive algorithm to be linear**

Suppose we have a coincidence in the first $(c-1)$ characters of P and then we get a mismatch in the $c^{th}$ character. Then any shift from 1 to $c-1$ has to be invalid since any of those characters have to be different from the first one of the pattern. Therefore we can start directly searching for shifts in the character that has produced the mismatch. Thus, the worst case is to have all the characters in the text equal to the first character in the pattern giving a complexity of $O(2n) = O(n)$.

Finally, we should add that in case of getting a valid shift we should just continue with the next character after the end of the shift instead of continuing with the second character of the shift.

**Gap character:** $\diamond$

A very important way of search is with the gap character, which allows a separation between the parts that separates. Take for example pattern $P =$Obama$\diamond$ McCain and the text $T =$*Obama won the 2008 elections against senator McCain.* This would give a valid shift since we have the word *Obama*, then a separation ( that can be equal to $\varepsilon$ ) and then the word *McCain.* The utility of this type of search is immense.

In fact, we can get a polynomic algorithm to find patterns containing $\diamond$ using the naive algorithm. Note again the importance of finding a polynomic algorithm instead of an exponential one.

Let us take the pattern $P_1 \diamond P_2 \diamond \cdots \diamond P_t$ of length n.

We look at the first appearance of $P_1$ with the naive algorithm, if we do not find it we are done as it clearly implies that we cannot find the total text. If we do find it we use again the naive algorithm to look for pattern $P_2$ after the end of the first appearance of pattern $P_1$ and we do the same procedure.

Example:
$P =$AAA$\diamond$BBB$\diamond$ABA
$T =$BAAABABBBABABB
We can find AAA: B**AAA**BABBBABABB, now we can start looking for BBB in *BABBBABABB* and we find it: BA**BBB**ABABB. Finally we look for ABA in **ABA**BB, so we have found the pattern P. The complexity of this algorithm is $O(nm)$ too, as the normal naive algorithm. This is because at most we look at n shifts ( in case we do not find the pattern ) and in each shift we do at most m comparisons, depending on the length of the word, which is limited by the length of the pattern, m. We proved this to be $O(n^2)$ and therefore we have found a polynomic algorithm. As we will later see we can make improvements to this algorithm.

## 4.6 Rabin-Karp algorithm

### 4.6.1 Explanation of the algorithm

From now on, the other three algorithms that we will analyze use a preprocessing algorithm of a relatively small cost to decrease dramatically the cost of the search.

This algorithm is based in basic notions of number theory, basically if $a \neq b$ mod m $\rightarrow a \neq b$. As you can see this algorithm is based in properties of the naturals, so most of our examples will be done with this, however it is very important to notice that we could eventually make some easy translation from signs from an alphabet to numbers and it would work as well, as long as this conversion is always the same.

Let us start without doing modular arithmetic and simply adding the numerical values of the pattern:
Define s as:

$$s = \sum_{i=1}^{m} m^{m-i} P[i].$$

This formula is just the interpretation of the string of signs in base m. For example if $\Sigma = 0, 1, 2...9$ and $P = 168103$, then the pattern would just simply be interpreted as the base 10 number 168103. We can also calculate s' for a substring of the text:
Define s' as

$$s' = \sum_{i=1}^{m} m^{m-i} P[i+m].$$

It is clear that $P[1...m] = T[k+1...k+m] \Rightarrow s = s_k$ so we can also get: $s_k \neq s \Rightarrow P[1...m] \neq T[k+1...k+m]$. In fact we have:

$$P[1...m] = T[k+1...k+m] \Longleftrightarrow s = s_k.$$

Now, the important part is that we can in fact calculate $s_k$ in $O(1)$ as:

$$s_{k+1} = m(s_k - 10^{m-1}T[k+1]) + T[s+m+1].$$

This can prove very useful with small patterns and small alphabets, such the binary alphabet, but you have to take into account that when patterns start getting big:

1. You can have an overflow: you need to store a value that is too big to store in a variable
2. Comparing numerical values can be as costly as comparing the sequences

Now we can apply modular arithmetic and considering the values mod a certain value w. Now we have:

$$P[1...m] = T[k+1...k+m] \Rightarrow s \equiv s_k \text{ mod w}.$$

Since that the two sequences are equal implies that their values are equal and this does imply that they are equivalent mod any w. However the fact that two values are equivalent mod w does not imply that they are equal, so it does not imply that the sequences are equal. However, we still know that if $s_k \neq s$ then $P[1...m] \neq T[k+1...k+m]$; a fact used by the algorithm.

Note that the operation to get $s_{k+1}$ in $O(1)$ is still valid in mod w.So this congruence mod w will be a fast heuristic to filter the great majority of cases.

Now, we still have to calculate both s and $s_0$ in $\Theta(m)$, which is the preprocessing time.

Then, for the normal search we have that if $s_k = s$ we check if they are equal or not in $\Theta(m)$ by just trying to match each character. A shift for which $s_k = s$ but $P[1..m] \neq T[k+1...k+m]$ is called spurious hit; we want to have the minimum of those to approach the linear complexity ( since if we do not have to check spurious hits we need only $O(1)$ to check every shift). For this we want the biggest possible q since the probability of have a spurious hit is approximately $\frac{1}{q}$. However, note that for very big q's we may have overflows or comparisons that are so big that cannot be considered constant.

In this case we usually take $q$ ti be a prime number such that $|\Sigma| \cdot q$ just fits within one computer word, allowing all necessary computations do be performed. For example if $|\Sigma| = 26$ the limit is $2^{31} - 1$ then $q = max\{$p: p is prime and p $< \lfloor \frac{2^{31}-1}{26} \rfloor\} = 82.595.483$ giving a probability of a spurious hit of $\frac{1}{82595483} = 0.00000121\%$. As you can see in this ordinary example the probability of getting a spurious hit can be, and usually is, *very* small.

### 4.6.2 Pseudocode and complexity

For pattern P, text T, $|\Sigma| = d$, working modulo $q$ we have: With the pseudocode it is quite easy to follow the procedure and determine the best, worst and average case. To begin with, the preprocessing time is always: $O(m)$ since we do 2 times a for of m constant operations.
Then the processing algorithm:
**Best case**: in all cases the heuristic gives a negative $\rightarrow O(n-m)$.
**Worst case**: in all cases the heuristic gives a positive (fake or real) $\rightarrow O((n-m+1)m)$.
**Average case**: the number of positives is equal to the *real* positives where there is a shift of the pattern and the spurious hits that appear with the frequency $\frac{1}{q}$. Therefore: $O(m(v + \frac{n-v}{q}))$ where $v$ is the number of matchings. This gives a total complexity of:
**Best case**: $O(m) + O(n-m) = O(n)$
**Worst case**: $O(m) + O((n-m+1)m) = O(nm)$
**Average case**: $O(n) + O(m(v + n/q))$

**Algorithm 3** Rabin-Karp Algorithm

```
$n \leftarrow length[T]$
$n \leftarrow length[P]$
$h \leftarrow d^{m-1}$ mod q
$p \leftarrow 0$
$s_0 \leftarrow 0$
for $i = 1$ to m do
  $p \leftarrow (d \cdot p + P[i])$ mod q
  $t_0 \leftarrow (d \cdot s_0 + T[i])$ mod q
end for
for $s = 0$ to $n - m$ do
  if $P = t_s$ then
    if $P[1...m] = T[s+1...s+m]$ then
      print "Pattern with shift" s
    end if
  end if
  if $s < n - m$ then
    $s_k + 1 \leftarrow (d(t_s - T[s+1] \cdot h) + T[s+m+1])$ mod q
  end if
end for
```

### 4.6.3 Problems

#### Example with pattern P=26, T=3141592653589793 and q=11

*This example is taken from [2][p. 915] because the author find it interesting to do searches using the famous number $\pi$*

First: 26 mod 11 = 4. Now:

- 31 mod 11 = 9 $\rightarrow$ nothing
- 14 mod 11 = 3 $\rightarrow$ nothing
- 41 mod 11 = 8 $\rightarrow$ nothing
- 15 mod 11 = 4 $\rightarrow$ **spurious hit**

- 59 mod 11 = 4 $\rightarrow$ **spurious hit**
- 92 mod 11 = 4 $\rightarrow$ **spurious hit**
- 26 mod 11 = 4 $\rightarrow$ ***found***
- 65 mod 11 = 10 $\rightarrow$ nothing
- 53 mod 11 = 9 $\rightarrow$ nothing
- 35 mod 11 = 2 $\rightarrow$ nothing
- 58 mod 11 = 3 $\rightarrow$ nothing
- 89 mod 11 = 1 $\rightarrow$ nothing
- 97 mod 11 = 9 $\rightarrow$ nothing
- 79 mod 11 = 2 $\rightarrow$ nothing
- 93 mod 11 = 5 $\rightarrow$ nothing

In total we got 3 spurious hits and 1 found.

### Rabin-Karp for more than one pattern

First let us make the simplification that all the patterns have the same length m. Then, the algorithm is quite simple: with one word we only checked if the last m characters had the same value mod q than the pattern we were looking for. If we are looking for more than one pattern but they are all of the same size we can do the same and check at each step all the patterns that match the value mod q of the last m digits.

Now, without this simplification the problem gets tougher: we would have to take account of the value of the last $m_i$ digits for every i. In fact, this is one possible solution but there is another one which, although it does not change the time complexity much, overcomes the necessity of having multiple counters giving a much more simple and elegant program.

We can do a preprocessing algorithm that stores into a vector the value of the first i digits mod q in v[i]. Then $v[0] = T[0]$ mod q and $v[i] = (|\Sigma| v[i-1] + T[i])$mod q for $i \geqslant 1$. It is also convenient to calculate another vector w where each w[i] is $|\Sigma|^{m_i}$ mod q for every $m_i$, each calculated in $O(lgm_i)$.

Now, for each pattern of size $m_i$ one can calculate in constant time the last $m_i$ digits ending in position j by doing $v[j] - v[j-m_i] \cdot b^{m_i - 1}$ mod q. Apart from this, the rest of the algorithm is the same as the one that has the simplification of equal sizes.

# Chapter 5

# String matching with finite automata

Still having a quadratic upper-bound, we would like to have an algorithm with lineal upper-bound. In this chapter we will introduce and prove an algorithm with such a bound: finite automata. After doing so we will create some personal variations of this algorithm to find other types of patterns apart from the conventional ones. This algorithm will have a high importance later in the paper by helping to solve our main mathematic problem.

## 5.1 Description of an automaton

As described in [2, p. 916] a **finite automata is** a 5-tuple (Q,q$_0$,$A$,Σ,$\delta$), where:

- Q is a finite set of ***states***,
- q$_0$ $\in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- $\delta$ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.

Then, the automaton begins in state $q_0$ and while reading the input string one character at a time will also move through the states. Specifically, if the automaton is at state q and it receives character a it will move ( 'make a transition' ) from state q to state $\delta(q, a)$. Moreover, whenever the current state of the automaton is a member of A, we will say that the automaton has ***accepted*** the string read so far. When an input is not accepted is said to be rejected.

Take for example, consider the following automaton in Figure 5.1 for automaton M; there, string "000" would end in state B and therefore would be accepted whereas string "010" would end in state A and therefore it would not be accepted. In fact, note that this particular automaton accepts strings with an odd number of 0, since getting a 1 does not change the state but 0 does.

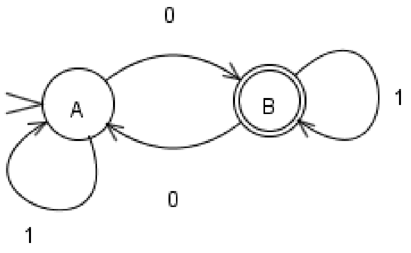


Figure 5.1: Automaton that accepts words with an odd number of zeros

A finite automaton M induces a function $\phi$, called the ***final-state function***, from $\Sigma*$ to Q such that $\phi(w)$ is the state M ends up in after scanning the string w; then we can check if M accepts w by checking if $\phi(w) \in A$. From the definition of our automaton, $\phi$ is defined recursively as:

$$\phi(\varepsilon) = q_0$$

$$\phi(wa) = \delta(\phi(w), a) \text{ for w } \in \Sigma*, \text{ a } \in \Sigma.$$

Therefore, for our two examples, $\phi(000) = B$ and $\phi(010) = A$.

## 5.2 Description of the algorithm

*Note: This explanation is inspired in [2, p. 917]; the author made an effort to provide personal examples and make a summary and simplification of the explanation of the algorithm.*

We can build a string-matching automaton for every pattern $P$, in a preprocessing step to then accelerate the processing time down to $\Theta(n)$; it is very important to understand that the automaton will not depend on the text or word it is looking in but on the pattern it is looking for.

Let us first define an auxiliary function $\sigma$, called the *suffix function* corresponding to $P$ and going from $\Sigma*$ to $0, 1, ..., m$, such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}.$$

Remember that $P_0 = \varepsilon$ and therefore the function is well defined. For example, taking pattern $P = 0101$ we have $\sigma(\varepsilon) = 0, \sigma(00000) = 1, \sigma(11010) = 3$.

In particular, our automaton will be such that it has a start state $q_0$ which will be called 0 and an only accepting state, state m. Furthermore, the transition function $\delta$ is defined such that for any character a and state q:

$$\delta(q, a) = \sigma(P_q a)$$

Which maintains the equality: $\phi(T_i) = \sigma(T_i)$ as proved in Theorem 5.2.4; in plain words this means that after scanning the first $i$ characters of the text T, the machine will be in state $\phi(T_i)$, which, by its definition, is the length of the longest suffix of $T_i$ that is also a prefix of P, our pattern, as we will prove now.

### 5.2.1 Proof of the algorithm

In this case a very simple and beautiful demonstration by induction is presented in [2][p. 921] with some previous lemmas with some modifications made by the author to increase its readability providing slightly different proves for them.

**Lemma 5.2.1** *Suppose that x, y and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leqslant |y|$, then $x \sqsupset y$. If $|x| \geqslant |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.*

**Proof:** Let n be the size of x and let m be the size of y. If $n \leqslant m$ this means that both the last $n$ characters of $x$ and the $n$ last characters of $y$ are equal to the last $n$ characters of $z$. Therefore the last $n$ characters of x are equal to the last $n$ characters of y. Thus by definition: $x \sqsupset y$.

If $n \geqslant m$ this means that both the last $m$ characters of $x$ and the $m$ last characters of $y$ are equal to the last $m$ characters of $z$. Therefore the last $m$ characters of x are equal to the last $m$ characters of y. Thus by definition: $y \sqsupset x$.

If $n = m$ then $n \leqslant m$ ($\Rightarrow x \sqsupset y$) and $n \geqslant m$ ($\Rightarrow y \sqsupset x$). $x \sqsupset y$ and $n \geqslant m$ can only happen at the same time if $x = y$. □

**Lemma 5.2.2** ***For any string x and character a, we have*** $\sigma(xa) \leqslant \sigma(x) + 1$**.**

**Proof:** Suppose that $\sigma(xa) > \sigma(x)+1$. This, by definition implies that the last $\sigma(xa)$ are both a prefix of P and a suffix of $x$. In other words, if n is the size of $x$:

$$T_{n-i+1} = P_{i+1}$$

$$\text{for every } i \leqslant \sigma(xa)$$

Which implies that if we are looking at the penultimate $\sigma(xa)-1$ characters they will also coincide. But by definition of $\sigma(x)$, this cannot happen because only the penultimate $\sigma(x)$ coincide and $\sigma(xa)-1 > \sigma(x)$. Thus:

$$\sigma(xa) \leqslant \sigma(x)+1$$

□

**Lemma 5.2.3** ***For any string*** $x$ ***and character*** $a$***, then if*** $q=\sigma(x)$***, then*** $\sigma(xa)=\sigma(P_q a)$***.***

**Proof:** From the definition of $\sigma$, we have $P_q \sqsupset x$.
Logically, if we add the same character at the end of two words that shared their last q characters, they will now share their last $q+1$ because they still share their $q+1$ last characters except the last by the definition of the words and they share the last one because we have added the same character to both. Therefore $P_q|sqsupsetxa$. Call r $\sigma(xa)$, then $r \leqslant q+1$ by Lemma 5.2.2. Now we have both $P_q a \sqsupset xa, P_r \sqsupset xa$, and $|P_r| = |P_q a|$, which implies $P_r \sqsupset P_q a$ by Lemma 5.2.1.

Therefore $r \leqslant \sigma(P_q a)$, in other words, $\sigma(xa) \leqslant \sigma(P_q a)$. But since $P_q a \sqsupset xa$ we also have $\sigma(P_q a) \leqslant \sigma(xa)$.
Thus, $\sigma(xa) = \sigma(P_q a)$. □

**Theorem 5.2.4** *If $\phi$ is the final-state function of a string-matching automaton for a given pattern P and T[1...n] is an input text for the automaton, then $\phi(T_i) = \sigma(T_i)$ for every $i \leqslant n$.*

**Proof:** First, we check the base case $\phi(T_0) = \phi(\varepsilon) = 0 = \sigma(\varepsilon)$ by definitions of $T_0$ and $\sigma(0)$.

Now, assuming that $\phi(T_i) = \sigma(T_i)$ we want to prove that $\phi(T_{i+1}) = \sigma(T_{i+1})$. Let $q$ denote $\phi(T_i)$, and let $a$ denote $T[i+1]$. Then,

$$\sigma(T_{i+1}) = \phi(T_i a)(\text{by the definition of } T_{i+1} \text{ and } a)$$

$$= \delta(\phi(T_i), a) \text{ (by the definition of } \phi)$$

$$= \delta(q, a) \text{ (by the definition of q)}$$

$$= \sigma(P_q a) \text{ (by the definition of } \delta)$$

$$= \sigma(T_i a) \text{ (by Lemma 5.2.3 and induction)}$$

$$= \sigma(T_{i+1}) \text{ (by the definition of } T_{i+1}) \text{ .}$$

□

### 5.2.2 Illustration and explanation

Now that we have proved the algorithm, we can proceed to explain it less rigorously. As the other algorithms we are looking for the pattern in a particular shift; however, the big difference is that, in case we cannot find it we use the information we have to avoid the necessity of going back and analyze the same substring again.

The states of the automaton can be seen as levels of alert, indicating the maximum prefix of the pattern that coincides with a suffix of the text. Take for example pattern 011, which will later be analyzed mathematically. We have an associated automaton illustrated in Figure 5.2.

*Note: to make it clearer states will be called "a,b,c..." instead of by numbers to avoid the confusion with the numbers (that represent the characters of the pattern ).*
Consider text $T = 0101100$.

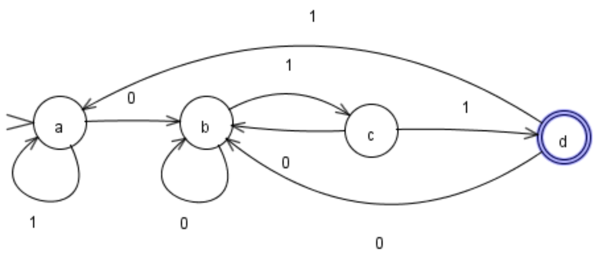


Figure 5.2: Automaton that looks for appearances of pattern 011

1. We start in state a (state $q_0$ ).
2. Since we scan a 0 we move to state b, equivalent to have a coincidence of 1 character,
3. Scanning a 1 we move to state c ( 2 character-coincidence ).
4. Scanning a 0 we do not get back to state a as we did with the naive algorithm but move back only to b.
5. Scanning a 1 we move again to state c.
6. Scanning a 1 we move to state d, which is an acceptor state and we indicate the found.
7. However, we can continue looking for more appearances, we scan a 0 and we move back to b.
8. Finally we scan another 0 and we stay in b, ending in state b.

As one can see, the current state indicates the maximum coincidence we have up to that moment; this information and the next character are the only requirements to know the future state. Using this, we can move up and down the automaton until we discover the pattern. We can also build automata that just tell us if a pattern appears in the text, but not how many times or where. In fact, this type of automata will be the most used in our theoretical and mathematical applications. The only difference with the previous type of automata is that once you get to the accepting state, instead of printing that the automaton found the pattern and continue looking the automaton never goes out from the accepting state. At the end, to look if the pattern appears in the text, we will have to look if the final state is the accepting state. Take for example the same 011 but with a closed end, just to look if a text contains 011 one or more times represented in Figure 5.3.

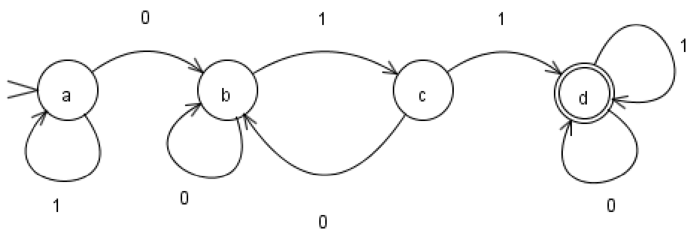


Figure 5.3: Automaton accepting patterns containing pattern 011

## 5.3 Personal variations of the algorithm

Interested in this algorithm, the author created its own variations of it. Some of them were done just for curiosity or the desire to go deeper and create personal algorithms, but they turned out to have a great importance later on in the paper with some unexpected, and yet of high interest, applications.

All in all, those algorithms provide an understandable illustration of this crucial algorithm. Moreover, they are some very good examples because they are much more difficult to create than they are to understand, since the first requires to have a great insight of the algorithm and the latter provides this insight. In this section, we will analyze four different variations of the algorithm, from most simple to more complex.

### 5.3.1 Multiple-possibilities character

This only implies a small modifications of the automaton. If a is the accepted n-th character then we have $\cdot\delta(n-1,a)=n$. Normally we only have this for one character a; however, if we accept more than one character we can let $\delta(n-1,b)$ and $\delta(n-1,c)$ also to be n, etc. This multiple-possibilities character also has to be taken into account when looking for possible suffixes for future states. For example both 011 and 111 are suffixes of 1?11; therefore every time 'O' options are possible we have to divide the automaton into 'O' different automatons because the future transitions will be affected by the character that is put as '?'. For example consider pattern P=0?1?1 and state 2( you already got '0?') if you get a 0 you would go to state 1 if '?'=1 and to state 2 if '?'=0.

Figure 5.4 describes the automaton for this pattern P=0?1??1. This type of automaton are quite complex, for example the change between E01 and E10 when a 0 is scanned: 0010**0** we can take the suffix 0100 which changes the value of both '?' in order to create the largest possible suffix.

Moreover, we can go further in our analysis and just accept some, but not all the characters; this could be done in a similar manner; as illustrated in Figure 5.5 for pattern GA(A/G)C where (A/G) means character A or G.

Finally, note that one can ask for a n-character space between two 'words'. For example if one wants to search for pattern 0011 then a 8-character space and then 1100, this is equivalent to search pattern: $0011??\overset{8}{\ldots}??1100$.

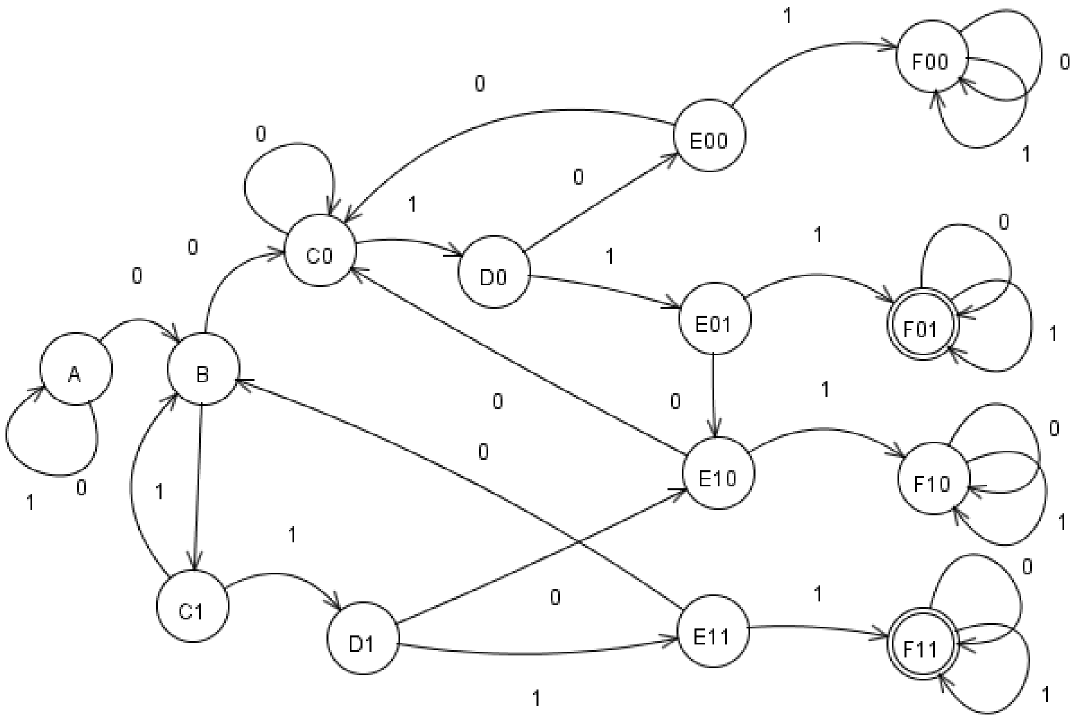


Figure 5.4: Automata accepting words that contain pattern 0?1??1

### 5.3.2 Gap character ⋄

As we did with the naive algorithm, we want to find patterns with the gap character ⋄, which remember that it stands for an undetermined space, including the empty space.

If we analyze it, looking for pattern: $P = P_1 \diamond P_2 \diamond \cdots \diamond P_n$ is equivalent to first look for pattern $P_1$, then from the end of it, look for pattern $P_2$ and so on. We used this to get a polynomial-time algorithm using the naive approach.

In automatons we can do the following:

1. Let P be the pattern we are looking for; for example 00 ⋄ 111 ⋄ 01.
2. Mark all the characters that go before a ⋄: 0**0**⋄11**1**⋄01.
3. Eliminate the diamonds: 0**0**11**1**01.
4. Every marked character will have its state treated as a start state.

It is very important to understand that we take every character before a ⋄ since the end state for $P_i$ is the start state for $P_{i+1}$ since we still have not found any character of $P_{i+1}$.

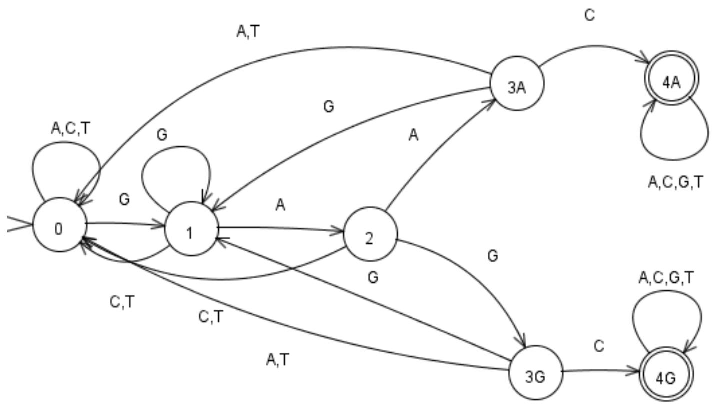


Figure 5.5: Automata accepting words that contain pattern GA(A/G)C

By start state it is meant the only state from which we cannot go back. Looking at the automaton in Figure 5.6 this would represent that state A,C and G are a point of no return: once you get there, the automaton will never go back to previous states than A,C or G respectively. This $\diamond$ character can be very useful to search for an ordered sequence of patterns $P_1, P_2, ..., P_n$ all

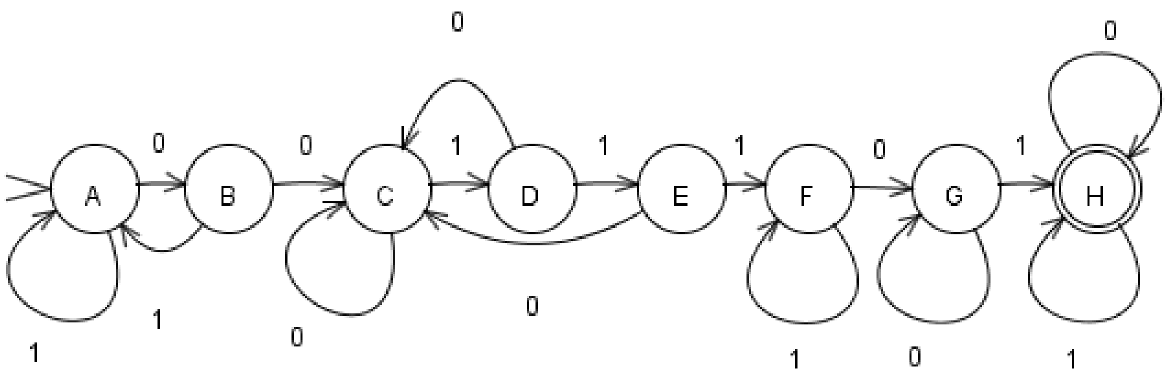


Figure 5.6: Automaton accepting words that contain pattern $00 \diamond 111 \diamond 01$

of size $\leqslant m$ in time $O(\sum |P_i|) = O(n \cdot m)$. Ordered sequence of words appear for example in genes when one is only looking for exons, which are separated by introns of variable size and content.

On the other hand, we can also look in a text for all the words without any particular order, by trying all the different permutations of $P_1, ..., P_n$, which are $n!$ as we saw with permutations without repetition in the first chapter. Thus searching for all the words could be done in $O(n! \cdot n \cdot m)$. However, we will later see how to improve this complexity.

### 5.3.3 Detecting *either* of some patterns

Given a list of patterns $P_1, P_2, ..., P_n$ we would like to create an automaton that finds one of them. A first thought could be to do it with $O(m^n)$ states by having all the possible combinations of states for each pattern such as $0, 0, 0.., 0$ , $0, 0, 0, ..., 3$ or $2, 4, 0, 0, 2, .., 5$. This would be impracticable as $O(m^n)$ is exponential.

Instead, we should notice that lots of possible combinations of states will never happen. For example, if one is looking for 0000 or 1111, one of the states must be 0 at any time. Instead, we can use the same fact we used for the simple algorithm: the longest known suffix and the next character are the only requirements to know the following state. However, now the longest known suffix will be of all the patterns we are looking for. Note that sometimes there might be a tie ( if two words share a prefix ); in this case we can just say arbitrarily that the first word takes the state. Thus, we only have to consider the m possible state for each of the n patterns, which is $O(m \cdot n)$ states.

Let us take the example of patterns $P_1 = 010$, $P_2 = 111$ and $P_3 = 000$ for which we get the automaton in Figure 5.7.

Let us analyze text T=00111. *Each state will be given in two coordinates (word,state in the word).*

1. We start at (1,0)
2. Scanning a 0 we get suffixes of size 1,0,1 respectively; thus the next state will be (1,1).
3. Scanning a 0 we get suffixes of size 1,0,2 respectively; thus the next state will be (3,2).
4. Scanning a 1 we get suffixes of size 2,1,0 respectively; thus the next state will be (1,2).
5. Scanning a 1 we get suffixes of size 0,2,0 respectively; thus the next state will be (2,2).
6. Scanning a 1 we get suffixes of size 0,3,0 respectively; thus the next state will be (2,3).
7. We have arrived at the final state of word 2.

Some uses of this algorithm could be to Google-style searches, where we just want to find one of the patterns from a list or, also in genetics, we want to find the nitrogenous bases that represent an amino acid, since more than one representation is possible, this algorithm also proved useful.

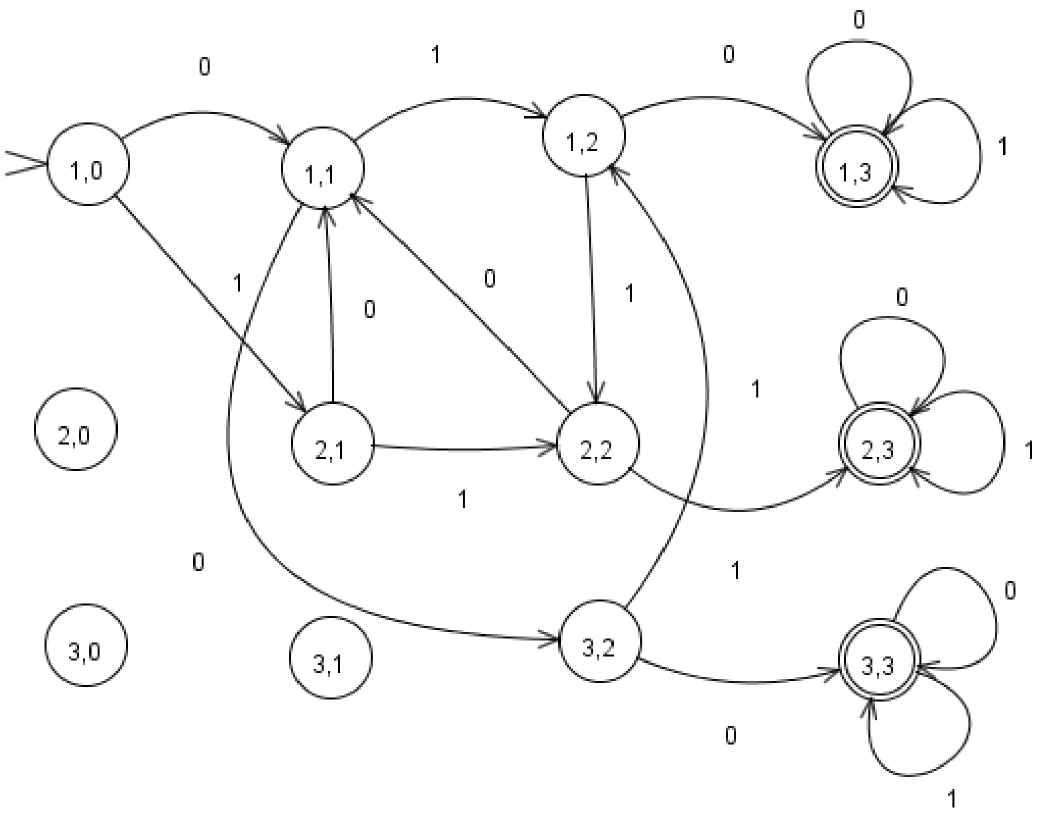


Figure 5.7: Automaton accepting words that contain 010,111 or 000

### 5.3.4 Detecting *all* the patterns

When one is looking for two things and has no idea about where they might be it is usually convenient to start looking for both of them and when one is found, just look for the second one. In general if one is looking for patterns $A_1, A_2, ..., A_n$ it is useful to always look for the subset of $A_i$ which still has not been found.

Take the case of only two words: looking for patterns A and B: we will first look for *either* A or B. Then, if we find A, start looking only for B and, in case we first found B, start looking only for A.

For this we will have an automaton build from three different automatons, the first line looks for A or B, the second one for A and the third one for B.

For example take A=000 and B=111. The automaton build for finding both of them, in any order, is described in Figure 5.8.

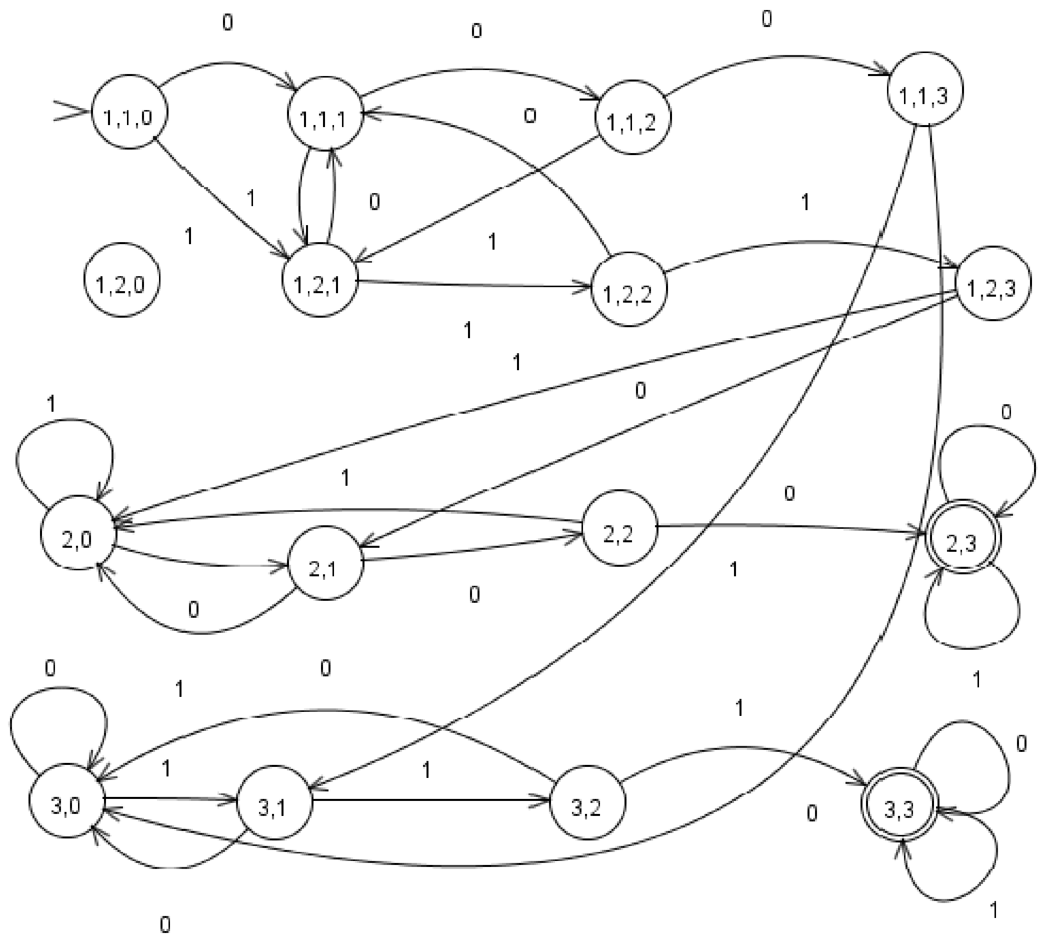


Figure 5.8: Automaton accepting words that contain patterns 000 and 111

In Figure 5.8 we separated graphically line 1.1 and 1.2 because a pattern that looks for *either* of n patterns will have n lines. Therefore line 1.1 refers to pattern 000 and line 1.2 refers to pattern 111 but together they work as 'sub-automaton' to find *either* of the two patterns.

Let us analyze the number of states when looking if a word contains *all* of n patterns. Let S be the set of patterns $P_1, P_2, ..., P_n$ we are looking for, all of sizes $\leqslant m$. First of all, remember that we will have one "*either*" automaton for each subset and since we have $2^n$ possible subsets we will have $2^n$ "*either*" automata, but they will be of different sizes; therefore we have to go deeper in our analysis.

As we described in the previous subsection to find either of n patterns of sizes $\leqslant m$ we will have an automaton of at most $O(n \cdot m)$ states. We know we have $\binom{n}{i}$ automata looking for *either* of i patterns, because we are choosing i patterns from n, as we showed in the first chapter. Therefore the total number of states should be:

$$\sum_{i=0}^{n} \binom{n}{i} \cdot i \cdot m$$

We can factor out the m getting:

$$m \cdot \sum_{i=0}^{n} \binom{n}{i} \cdot i$$

Which as proved in Theorem 5.3.1 is:

$$2^{n-1} \cdot n \cdot m.$$

This complexity is much faster than the one proposed using gap characters as one can see in Figure 5.9 which fixes m constant.

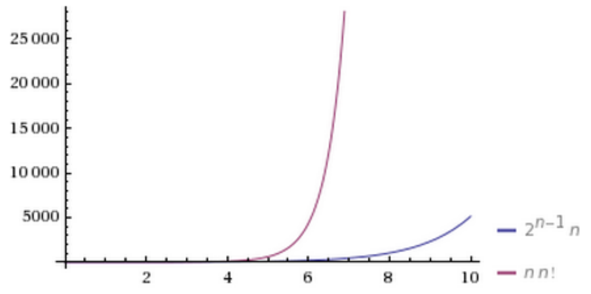


Figure 5.9: Comparison of $2^n$ with $n!$

**Theorem 5.3.1** $\sum_{i=0}^{n} \binom{n}{i} \cdot i = 2^{n-1} \cdot n$

**Proof:** Since $\binom{n}{i} = \binom{n-i}{i}$ we can factor group each $i$ with its complementary so that they add up to $n$. Which: If n is odd we have:

$$\sum_{i=0}^{n} \binom{n}{i} \cdot i = \sum_{i=0}^{n/2} \binom{n}{i} \cdot (i + (n-i)),$$

$$n \cdot \sum_{i=0}^{n/2} \binom{n}{i}.$$

Knowing $\sum_{i=0}^{n}\binom{n}{i} = 2^n$ (Theorem 1.2.1 ) and that $\binom{n}{i} = \binom{n-i}{i}$ we have that our sum must be half of $2^n$ and thus:

$$n \cdot \sum_{i=0}^{n/2} \binom{n}{i} = n \cdot 2^{n-1}.$$

$\square$

## 5.4 Chapter remarks

All those diverse algorithms provide an enormous diversity of tools to find different types of patterns or combinations of patterns. Starting by the basic algorithm we changed it to a combination of flexible algorithms ultimately capable of determining together if a text contains "(A or B) and ( C or D ) $\diamond$ (E or F)" given patterns A-F.

On the other hand, one could think that it is useless to create a unique automaton to determine such complex combinations; for example, to find if a text contains patterns A or B, instead of creating a single complex automaton we could create two standard automata, one for each pattern, and see if both find their pattern. However it is of high importance to have a single automaton to determine these combinations. Having a single automaton is needed to perform a mathematical analysis that will be seen in the next chapter, where we connect generating functions with all those automata algorithms.

# Chapter 6

# Connecting Generating Functions and Automata

*Pure mathematics is, in its way, the poetry of logical ideas.* **Albert Einstein**

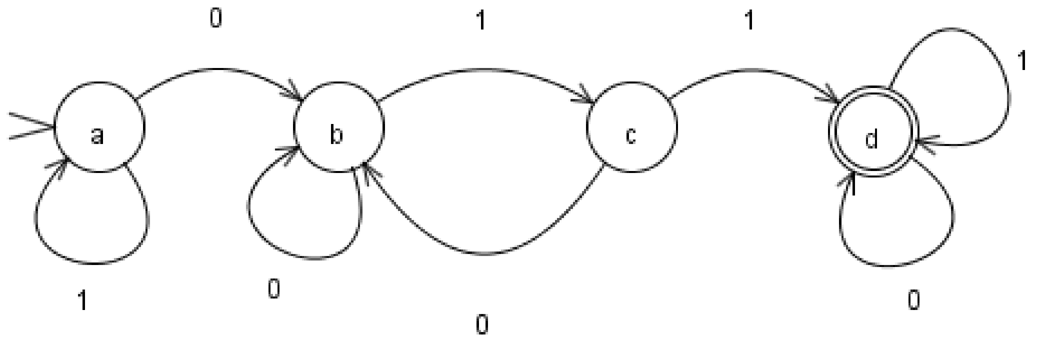
Once solved our computer science problem, let us get back to the big mathematical question we had at the beginning:

**Given an alphabet and a given size, how many words are there with a given pattern of that size?**

We first tried with the basic formulas, it did not work. Then we tried with recurrences and it did not work either. Generating functions along with the symbolic method seemed a good approach, but we could not finalize it. Once again, one part of the study helps the other; in this case the algorithm gave us some insight about the mathematical part. Let us show it with an example.

## 6.1 A first example

We want to know the number of words of each size that contain the pattern 011. To do it we start with the computer science approach, drawing the automat, as we described it in a previous chapter.



Now, remember that asking the question how many words of a given size have pattern $'011'$ is *exactly* the same question as how many words, when passed through this automat will end in state d.

Before asking this question, let us ask first:

*Suppose they start in state d, how many words will end up in state d?*

This is not a very complicated question, the empty word will certainly be there as not doing anything will certainly guarantee to stay in the same state. Now let us look all the possibilities:

- The empty word
- Words that start with a 0 ( thus stay in state d ) and then go from state d to state d.
- Words that start with a 1 ( thus stay in state d ) and then go from state d to state d.

Now, let us do it for words starting from state c. We still want to know how many words starting in state c will get to state d. Now note that you cannot get to state d with the empty word.Let us, once again, analyze all the possibilities:

- Words that start with a 0 ( thus go to state b ) and then go from state b to state d.
- Words that start with a 1 ( thus go to state d ) and then go from state d to state d.

We can do the same for state b:

- Words that start with a 0 ( thus stay in state b ) and then go from state b to state d.
- Words that start with a 1 ( thus go to state c ) and then go from state c to state d.

And for state a:

- Words that start with a 0 ( thus go to state b ) and then go from state b to state d.
- Words that start with a 1 ( thus stay in state a ) and then go from state a to state d.

We can describe this formally:

$$\begin{aligned}
\mathcal{L}_a &= 0\mathcal{L}_b \cup 1\mathcal{L}_a, \\
\mathcal{L}_b &= 0\mathcal{L}_b \cup 1\mathcal{L}_c, \\
\mathcal{L}_c &= 0\mathcal{L}_b \cup 1\mathcal{L}_d, \\
\mathcal{L}_d &= 0\mathcal{L}_d \cup 1\mathcal{L}_d \cup \{\epsilon\},
\end{aligned}$$

And now using the symbolic method transform it to equations with generating functions:

$$\begin{aligned}
L_a(z) &= zL_b(z) + zL_a(z) \\
L_b(z) &= zL_b(z) + zL_c(z) \\
L_c(z) &= zL_b(z) + zL_d(z) \\
L_d(z) &= zL_d(z) + zL_d(z) + 1,
\end{aligned}$$

We can rewrite this using matrices:

$$\begin{pmatrix} L_a(z) \\ L_b(z) \\ L_c(z) \\ L_d(z) \end{pmatrix} = \begin{pmatrix} z & z & 0 & 0 \\ 0 & z & z & 0 \\ 0 & z & 0 & z \\ 0 & 0 & 0 & 2z \end{pmatrix} \begin{pmatrix} L_a(z) \\ L_b(z) \\ L_c(z) \\ L_d(z) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Now let us define a more compacted definition for each matrix:

$$I = \begin{pmatrix} L_a(z) \\ L_b(z) \\ L_c(z) \\ L_d(z) \end{pmatrix}$$

$$T = \begin{pmatrix} z & z & 0 & 0 \\ 0 & z & z & 0 \\ 0 & z & 0 & z \\ 0 & 0 & 0 & 2z \end{pmatrix}$$

$$v = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

With all this, the matrix equation can be rewritten to:

$$I = T \cdot I + v$$

$$v = I - T \cdot I$$

Factor out '$I$' and let $Id$ be the identity matrix of size 4.

$$(Id - T)I = v$$

We rewrite it in the expanded form:

$$\begin{pmatrix} 1-z & -z & 0 & 0 \\ 0 & 1-z & -z & 0 \\ 0 & -z & 1 & -z \\ 0 & 0 & 0 & 1-2z \end{pmatrix} \begin{pmatrix} L_a(z) \\ L_b(z) \\ L_c(z) \\ L_d(z) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Now, note that this is a system of 4 equations and 4 unknowns and that those equations are independent because each have a '1' in different position. Therefore we will get a single solution.

We can solve the system by using the Gaussian Elimination we learn in school, with the difficulty of being a system involving functions and not real numbers. Let us show the procedure ( for an easier following, changes are in **bold** ):

$$\left(\begin{array}{cccc|c} 1-z & -z & 0 & 0 & 0 \\ 0 & 1-z & -z & 0 & 0 \\ 0 & -z & 1 & -z & 0 \\ 0 & 0 & 0 & 1-2z & 1 \end{array}\right)$$

We put the first non-zero coefficients of the first two equations to 1.

$$\left(\begin{array}{cccc|c} \boldsymbol{1} & \boldsymbol{\frac{-z}{1-z}} & 0 & 0 & 0 \\ 0 & \boldsymbol{1} & \boldsymbol{\frac{-z}{1-z}} & 0 & 0 \\ 0 & -z & 1 & -z & 0 \\ 0 & 0 & 0 & 1-2z & 1 \end{array}\right)$$

We clear out the second column.

$$\left(\begin{array}{cccc|c} 1 & \boldsymbol{0} & \boldsymbol{\frac{-z^2}{(1-z)^2}} & 0 & 0 \\ 0 & 1 & \frac{-z}{1-z} & 0 & 0 \\ 0 & \boldsymbol{0} & \boldsymbol{\frac{1-z-z^2}{1-z}} & -z & 0 \\ 0 & 0 & 0 & 1-2z & 1 \end{array}\right)$$

We put the first non-zero coefficient of the third row to 1.

$$\left(\begin{array}{cccc|c} 1 & 0 & \frac{-z^2}{(1-z)^2} & 0 & 0 \\ 0 & 1 & \frac{-z}{1-z} & 0 & 0 \\ 0 & 0 & \boldsymbol{1} & \boldsymbol{\frac{-z(1-z)}{1-z-z^2}} & 0 \\ 0 & 0 & 0 & 1-2z & 1 \end{array}\right)$$

We clear out the third column.

$$\left(\begin{array}{cccc|c} 1 & 0 & \boldsymbol{0} & \boldsymbol{\frac{-z^3}{(1-z)(1-z-z^2)}} & 0 \\ 0 & 1 & \boldsymbol{0} & \boldsymbol{\frac{-z^2}{1-z-z^2}} & 0 \\ 0 & 0 & 1 & \frac{-z(1-z)}{1-z-z^2} & 0 \\ 0 & 0 & 0 & 1-2z & 1 \end{array}\right)$$

We put the first non-zero coefficient of the last row to 1.

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & \frac{-z^3}{(1-z)(1-z-z^2)} & 0 \\ 0 & 1 & 0 & \frac{-z^2}{1-z-z^2} & 0 \\ 0 & 0 & 1 & \frac{-z(1-z)}{1-z-z^2} & 0 \\ 0 & 0 & 0 & \boldsymbol{1} & \boldsymbol{\frac{1}{1-2z}} \end{array}\right)$$

We clear out the last column and we finally get the solutions.

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & \mathbf{0} & \boldsymbol{\frac{z^3}{(1-z)(1-2z)(1-z-z^2)}} \\ 0 & 1 & 0 & \mathbf{0} & \boldsymbol{\frac{z^2}{(1-2z)(1-z-z^2)}} \\ 0 & 0 & 1 & \mathbf{0} & \boldsymbol{\frac{z(1-z)}{(1-z-z^2)(1-2z)}} \\ 0 & 0 & 0 & 1 & \frac{1}{1-2z} \end{array}\right)$$

In other words:

$$L_a = \frac{z^3}{(1-z)(1-2z)(1-z-z^2)}$$
$$L_b = \frac{z^2}{(1-2z)(1-z-z^2)}$$
$$L_c = \frac{z(1-z)}{(1-z-z^2)(1-2z)}$$
$$L_d = \frac{1}{1-2z}$$

Now note that we are mainly interested in $L_a$ because we will make the automata start at a, it would not make sense otherwise. To better analyze $L_a$ we can decompose it in partial fractions:

$$L_a = \frac{1}{1-z} + \frac{1}{1-2z} - \frac{z+2}{1-z-z^2}$$

Here we recognize the first two generating functions:

$$\frac{1}{1-z} = \sum_{k=0}^{\infty} z^k$$

$$\frac{1}{1-2z} = \sum_{k=0}^{\infty} 2^k z^k$$

We only have to analyze the last one:

$$\frac{z+2}{1-z-z^2} = 2 + 3z + 5z^2 + 8z^3 + ... = \sum_{k=0}^{\infty} F_{k+3} z^k$$

In fact, note that we do recognize the denominator as the characteristic polynomial found in the first chapter.

So in general we now know that for a given size n, the number of binary words containing "011" is:

$$2^n + 1 - F_{n+3}.$$

Finally note that $F_n$ as n $\to \infty$ grows as $\frac{\phi^n}{\sqrt{5}}$ (Proved in Chapter 1) and since $\phi < 2$ we know that:

$$\frac{2^n + 1 - F_{n+3}}{2^n} \text{ as n} \to \infty = 1$$

And that:

$$2^n + 1 - F_{n+3} \text{ as n} \to \infty = 2^n - \frac{\phi^n}{\sqrt{5}}.$$

This is pretty logical since the probability of not having any 011 in an infinite sequence must be 0.

## 6.2 Generalization

The procedure we did with our example of 011 can always be applied to any pair of pattern and alphabet ( given than the pattern is written in the alphabet ). We could get the generating function associated to another binary words such as '000' ($\frac{1}{1-2z} - \frac{z^2+z+1}{1-z-z^2-z^3}$ )or to the codon AUG, methionine ($\frac{1}{1-4z} - \frac{1}{1-4z+z^3}$).

We always have to do the same procedure:

1. Build the automata as described in the previous chapter
2. Get the matrix from the automata
3. Solve the system of equations using Gaussian Elimination
4. Separate into partial fractions
5. Try to get a formula from the partial fraction sum

The first point has already been shown, let us analyze the second one:

**Getting the matrix from the automata**

Start with the identity matrix. After this, take the automata that detects if there is a pattern in a text (Important: we must not take the automata that detects *each* appearance). For each arrow in the automata we subtract a 'z' in the position : row= first state, column= second state. For example if we have an arrow going from state A to B we will subtract 'z' to the position (1st row, 2nd column).

**Solve the system of equations using Gaussian Elimination**

Gaussian Elimination consists in the following algorithm:
For each row:

1. Divide all the coefficients so that the first non-zero term in the row is 1.

2. Subtract the necessary quantities from the others rows so that all the terms in the column where our row start, the other rows have a 0.

This ensures that at the end we will have the extended matrix with the identity matrix left and the solutions at the right column.

**Separate into partial fractions**

Now we have the simple form of the generating function. However, to understand it better and occasionally get a closed formula it is very helpful to have it in partial fractions. Partial fraction means the following:

$$\text{Given a fraction:}\frac{P(x)}{D_1(x)...D_n(x)},$$
$$\text{we want to get: } \frac{P_1(x)}{D_1(x)} + ... + \frac{P_n(x)}{D_n(x)} \text{ where } P_1, ..., P_n \text{ are polynomials.}$$

It is very important to note the difference with the normal partial fraction derivation, where each $P_i(x)$ is irreducible in $\mathbb{Q}[x]$, and therefore a linear, a power of a linear, or an irreducible quadratic polynomial. However in $\mathbb{Z}[x]$ a polynomial can be irreducible in any degree, not just in 1 or 2.
To find the partial fractions, we can once again solve a system of equations, because we have one equation for each degree of P(x). Since $deg(P(x)) + 1$ (there is also the constant term ) is the same as the number of fractions, we can always solve the system.

The logic of these equations is the following: imagine you already have the partial fraction decomposition and you want to return to the initial fraction. The contribution of $P_1(x)$ to the numerator would then be:

$$P_1(x) \cdot D_2(x) \dots D_n(x)$$

The contribution of $P_2(x)$ would be:

$$P_2(x) \cdot D_1(x) \cdot D_3(x) \cdot D_4(x) \dots D_n(x)$$

And so on. Counting the contribution to each degree that every $P_i(x)$ does we have this system of equations, which we can also solve using Gaussian Elimination.

**Try to get a formula from the partial fraction sum** Unless the previous ones, there is not a clear perfect mathematical approach to this one. The only way to get the formula is by trying to identify each of the fractions as some known generating function with a known formula and then add up all the formulas.

| | | Size of the alphabet | | | |
|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 |
| Pattern | 0 | $\frac{z}{1-z}$ | $\frac{z}{(1-2z)(1-z)}$ | $\frac{z}{(1-3z)(1-2z)}$ | $\frac{z}{(1-4z)(1-3z)}$ |
| | 00 | $\frac{z^2}{1-z}$ | $\frac{z^2}{(1-2z)(1-z-z^2)}$ | $\frac{z^2}{(1-3z)(1-2z-2z^2)}$ | $\frac{z^2}{(1-4z)(1-3z-3z^2)}$ |
| | 000 | $\frac{z^3}{1-z}$ | $\frac{z^3}{(1-2z)(1-z-z^2-z^3)}$ | $\frac{z^3}{(1-3z)(1-2z-2z^2-2z^3)}$ | $\frac{z^4}{(1-4z)(1-3z-3z^2-3z^3)}$ |
| | 0000 | $\frac{z^4}{1-z}$ | $\frac{z^4}{(1-2z)(1-z-z^2-z^3-z^4)}$ | $\frac{z^4}{(1-3z)(1-2z-2z^2-2z^3-2z^4)}$ | $\frac{z^4}{(1-4z)(1-3z-3z^2-3z^3-3z^4)}$ |

Table 6.1: Generating functions for pattern $00 \overset{n}{\ldots} 00$

## 6.3 Some particular examples

Applying this algorithm we can find the associated generating functions for many patterns.

**Pattern** $00 \overset{n}{\ldots} 00$ **in alphabet of size m**

Introducing the following patterns and sizes of alphabets into the self-made program we obtain table 6.1.

Two patterns appear in this table 6.1.

First, fixing the size of the alphabet (thus, staying in the same column) and letting $n$ be the size of the pattern then the simple form of the generating function has $z^n$ in the numerator and a degree-n polynomial in the denominator. Second, fixing the size of pattern (thus, staying in the same row) and letting m be the size of the alphabet we get the denominator of the form $(1 - mz) \cdot P(z)$ where $P(z)$ is $1 - (m-1)z - (m-1)z^2 - ....$
Therefore, in general we get the following simple form for a pattern of size n and an alphabet of size m:

$$\frac{z^n}{(1 - mz)(1 - \sum_{k=0}^{n}(m-1)z^k)}$$

Now if we separate it into two fractions we always get something of the form:

$$\frac{1}{(1 - mz)} - \frac{\sum_{k=0}^{n-1} z^k}{1 - \sum_{k=1}^{n}(m-1)z^k}$$

To get the formula we showed in a previous chapter that $\frac{1}{1-mz}$ was equivalent to the formula $m^n$. In that same chapter we showed how in general the numerator affected the initial terms and the denominator fixed the recurrence.

To make it clearer, we can take the example of pattern $AAAA$ ( $n = 3$ ) in the genetic code ($m = 4$).This gives us the following simple form:

$$A(x) = \frac{z^4}{(1-4z)(1-3z-3z^2-3z^3-3z^4}.$$

From here we separate it into two fractions:

$$\frac{1}{(1-4z)} - \frac{1+z+z^2+z^3}{1-3z-3z^2-3z^3-3z^4}.$$

Then, we know that the solution will be something of the form

$$A(n) = 4^n - P_n,$$

where $P_n$ is the linear recurrence $P_n - 3P_{n-1} - 3P_{n-2} - 3P_{n-3} = 0$ and initial terms equal to 1,4,16 and 64. The initial terms can be extracted from the following.

1. Start with the 1 we have in the numerator.
2. Only following the recurrence we would get $3 \cdot 1 = 3$ for the linear term, but we have to add z, therefore $3 + 1 = 4$.
3. For the quadratic term we would get $3 \cdot 1 + 3 \cdot 4 = 15$, $+1z^3$ in the numerator we get 16.
4. For the cubic term we would get $3 \cdot 1 + 3 \cdot 4 + 3 \cdot 16 = 63$. Again we have to add 1 giving 64.

Note that all the initial terms are powers of 4. This in fact is quite logical since we must get the terms 0 to 3 equal to 0 because we words cannot contain a pattern bigger than itself. Therefore in general we know that the recurrence $P_n$ will have n initial terms equal to $m^0, m^1, m^2, ..., m^{n-1}$. Having the recurrence and the necessary initial terms we can get the formula for the pattern in linear time (calculating the recurrence) or constant time, by solving the formula of the recurrence using the characteristic polynomial explained in the introductory chapter.

Note that since the recurrence grows asymptotically slower than $m^n$ then $\liminf \frac{P_n}{m^n} = 0$ and therefore $\liminf \frac{m^n - P_n}{m^n} = 1$ as we see in figure 6.1, with the example of pattern 00 and binary alphabet which follow the formula $2^n - F_n$ as showed in Chapter 1. **Pattern $0 \diamond 0 \diamond \overset{n}{\ldots} \diamond 0 \diamond 0$ and binomials**

A possible application of the algorithm created in the previous chapter using gap characters is counting the words with a particular number of characters. For example the generating function associated to pattern $0\diamond0$ counts the number of words containing *at least* two zeros and the pattern $0 \diamond 0 \diamond 0$ *at least* three zeros. We can generate the table depending on the size of the pattern ( counting only the number of zeros ) and fixing the size of the alphabet to binary.Results are shown in table 6.2

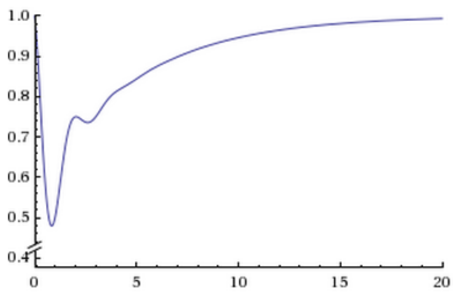


Figure 6.1: $\frac{2^n-F_n}{2^n}$

| Pattern | Simple form | Simple Fractions | Formula |
|---|---|---|---|
| $0$ | $\frac{z}{(1-z)(1-2z)}$ | $\frac{1}{1-2z}-\frac{1}{1-z}$ | $2^n-1$ |
| $0\diamond 0$ | $\frac{z^2}{(1-2z)(1-z)^2}$ | $\frac{1}{1-2z}-\frac{1}{(1-z)^2}$ | $2^n-1-n$ |
| $0\diamond 0\diamond 0$ | $\frac{z^3}{(1-2z)(1-z)^3}$ | $\frac{1}{1-2z}-\frac{1}{1-z}+\frac{1}{(1-z)^2}-\frac{1}{(1-z)^3}$ | $2^n-1-\frac{n(n+1)}{2!}$ |
| $0\diamond 0\diamond 0\diamond 0$ | $\frac{z^4}{(1-2z)(1-z)^4}$ | $\frac{1}{1-2z}-\frac{2}{(1-z)^2}+\frac{2}{(1-z)^3}-\frac{1}{(1-z)^4}$ | $2^n-1-\frac{n(n^2+5)}{3!}$ |
| $0\diamond 0\diamond 0\diamond 0\diamond 0$ | $\frac{z^5}{(1-2z)(1-z)^5}$ | $\frac{1}{1-2z}-\frac{1}{1-z}+\frac{2}{(1-z)^2}-\frac{4}{(1-z)^3}+\frac{3}{(1-z)^4}+\frac{1}{(1-z)^5}$ | $2^n-1-\frac{n(n+1)(n^2-3n+14)}{4!}$ |

Table 6.2: Pattern $0\diamond 0\diamond \overset{n}{\ldots} \diamond 0\diamond 0$

In general we can see that the simple form of the generating function will be:

$$\frac{z^n}{(1-2z)(1-z)^n},$$

which gives a simple fraction decomposition involving $\frac{1}{1-2z}$ and all the $\frac{1}{(1-z)^i}$ for $1\leq i\leq n$, for which we determined the formulas in the introduction to generating functions. With this we always get a formula of the form $2^n-1-P(x)$ where $P(x)$ is a polynomial of $n-1$ degree divided by $(n-1)!$.
Apart from the interest of getting a generating function to count this type of combinatoric classes, and the importance of the simplicity of doing so with respect to the difficulty of doing it with recurrences or formulas, we can find another application.

We can only tell how many patterns appear with a given pattern, without knowing anything else about those words.Therefore, it is not possible to create a pattern which is equivalent to count the number of words with *exactly* n zeros because we would not be able to tell if other zeros do appear in those words. However, we can subtract generating functions to get other results.

For example, if we want to get the number of words with *exactly* 3 zeros, we can take the

| Pattern | Simple form | Simple Fractions | Formula |
|---|---|---|---|
| one 0 | $\frac{z}{(1-z)^2}$ | $\frac{-1}{(1-z)}+\frac{1}{(1-z)^2}$ | $n$ |
| two 0 | $\frac{z^2}{(1-z)^3}$ | $\frac{1}{(1-z)}-\frac{1}{(1-z)^2}+\frac{1}{(1-z)^3}$ | $\frac{n(n-1)}{2!}$ |
| three 0 | $\frac{z^3}{(1-z)^4}$ | $\frac{-1}{1-z}+\frac{3}{(1-z)^2}-\frac{1}{(1-z)^3}+\frac{1}{(1-z)^4}$ | $\frac{n(n-1)(n-2)}{3!}$ |
| four 0 | $\frac{z^4}{(1-z)^5}$ | $\frac{1}{1-z}-\frac{4}{(1-z)^2}+\frac{6}{(1-z)^3}-\frac{4}{(1-z)^4}+\frac{1}{(1-z)^5}$ | $\frac{n(n-1)(n-2)(n-3)}{4!}$ |
| m 0 | $\frac{z^m}{(1-z)^{m+1}}$ | $\sum_{k=0}^{m}\frac{(-1)^{m+k}\binom{m}{k}}{(1-z)^k}$ | $\binom{n}{m}$ |

Table 6.3: Table for binomials

generating function associated to $0 \diamond 0 \diamond 0$ which counts the number of words with *at least* 3 zeros and subtract the one associated to $0 \diamond 0 \diamond 0 \diamond 0$ which counts the number of words with *at least* 4 zeros. From this we get Table 6.3:

Note the importance of the formula in the last column. Counting the number of words of size m with 3 zeros is equivalent to $\binom{n}{3}$, n *choose* 3, since we have to choose 3 positions to put the zeros from n possible positions. Using both an algorithm of computer science and combinatorics logic we get a result that could also be considered arithmetic. To get the binomial $\binom{n}{m}$ we have to look at the generating function

$$\frac{z^m}{(1-z)^{m+1}}$$

and look at the $n^{th}$ coefficient.
Moreover, a very interesting pattern occurs with simple fractions since the numerators follow the famous Pascal's Triangle, from which we can get the binomials: a great example of combination of different fields to get an important result.

## 6.4 Chapter Remarks

In this chapter we analyzed a possible algorithm to get the generating function associated to a particular pattern. We needed the inspiration of searching automata to then build a system of equations based in the symbolic method representation. To solve that method we used matrices and Gaussian Elimination in Particular. After this, we may try separating the simple form into partial fractions and obtain a formula or obtaining the first coefficients.

However, we also saw that this method was very slow to calculate and thus a program would be needed to solve reasonably long patterns, which will be explained in the next chapter.

# Chapter 7

# Programming a solver

*The purpose of computing is insight, not numbers.* **Richard Hamming**

As we have shown, all the necessary calculations to get a closed formula are very long, complicated and tedious. Therefore, programming a solver was not only interesting, but also necessary to do more complicated projects in a reasonable time. An example of that is that only to get the generating function for methionine which has only size 3, the author had to stop his calculations after 3 pages without even getting through half of the Gaussian Elimination. Therefore, all the generating functions shown in the previous chapter were calculated with some of the next programs. Combinatorics and recurrences were used to check the programs with some of the simpler cases.

Some of the programs, however, turned out to be also of a high difficulty, not so much because of the necessity to use complex programming tools (which is also true, such as vectors of 4 dimensions), but the complexity of the programs themselves, as the reader will later see. Therefore, to simplify the reading none of the following codes is the execution one (they are all in the appendix), but are what we call *pseudo codes* which explain the algorithm without giving all the details as the real codes have some hundreds of lines. In this case the author decided to write in a simplification of C++ language in order to facilitate to the interested reader the lecture of the real codes.

In fact, more than one approach was necessary until very satisfactory results were accomplished; this chapter reflects with its structure the actual procedure of search of a good approach to program a pattern solver into generating functions.

Again, the procedure of building the automata has already been explained. Note that the function that calculates it is called in Algorithm 9.2, line 3.

## 7.1 From pattern to matrix

The hard part of obtaining the matrix is putting the z's in the correct places. Taking the 1's from the identity and subtracting the matrix with the z's is not shown because the complexity of the code increases substantially because we have to store polynomials and not integers.However, note that the idea behind it, is straightforward. The explanation line by line of this algorithm can be

**Algorithm 4** Getting the Matrix

```
Require: string P, integer $z = |\Sigma|$
  $d \leftarrow Compute_Automata(d, P, z)$
  $m \leftarrow size(P)$
  $v \leftarrow matrix(m + 1bym + 2)$
  for all $i \in d$ do
    for all $j \in d[i]$ do
      $v[i][d[i][j]] \leftarrow v[i][d[i][j]] + 1$
    end for
  end for
  return v
```

found in the annex. Here we will only indicate the main point, inside both for's: Given the row ( determined by the first for ), check every column (possible character in the alphabet ). These two parameters describe an arrow. The start state is clearly the row we are analyzing, and the end state is saved in d[i][j]. Therefore we have to add a z (+1) to v[i][d[i][j]].

Now we only have to subtract this matrix from the identity, add the column with the solutions (all 0's and a 1 in the last row). Let us now analyze how we solve the following system.

## 7.2 From matrix to Generating Function

### 7.2.1 First approach to Gaussian Elimination with real numbers

The author found convenient to first write a program to solve a system of equations in the reals. That program was much simple than the one using polynomials, as it will later be analyzed, but it was very helpful to get the initial idea. Therefore, let us use this program to illustrate the general idea of the program to later see the adjustments to polynomials. Again, the real program can be found in the appendix, as we will use a *pseudocode*: This relatively simple program shows the

**Algorithm 5** Gaussian Elimination

```
Require: matrix v
  $n \leftarrow size(v)$
  $ok \leftarrow true$
  for $i = 0$ to $n - 1$ and while ok do
    if $v[i][j] == 0$ then
      $ok \leftarrow false$
      for $j = i + 1$ to $n - 1$ and while ok do
        if $v[j][i] \neq 0$ then
          $aux \leftarrow v[i]$
          $v[i] \leftarrow v[j]$
          $v[j] \leftarrow aux$
          $ok \leftarrow true$
        end if
      end for
    end if
    if ok then
      $divisor \leftarrow v[i][i]$
      for $j = 0$ to n do
        $v[i][j] \leftarrow v[i][j]/divisor$
      end for
      for $j = 0$ to $n - 1$ do
        if $j \neq i$ and $v[j][i] \neq 0$ then
          $val \leftarrow v[j][i]$
          for $k = 0$ to n do
            $v[j][k] \leftarrow v[j][k] - (v[i][k] \cdot val)$
          end for
        end if
      end for
    end if
  end for
  if ok then
    print v
  else
    print "Depending system"
  end if
```

procedure to do the Gaussian Elimination for reals, however for functions it is much harder. First, instead of storing it in a matrix (Dimension 2), we will have to store it in a Dimension 4 container. To store every function we need a size 2 container, each storing a polynomial: 1 container for the numerator and 1 for the denominator. Then each polynomial is itself a linear array containing each of the coefficients. Moreover, for the purposes of our research, we need every polynomial to have integer coefficients.

The author considered the option of creating a Dimension 5 container. Instead of being two arrays, both the numerator and denominator would instead be an array of polynomials. The advantage of doing so would be an easy multiplication, division and simplification of fractions. However, the fact of having to add or subtract at one point in the algorithm made it hard to do this approach and working in 5 dimensions is substantially harder than it is to work with 4, which is already hard. Therefore, after trying, the author decided to do the approach in dimensions 4.

Now let us analyze the dimension 4 algorithm.

- **Multiplication**Consider two polynomials of degree n and m. Can multiply and then add all the possibilities of multiplications, as we do with the cartesian product or we do in school. This gives us a $\Theta(nm)$ algorithm, which translates into a $O(n^2)$ algorithm with both polynomial of at most degree n. The author considered the option of doing a known algorithm in $O(n\lg(n))$, but there is a high constant term in this algorithm since we have to do three procedures (called Evaluation, Pointwise multiplication and Interpolation), which makes it useless for polynomials of very low degree as ours.
- **Sum and subtraction** To add or subtract, we can simply do:
$$\frac{A}{B} - \frac{C}{D} = \frac{AD - BC}{AD}$$
or the equivalent for addition.
- **Division**: we use again multiplication:
$$\frac{A}{B}/\frac{C}{D} = \frac{AD}{BC}$$

Since all the algorithms are based in multiplication we will only show the algorithm for multiplication:

This is exactly the algorithm described previously doing the cartesian product of the polynomials. Note that VI means vector of integers, which stores a polynomial. Also note that the size of v is $A + B - 1$ because the size of A is $deg(A) + 1$ since we also have the constant term, and the same for B. Therefore the size of v must be $deg(A) + deg(B) + 1 = A - 1 + B - 1 + 1 = A + B - 1$.

**Algorithm 6** Multiplication of Polynomials

```
Require: polynomial a, polynomial b
  $v \leftarrow$ polynomial of size(size(a)+size(b)-1)
  for all $i \in a$ do
    for all $j \in b$ do
      $v[i+j] \leftarrow v[i+j] + (a[i] \cdot b[j])$
    end for
  end for
  return v
```

**Problems to this algorithm**

This algorithm has a very high number of multiplications: apart from being a slow computation, it increases the degree of the polynomials very fast.

In fact, the major problem turned out to be the second one and not the first. Having to change the size of a vector (which can have a great cost) or multiplying polynomials did not cost so much as having to deal with both high degrees and high coefficients. Therefore there is the major necessity of reducing both, since there is a maximum capacity for both size of the containers and size of the coefficients.

### 7.2.2 First approach: guess and check

To solve this problem it was necessary to create a function to simplify the coefficients. Factorizing a polynomial, as with integers, has a sub-exponential complexity, so it was not a reasonable approach. The other approach was to do the Euclidean algorithm to find the Greatest Common Divisor, however it could not be done in $\mathbb{Z}$, for example with $x^2 + 2x^2 + x^3$ and $2x^2 + 2x$ since we cannot eliminate the the coefficient of third degree.

The only possible approach in $\mathbb{Z}$ was the most naive approach: to create a list of 14.000 irreducible (in $\mathbb{Z}$) polynomials of at most degree 5 (in 2 hours of computation, the program is added in the annex) and then do the the algorithm represented in 7.

**Explanation of the algorithm:** Given the list of possible divisors, we check every one of them, if both numerator and denominator are divisible. Then we divide both of them by this divisor, in case one or both of them are not divisible, we continue to look through the list of divisors. Note that if both of them are divisible in the next loop we will continue to look at the same divisor since it is possible that they share this divisor more than once.

**Algorithm 7** Simplifying a fraction of polynomials

```
Require: polynomial num, polynomial den
  for all divisors D in the list do
    while D|num and D|den do
      $num \leftarrow num/D$
      $den \leftarrow den/D$
    end while
  end for
```

### 7.2.3 Second approach: Euclidean algorithm with reals

However, there is a way of using the Euclidean algorithm, going from $\mathbb{Z}$ to $\mathbb{R}$. This approach, was not the first option because there was a loss of precision and rigor, since we stated that the coefficients had to be in $\mathbb{Z}$: it is quite clear that, even working in $\mathbb{R}[x]$, the results should be in $\mathbb{Z}[x]$ since the results should not change depending in the method. However, the hypothesis was that there could be a loss of precision with reals that modified the output, as it happened.

The Euclidean algorithm is a simple,old and still very useful algorithm, which uses the definition of the greatest common divisor.
The Euclidean algorithm is represented in Algorithm 8.

**Algorithm 8** Euclidean Algorithm

```
Require: a,b
  if b==0 then
    return a
  else
    return EuclideanAlgorithm(b,a mod b)
  end if
```

**Theorem 7.2.1** ***Correctness of the Euclidean algorithm:***

**Proof:** Let $g = GCD(a, b)$, by definition g is the biggest integer so that it divides both a and b. Take $a = bq + r$ where q,r$\in \mathbb{Z}$ and r<b.

$g \mid b \Rightarrow g \mid bq$. Moreover, by definition $g \mid a$, therefore $g \mid (a - bq)$.

Since $a - bq = r$, $g$ must also divide $r$. Therefore $gcd(a, b) = gcd(b, r)$.

Now, since $r < b$ the sequences of b are strictly decreasing. A strictly decreasing sequence of positive integers must at some point arrive at 0.

However, everything divides 0 ( since $a \cdot 0 = 0$ ), therefore $gcd(a', 0) = a'$.

Going back, we know that this a' is the gcd of the initial a and b. $\square$

We proved the Euclidean algorithm for integers to use more familiar terms, however note that this prove can be done in any field with division. Division is defined formally as: given $b \neq 0$ and $a \in \mathbb{F}$ there exist $q, r \in \mathbb{F}$ such that $a = bq + r$ and r<b.Note, for instance, that as we showed $\mathbb{Z}[x]$ had no division, but $\mathbb{Q}[x]$ does have division.

**Lemma 7.2.2** ***Existence of division in*** $\mathbb{R}[x]$**.**

**Proof:** Take $a, b \in \mathbb{Q}[x]$, suppose $deg(a) \geq deg(b)$. Otherwise $deg(a) < deg(b)$ let $q = 0$ and $r = a$ and we are done.

First, divide both polynomials by the leading coefficient of b, making it 1. We can do this since division certainly exists in $\mathbb{Q}$. Now increase q by $a_n x^{n-m}$ where n is the degree of a, m is the degree of b and $a_n$ is the leading coefficient of a. Note that this reduces at least by one the degree of $a - bq$ and therefore of r.

Since we can do this as long as $deg(b) < deg(a)$, we can use the same fact we used to prove the Euclidean algorithm to see that at some point we can make $deg(r) < deg(b)$, finishing the proof. $\square$

To perform the Euclidean algorithm to find the greatest common divisor to after divide both numerator and denominator by it, only an algorithm for division is needed. However, the algorithm used for integers works very similarly as it does in rationales.

The type of C++ variables used to implement this algorithm were 'doubles', used to store reals and not rationales. This first had a reason: using two integers to represent each coefficient would be both more complex to program and as inefficient as integers itself, because the size of the fractions would also grow. However, doubles have also a limit of precision of around 9 decimal places. This create a lot of problems since, for example 2.000000001 does not divide 10. To solve them, some margin of error was added, and periodically, reals very similar to integers were converted to integers such as 9.9999 to 10.0 or 0.00000023 to 0.

However, although the performance increased in time ( since the Euclidean algorithm is much faster than guess and check ) and precision being able to generate functions for sequence of 7 binary digits instead of 3. However, this could not be considered enough.

### 7.2.4 Third approach: $\mathbb{Z}_P$

$\mathbb{Z}_P$ is the whole numbers considered modulo a prime. Therefore some examples would be $\mathbb{Z}_2$,$\mathbb{Z}_{23}$ or $\mathbb{Z}_{215443}$. In $\mathbb{Z}_2$ all odd numbers are equal to 1 and all even numbers are equal to 0. In $\mathbb{Z}_{23}$ all numbers of the form $17 + 23k$ would essentially be equal to 17, and all the numbers of the form $13 + 23k$ would be equal to 13, etc.

$\mathbb{Z}_P$ satisfies a lot of properties that other rings do not. For example, in $\mathbb{Z}_P$ every non-zero element divides 1 as proved in Theorem 7.2.3. $\frac{1}{15}$ mod 23 is 20, and $\frac{1}{15}$ mod 215443 is 114903; this means that $15 \cdot 20 \equiv 1$ mod 23 and $15 \cdot 114903 \equiv 1$ mod 215443.

**Theorem 7.2.3** ***Every non-zero element of $\mathbb{Z}_p$ is a unit.***

**Proof:** Notice that every number that in order for a not to be coprime with p is GCD(a,p)=p, since GCD=1 implies they are coprime and by definition, P has only 1 and itself as divisors. But $P \mid a \to a \equiv 0$ mod p. Therefore if a is not 0 mod p, it is coprime with p. Now we can prove:

$$GCD(a, m) = 1 \Leftrightarrow \text{a is a unit mod m}$$

First let us prove $GCD(a, m) = 1 \to$ a is a unit mod m: $u \cdot a + m \cdot b = 1$ for some $a, b \in \mathbb{Z}$ follows from the Diophantine equation ( the author has proved it from the axioms, but it is not in the limits of this work ).

$$ua = 1 + m(-b)$$

$$ua \equiv 1\text{mod m (by the definition of mod)}$$

$$u \in U_m \text{ by definition of unit)}$$

Now we can prove the other sense (which is following the proof in the inverse sense):

$$u \in U_m$$

$$ua \equiv 1\text{mod m (by definition of unit)}$$

$$ua = 1 + m(b')\text{by definition of mod)}$$

$$ua + m(-b') = 1$$

$$GCD(a, m) = 1\text{by the Diophantine equation}$$

□

If every non-zero element divides 1, every element divides every element in $\mathbb{Z}_P$ since $\frac{A}{B} = \frac{A}{1} \cdot \frac{1}{B} = A \cdot \frac{1}{B}$.

Therefore if we take a polynomial of coefficients with degrees in $\mathbb{Z}_P$ we will have division and therefore we will be able to do the Euclidean Algorithm as proved before.

The only problem that arises is how to compute $\frac{1}{B}$.

This problem is equivalent to say: what is the number X that satisfies $X \cdot B \equiv 1$ mod p?

We can use Fermat's Little Theorem ( 7.2.4) which states:

$$a^{p-1} \equiv 1 \text{ mod p},$$

Let us start with:

$$X \cdot B^1 \equiv 1 \text{ mod p},$$

$$X \equiv B^{-1} \text{ mod p},$$

and now we can use the fact that $B^{p-1} \equiv 1$ mod p to get:

$$X \equiv B^{-1} \cdot B^{p-1} = B^{p-2} \text{ mod p}.$$

Using binary exponentiation we can get $B^{p-2}$ in $O(lg(p))$ time, which is very fast. The algorithm for binary exponentiation is illustrated in Algorithm 9 Let us take the example with $2^{23}$:

1. $2^{23} = 2 \cdot (2^{11})^2$
2. $2^{11} = 2 \cdot (2^5)^2$
3. $2^5 = 2 \cdot (2^2)^2$
4. $2^2 = (2^1)^2$
5. $2^1 = 2 \cdot (2^0)^2$
6. $2^0 = 1$

**Algorithm 9** Binary exponentiation

```
Require: int base, int exponent
  if EXPONENT==0 then
    return 1
  end if
  $aux \leftarrow Binaryexponentiation(base, \lfloor base/2 \rfloor)$
  $aux \leftarrow aux^2$
  if exponent mod 2==0 then
    $aux \leftarrow aux \cdot base$
  end if
  return aux
```

We can now do back substitution to get the result in very few calculations.

**Theorem 7.2.4** ***Fermat's Little Theorem:***

**Proof:**

$$a^{p-1} \equiv 1 \text{mod p}$$

To prove it we will simply prove its generalization:

$$a^{\varphi(m)} \equiv 1 \text{mod m}$$

Remember that $\varphi(m)$ means at the same time the number of units in $\mathbb{Z}_m$ and the number of elements coprime with m which are less than m. This equivalence follows from Theorem 7.2.3.

For every unit in $U_m$ take:

$$a \cdot u_i = v_i$$

Multiply all the products getting:

$$\Pi_{i=1}^{\varphi(m)} a \cdot u_i \equiv \Pi_{i=1}^{\varphi(m)} v_i$$

$$\Pi_{i=1}^{\varphi(m)} a \cdot \Pi_{i=1} u_i \equiv \Pi_{i=1}^{\varphi(m)} v_i$$

Since$\{u_1, u_2, ..., u_{\varphi(m)}\}$ is a permutation of $\{v_1, v_2, ..., v_{\varphi(m)}\}$ (proved in Lemma 7.2.5) $\Pi_{i=1}^{\varphi(m)} u_i = \Pi_{i=1}^{\varphi(m)} v_i$. The we can divide by the inverse of this product (since they are all units ) getting:

$$\Pi_{i=1}^{\varphi(m)} a \equiv 1 \text{mod m}$$

$$a^{\varphi(m)} \equiv 1 \text{mod m}$$

□

**Lemma 7.2.5**

**Proof:** $\{u_1, u_2, ..., u_{\varphi(m)}\}$ is a permutation of $\{v_1, v_2, ..., v_{\varphi(m)}\}$.
Suppose:

$$u \cdot a_j \equiv v_i$$

$$and$$

$$u \cdot a_g \equiv v_i$$

$$then$$

$$ua_j \equiv ua_g$$

We can multiply by $u^{-1}$,

$$a_j \equiv a_g$$

which proves that the function is 1-to-1.

Since $GCD(a, b) = 1$ and $GCD(a, c) = 1 \rightarrow GCD(a, bc) = 1$ $U_m$ is closed under multiplication (applying Theorem 7.2.3). A function which is 1-to-1 and goes from a set S to the same set S must also be onto, and therefore a bijection. (All these concepts are defined at the beginning of the study ). Therefore $\{u_1, u_2, ..., u_{\varphi(m)}\}$ is a permutation of $\{v_1, v_2, ..., v_{\varphi(m)}\}$. □
Now we can perform Gaussian Elimination with a lot of precision since we will never get an overflow ( with integers bounded by p ) or decimal problems ( since they are integers ). The only problem that might arise is the uncertainty that $\mathbb{Z}_P$ implies. We will have to assume, unfortunately, that every coefficient lies between $-\frac{1}{2}$ and $+\frac{P}{2}$. This is not a major catastrophe for two reasons:

1. Note that, for sure, working with numbers less than our limit but over the square root of this limit is dangerous since the multiplication would create an overflow. To be cautious we can be sure to have no overflow problems by working with numbers less than the cubic root of our limit. The biggest prime number less than the cube root of $2^{63} - 1$, our limit, is the already mentioned 215,443. It is very reasonable to assume that, for quite small or normal patterns and alphabets, the coefficients cannot get bigger than 100,000.

2. Another very important tool we can use is the *Chinese Remainder Theorem.*

The *Chinese Remainder Theorem* (proved outside this study ) states that given $a$ mod m and $a$ mod n, there is exactly one $a$ mod LCM(m,n) that satisfies both congruences. Since we are always using primes LCM of all the primes is equivalent to the product of the primes, which grows exponentially. For example, the product of primes less than 100 is 2305567963945518424753102147331756070 or the product of the 5 bigger primes less than $\sqrt[3]{2^{63}-1}$ is 3 orders of magnitude bigger than Avogadro's number. In other words, we can easily obtain very big ranges that will procure a reasonable credibility to our results.

By modifying the automaton (to process different types of patterns or combinations of patterns) those programs will easily adjust to get the generating function. As one can see, in this chapter math greatly influenced our Computer Science algorithms, showing again the value of an interdisciplinary approach. Furthermore, apart from a theoretically interesting result it was a necessary implementation. Some of the applications of this program were the calculations done in the previous chapter and the applications performed in the next chapter.

# Chapter 8

# Applications

*Science floats on a sea of mathematics*

In this last chapter we will analyze genetics with the tools we developed in the previous chapters to show one of the many possible implementations that type of analysis could have.

**Important note:** It is very important to take into account that this chapter is very experimental in the sense that it shows what a possible research could be and does not have the intention of proving or showing anything with certainty. Although some specifical genetical knowledge has been acquired by the author in order to do this experiment, it was out of the limits of the initial objectives (which ended in the previous chapter ).

In this case we took the gene of Cystic Fibrosis as a representative of a human gene [19]; although it would be desirable to take more than one gene.

The analysis is done at the level of nucleotides for the following reasons:

1. It is a small alphabet: perfect for our analysis
2. With this we have a huge amount of data ( more than 100,000 nucleotides )
3. Mutations and therefore evolution is done at the level of nucleotides
4. Some parts of genes do not go three by three like the coding genes

Although the analysis is done at the level of nucleotides, the analysis will be done by searching each amino acid as a structure, which is equivalent to 3 nucleotides. The author is conscious that genes do not function in every possible shift, as they will be analyzed now, however it was considered interesting to take this completely new approach.

The first step is to create a table with the generating function associated to every possible codon, a triple of nucleotides found in Table 8.1.

| | Last base | | | |
|---|---|---|---|---|
| | A | C | G | U |
| AA | $\frac{z^3}{(1-3z)(1-2z-2z^2-2z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| AC | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| AG | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| AU | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| CA | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| CC | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-3z)(1-2z-2z^2-2z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| CG | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| CU | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| GA | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| GC | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| GG | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-3z)(1-2z-2z^2-2z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| GU | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ |
| UA | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ |
| UC | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ |
| UG | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^2-3z^3)}$ |
| UU | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-4z)(1-4z+z^3)}$ | $\frac{z^3}{(1-3z)(1-2z-2z^2-2z^3)}$ |

Table 8.1: Generating functions associated to every codon

As one can see, although there are some repetitions, it may vary substantially between different codons. We can go further in our analysis and consider amino acids, which are associated to one or more codons. For this, we used the program that determined if in a text there is *any* of a list of patterns to analyze amino acids getting Table 8.2. This time we added a partial fraction decomposition to show a first image of what could be a possible formula.

| | Codon | Simple Form | Partial fraction |
|---|---|---|---|
| Alanine | GC_ | $\frac{4x^3}{1-8x+17x^2-4x^3}$ | $\frac{1}{1-4x}-\frac{4x}{1-4x-x^2}-1$ |
| Arginine | CG_,AGA,AGG | $\frac{6x^3+4x^5}{1-8x+18x^2-10x^3+9x^4-4x^5}$ | $\frac{1}{1-4x}-\frac{2x^3+4x}{1-4x+2x^2-2x^3+x^4}-1$ |
| Asparagine | AAU,AAC | $\frac{2x^3}{1-8x+16x^2+2x^3-8x^4}$ | $\frac{1}{1-4x}-\frac{1}{1-4x+2x^3}$ |
| Aspartic acid | GAU,GAC | $\frac{2x^3}{1-8x+16x^2+2x^3-8x^4}$ | $\frac{1}{1-4x}-\frac{1}{1-4x+2x^3}$ |
| Cysteine | UGU,UGC | $\frac{2x^3}{1-8x+17x^2-6x^3+8x^4}$ | $\frac{1}{1-4x}-\frac{1+x^2}{1-4x+x^2-2x^3}$ |
| Glutamic acid | GAA,GAG | $\frac{2x^3}{1-8x+17x^2-6x^3+8x^4}$ | $\frac{1}{1-4x}-\frac{1+x^2}{1-4x+x^2-2x^3}$ |
| Glutamine | CAA,CAG | $\frac{2x^3}{1-8x+16x^2+2x^3-8x^4}$ | $\frac{1}{1-4x}-\frac{1}{1-4x+2x^3}$ |
| Glycine | GG_ | $\frac{4x^3}{1-7x+9x^2+12x^3}$ | $\frac{1}{1-4x}+\frac{1}{3}-\frac{4}{3-9x-9x^2}$ |
| Histidine | CAU,CAC | $\frac{2x^3}{1-8x+17x^2-6x^3+8x^4}$ | $\frac{1}{1-4x}-\frac{1+x^2}{1-4x+x^2-2x^3}$ |
| Isoleucine | AUU,AUC,AUA | $\frac{3x^3}{1-8x+17x^2-5x^3+4x^4}$ | $\frac{1}{1-4x}-\frac{1+x^2}{1-4x+x^2-x^3}$ |
| Leucine | CU_,UUA,UUG | $\frac{6x^3-2x^4}{1-8x+17x^2-2x^3-10x^4+8x^5}$ | $\frac{1}{1-4x}-\frac{1+x^2}{1-4x+x^2+2x^3-2x^4}$ |
| Lysine | AAA,AAG | $\frac{2x^3}{1-7x+9x^2+10x^3+8x^4}$ | $\frac{1}{1-4x}-\frac{1+x+x^2}{1-3x-3x^2-2x^3}$ |
| Methionine | AUG | $\frac{x^3}{1-8x+16x^2+x^3-4x^4}$ | $\frac{1}{1-4x}-\frac{1}{1-4x+x^3}$ |
| Phenylalanine | UUU,UUC | $\frac{2x^3}{1-7x+9x^2+10x^3+8x^4}$ | $\frac{1}{1-4x}-\frac{1+x+x^2}{1-3x-3x^2-2x^3}$ |
| Proline | CC_ | $\frac{4x^3}{1-7x+9x^2+12x^3}$ | $\frac{1-4x}{+}\frac{1}{3}-\frac{4}{3(1-3x-3x^2)}$ |
| Serine | UC_,AGU,AGC | $\frac{6x^3-4x^5}{1-8x+17x^2-2x^3-9x^4+4x^5}$ | $\frac{1}{1-4x}-1-\frac{2x^3-4x}{x^4-2x^3-x^2+4x-1}$ |
| Threonine | AC_ | $\frac{4x^3}{1-8x+17x^2-4x^3}$ | $\frac{1}{1-4x}-1-\frac{4x}{1-4x+x^2}$ |
| Tryptophan | UGG | $\frac{x^3}{1-8x+16x^2+x^3-4x^4}$ | $\frac{1}{1-4x}-\frac{1}{1-4x+x^3}$ |
| Tyrosine | UAC,UAU | $\frac{2x^3}{1-8x+17x^2-6x^3+8x^4}$ | $\frac{1}{1-4x}-\frac{1+x^2}{1-4x+x^2-2x^3}$ |
| Valine | GU_ | $\frac{4x^3}{1-8x+17x^2-4x^3}$ | $\frac{1}{1-4x}-1-\frac{4x}{1-4x+x^2}$ |
| Stop | UAA,UAG,UGA | $\frac{3x^3}{1-8x+16x^2+3x^3-12x^4}$ | $\frac{1}{1-4x}+\frac{1}{5-5x}-\frac{3(x+2)}{5(1-3x-3x^2)}$ |

Table 8.2: Generating function associated to every amino acid

We can look at the first coefficients of each generating function to look at how many texts there are that contain at least one time one of the patterns of a certain amino acid. In this case the sizes shown are 3,4,5 and 6; for example for alanine there would be 4 possible texts containing alanine of size 3, 32 of those of size 4, 188 of size 5 and 976 of size 6.

By dividing by $4^n$ we can get the probability that a text of size $n$ contains Alanine, or any other amino acid. Again the percentages for $n = 3$ to 6 are shown.

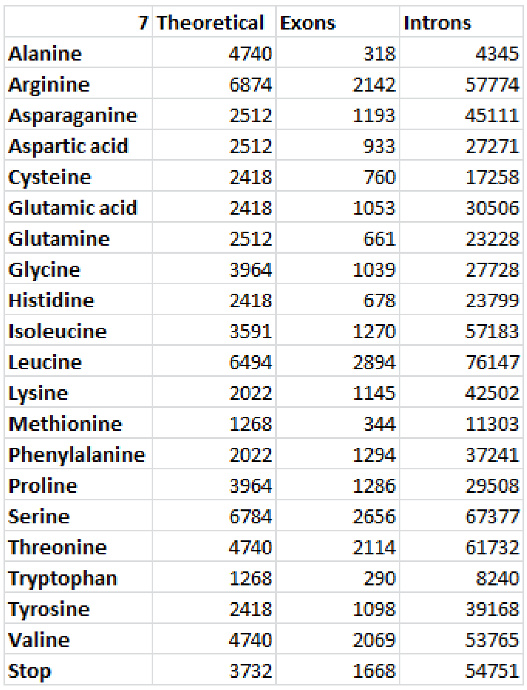
Finally a formula, if found, is shown. Those formulas were found by using a program [18] to solve the recurrences indicated by the partial fraction decompositions shown in table 8.2; however, the program was not powerful enough to find a formula for the recurrence; another example of the difficulty of dealing with recurrences to solve this complex structures. In Table 8.3 we find those three indicators, first coefficients, first probabilities and a formula, that would help us in the comparison with an actual gene.

| | First coefficients | First % | Formula |
|---|---|---|---|
| Alanine | 4,32,188,976 | 6.2,12.5,18.4,23.8 | $4^n + \frac{2(2-\sqrt{3})^n}{\sqrt{3}} - \frac{2(2+\sqrt{3})^n}{\sqrt{3}}$ |
| Arginine | 6,48,280,1436 | 9.4,18.75,27.3,35 | |
| Asparagine | 2,16,96,508 | 3.1,6.2,9.4,12.4 | |
| Aspartic acid | 2,16,96,508 | 3.1,6.2,9.4,12.4 | |
| Cysteine | 2,16,94,492 | 3.1,6.2,9.2,12 | |
| Glutamic acid | 2,16,94,492 | 3.1,6.2,9.2,12 | |
| Glutamine | 2,16,96,508 | 3.1,6.2,9.4,12.4 | |
| Glycine | 4,28,160,820 | 6.2,10.9,15.6,20 | |
| Histidine | 2,16,94,492 | 3.1,6.2,9.2,12 | |
| Isoleucine | 3,24,141,735 | 4.7,9.4,13.8,17.9 | |
| Leucine | 6,46,266,1358 | 9.4,18,26,33.2 | |
| Lysine | 2,14,80,414 | 3.1,5.5,7.8,10.1 | |
| Methionine | 1,8,48,255 | 1.6,3.1,4.7,6.2 | |
| Phenylalanine | 2,14,80,414 | 3.1,5.5,7.8,10.1 | |
| Proline | 4,28,160,820 | 6.2,10.1,15.6,20 | $4^n + \frac{(3-\sqrt{21})^{n+1}-(3+\sqrt{21})^{n+1}}{2^{n-1}3\sqrt{21}}$ |
| Serine | 6,48,278,1420 | 6.4,18.7,27.1,34.7 | |
| Threonine | 4,32,188,976 | 6.2,12.5,18.4,23.8 | $4^n + \frac{2(2-\sqrt{3})^n}{\sqrt{3}} - \frac{2(2+\sqrt{3})^n}{\sqrt{3}}$ |
| Tryptophan | 1,8,48,255 | 1.6,3.1,4.7,6.2 | |
| Tyrosine | 2,16,94,492 | 3.1,6.2,9.2,12 | |
| Valine | 4,32,188,976 | 6.2,12.5,18.4,23.8 | $4^n + \frac{2(2-\sqrt{3})^n}{\sqrt{3}} - \frac{2(2+\sqrt{3})^n}{\sqrt{3}}$ |
| Stop | 3,24,144,759 | 4.7,9.4,14,18.5 | $4^n - \frac{(21-4\sqrt{21})((3-\sqrt{21})^n)+(21+4\sqrt{21})(3+\sqrt{21})^n)-7\cdot 2^n}{2^n\cdot 35}$ |

Table 8.3: Theoretical results for aminoacids

We then took the gene of Cystic Fibrosis ([19]), looked at the start and end point of introns and exons, then divided the genes into introns and exons to analyze both separately.

We then looked at the frequency of each amino acid to appear in a substring of the exon/intron of size $n$. For example Figure 8.3 shows the results for $n = 7$. The rest of the calculations can be found in the Appendix.



| 7 | Theoretical | Exons | Introns |
|---|---|---|---|
| Alanine | 4740 | 318 | 4345 |
| Arginine | 6874 | 2142 | 57774 |
| Asparaganine | 2512 | 1193 | 45111 |
| Aspartic acid | 2512 | 933 | 27271 |
| Cysteine | 2418 | 760 | 17258 |
| Glutamic acid | 2418 | 1053 | 30506 |
| Glutamine | 2512 | 661 | 23228 |
| Glycine | 3964 | 1039 | 27728 |
| Histidine | 2418 | 678 | 23799 |
| Isoleucine | 3591 | 1270 | 57183 |
| Leucine | 6494 | 2894 | 76147 |
| Lysine | 2022 | 1145 | 42502 |
| Methionine | 1268 | 344 | 11303 |
| Phenylalanine | 2022 | 1294 | 37241 |
| Proline | 3964 | 1286 | 29508 |
| Serine | 6784 | 2656 | 67377 |
| Threonine | 4740 | 2114 | 61732 |
| Tryptophan | 1268 | 290 | 8240 |
| Tyrosine | 2418 | 1098 | 39168 |
| Valine | 4740 | 2069 | 53765 |
| Stop | 3732 | 1668 | 54751 |

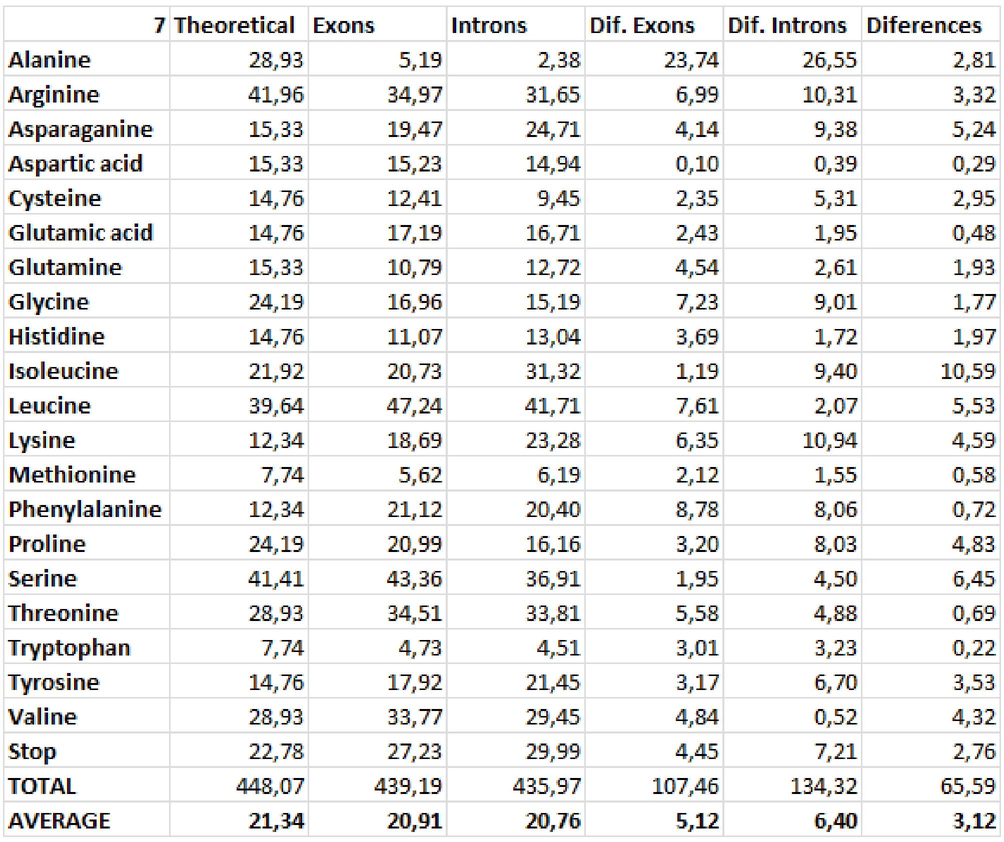
Figure 8.1: Appearances of amino acids in the gene for size 7

After this, we calculated all the percentages of each amino acid of appearing in a substring of size $n$. Again, we show the results for size $n = 7$ in Figure 8.2.

As one can see, some notable differences can be seen between what theory expects and the actual results such as for the amino acid Alanine. We can calculate the average difference in percentage between exons and theory, introns and theory and exons and introns and compare those results. To get an approximate result we should divide the percentage by the average percentage of appearance in theory, getting figure 8.3. As one can see, those results differ significantly from theory, supporting the idea of selection in those genes, instead of being completely random.

We can go further in our analysis and sketch a graph with the theoretical probability in the x-axis and the actual probability in the y-axis, as found in Figure 8.4.

As one can see from both graphs we have a slight ($R^2 = 0.4 - 0.6$) correlation between theory and practice. This could indicate that some random component can be found in both exons and introns, but that both exons and introns are not random (since they have been selected ). Surprisingly we can see that introns are even more 'antirandom' than exons, which would support the theory that they are not 'junk' DNA as it was thought. The fact that they are more



| 7 | Theoretical | Exons | Introns | Dif. Exons | Dif. Introns | Diferences |
|---|---|---|---|---|---|---|
| **Alanine** | 28,93 | 5,19 | 2,38 | 23,74 | 26,55 | 2,81 |
| **Arginine** | 41,96 | 34,97 | 31,65 | 6,99 | 10,31 | 3,32 |
| **Asparaganine** | 15,33 | 19,47 | 24,71 | 4,14 | 9,38 | 5,24 |
| **Aspartic acid** | 15,33 | 15,23 | 14,94 | 0,10 | 0,39 | 0,29 |
| **Cysteine** | 14,76 | 12,41 | 9,45 | 2,35 | 5,31 | 2,95 |
| **Glutamic acid** | 14,76 | 17,19 | 16,71 | 2,43 | 1,95 | 0,48 |
| **Glutamine** | 15,33 | 10,79 | 12,72 | 4,54 | 2,61 | 1,93 |
| **Glycine** | 24,19 | 16,96 | 15,19 | 7,23 | 9,01 | 1,77 |
| **Histidine** | 14,76 | 11,07 | 13,04 | 3,69 | 1,72 | 1,97 |
| **Isoleucine** | 21,92 | 20,73 | 31,32 | 1,19 | 9,40 | 10,59 |
| **Leucine** | 39,64 | 47,24 | 41,71 | 7,61 | 2,07 | 5,53 |
| **Lysine** | 12,34 | 18,69 | 23,28 | 6,35 | 10,94 | 4,59 |
| **Methionine** | 7,74 | 5,62 | 6,19 | 2,12 | 1,55 | 0,58 |
| **Phenylalanine** | 12,34 | 21,12 | 20,40 | 8,78 | 8,06 | 0,72 |
| **Proline** | 24,19 | 20,99 | 16,16 | 3,20 | 8,03 | 4,83 |
| **Serine** | 41,41 | 43,36 | 36,91 | 1,95 | 4,50 | 6,45 |
| **Threonine** | 28,93 | 34,51 | 33,81 | 5,58 | 4,88 | 0,69 |
| **Tryptophan** | 7,74 | 4,73 | 4,51 | 3,01 | 3,23 | 0,22 |
| **Tyrosine** | 14,76 | 17,92 | 21,45 | 3,17 | 6,70 | 3,53 |
| **Valine** | 28,93 | 33,77 | 29,45 | 4,84 | 0,52 | 4,32 |
| **Stop** | 22,78 | 27,23 | 29,99 | 4,45 | 7,21 | 2,76 |
| **TOTAL** | 448,07 | 439,19 | 435,97 | 107,46 | 134,32 | 65,59 |
| **AVERAGE** | **21,34** | **20,91** | **20,76** | **5,12** | **6,40** | **3,12** |

Figure 8.2: Comparison of frequencies of amino acids for size 7

'antirandom' is not significant as we have to remember that exons actually work only in some particular shifts, in which they are probably even more 'antirandom'.

*Note: the fact that they are all slightly decreasing is probably due to the fact that as they grow bigger and bigger percentages tend to be more similar. For example, at infinity all percentages for all aminoacids should be* 100% *and thus 'antirandomness' will be 0.*

Another very interesting point should be analyzed. If one examines the sum of the percentages in theory, exons and introns; it will observe that all are less than 500%. Our first thought might be that, since in length 7 there are 5 patterns of size 3, the sum of the probabilities should add up to 500%. However, this is not case; this is because in a word we can have a pattern more than once, but for our purposes this does not change anything for that pattern, but the total probability will decrease since maybe only 4 patterns or even 3 may be counted for each substring instead of 5. As one can see if genes were completely at random the probabilities would add up to 448%, but in practice this quantity is smaller both in exons and in introns.This means that more repetitions insight a substring are found in genes than they are at random, therefore we can conclude that

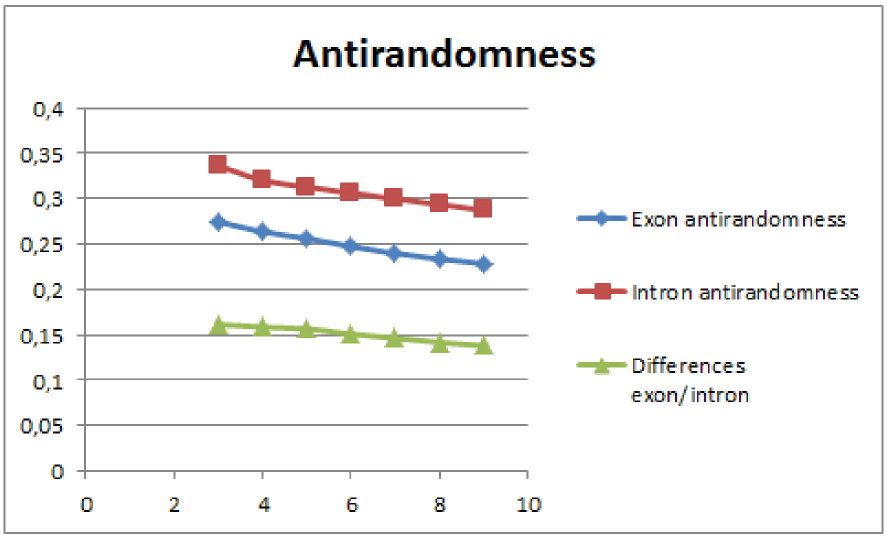


Figure 8.3: Graphic showing 'antirandomness'

genes are more repetitive than the average, as it can be seen in figure 8.5 were both graphs are bigger than one.

In conclusion, generating functions along with a searcher can help us determine some facts of the symmetry and randomness of a string, such as a gene. Although this small research cannot make strong conclusions, it is a small taste of what could be done with the combination of this tools.

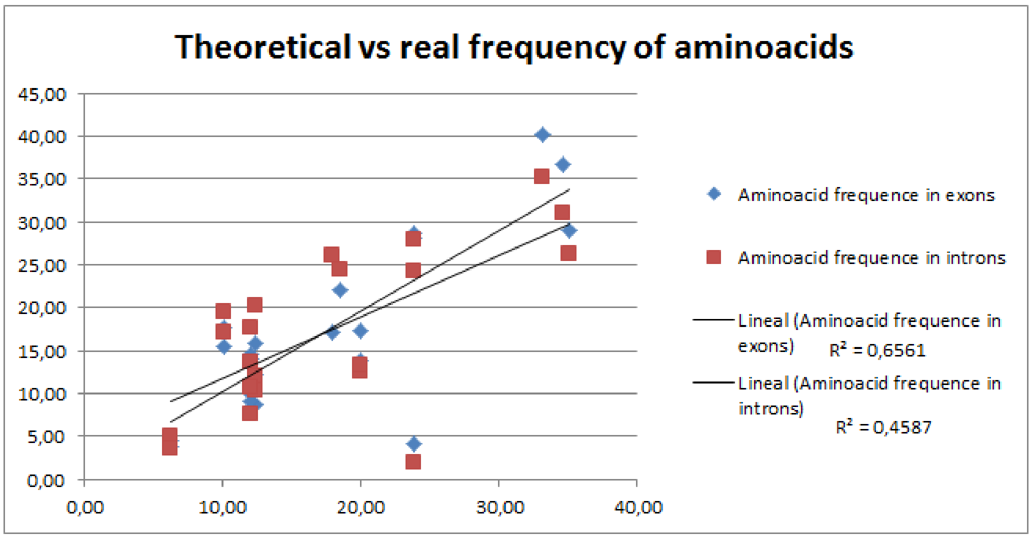


Figure 8.4: Theory versus real

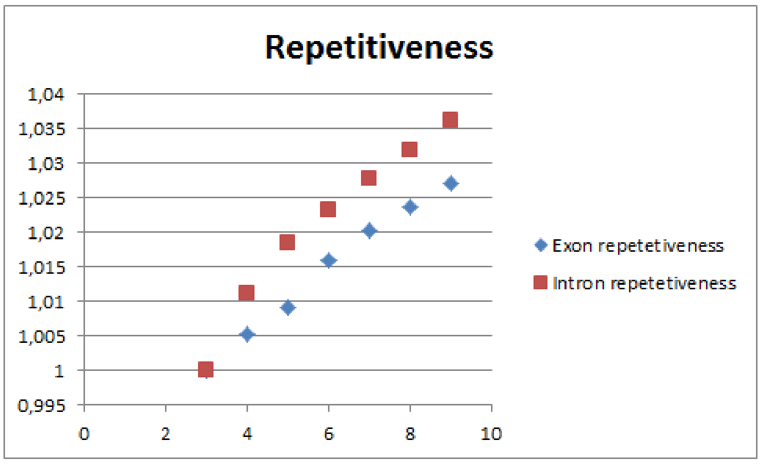


Figure 8.5: Graphic showing 'repetitiveness'

# Chapter 9

# Conclusions

*Answer the questions, then question the answers.* **Glenn Stevens**

## 9.1 General conclusions

We started from basic combinatorics, formulas and recurrences, we then introduced a much more complex tool: generating functions. After proving some properties of those objects and use them in various fields of mathematics. After this, we introduced the relation between generating functions and enumerative combinatorics in general and showing how the use of generating functions simplifies significantly some very hard problems.

However, we still could not solve our main question: *How many words of size $n$ contain a fixed pattern $P$?* Then, we moved to the algorithmic part, by showing and analyzing three different methods to solve the string-matching problem: two of them were quadratic in worst case and the last one, automata, was linear, hence it was much better. We then moved to the crucial part of the research, merging both parts together to solve the hardest question. Not only that, but all the different variations we created in the algorithmic part could be profited to also find generating functions for all those combinations of patterns. This time, the only problem were the tedious calculations that had to be done; to solve this problem, we created a program that reproduced the algorithm we explained, mainly doing the conversion between the automata transition function and a matrix and then programming Gaussian Elimination for functions to get the solutions of the system.

Finally, we gave a taste of a possible type of research that could be done thanks to this study by analyzing genomic sequences.

By defining a new mathematical object, we could prove a great diversity of important theorems in mathematics from the $cos^2(x) + sin^2(x) = 1$ in geometry to the Leibnitz rule in Calculus. Similarly, to make our pattern solver into generating functions, we had to develop a whole arithmetical base in modulus p, which was also interesting. On the other hand, it was also of great interest to study mathematically the complexity of some algorithms which in fact depend in very complex structures such as words, by doing a probabilistic analysis and analyzing automata theory. All in all we can conclude, that the learning part of this study was done satisfactorily.

After the learning part,we managed to solve the main question of the research by using all the theory we had learned and implemented. However, we went further by making an automatic solver, which is probably one of the more complicated processes of this study and one of the most important results.

We used it to find generating functions and formulas for some special patterns, although it is important to see that we could have done it for any pattern. Not only that, but any combination of patterns using 'and', 'or' and $'\diamond'$ could be constructed using the described methods.

Finally, we were able to create a specific application outside mathematics, combinatorics and computer science by analyzing genomic sequences. This showed how this study could have an application outside the study itself even if we started from very abstract mathematical ideas that seem to have absolutely no connection to reality showing a path from pure mathematics to more applied sciences.

Finally this study pretends to be a clear example of a polyvalent approach resulting in much deeper and complex results than the simple approach would have got.

Therefore, this research satisfied all its initial objectives by answering with precision its initial questions, analyzing specifical cases, creating new algorithms and finding applications.

## 9.2 Further research

The power of the combination of generating functions with automata being immense, a lot of possible ideas were found, but unfortunately the author had to stop at some point. We could divide it into two topics, more mathematical and computational theoretical studies and applications.

### Possible applications

- It is very recommendable to do the last research introducing data from more human genes.
- We could continue a genetic search for patterns by introducing more types of patterns such as the ones containing the $\diamond$ character or longer patterns.
- We could compare different species and see if there are significant differences in randomness and how superposable they are.
- We could also study music as a sequence of signs, as we did with the genetic code and find particular repetitive patterns.
- Another very interesting application, which was already considered and studied, would be to analyze stock exchanges or currency prices. For example, we could assign numbers from 1 to 5 depending on the level of absolute change in one day, thus measuring roughly volatility of the stock of a company like Apple or the value of the euro with respect to the dollar, and then look how random those sequences are and how repetitive they are. This would answer to questions like: ”are there periods of clearly more volatility than others?”. We could do the same, but instead of assigning a number from 1 to 5 depending in the change, we could put 1 and 2 a descent of the price, a 3 a stability, and a 4 and a 5 a rise of prices and then analyze changes.

### Math and Algorithmic ideas

- This research paper inspired a lot of mathematical and computer science questions that could be analyzed. For instance, we showed $\sum_{k=0}^{n}\binom{n}{k} = 2^n$ and $\sum_{k=0}^{n}\binom{2n}{2k} = 2^{2n-1}$; what is, in general, $\sum_{k=0}^{n}\binom{a\cdot n}{a\cdot k}$?
- A lot of *small* questions can be asked and solved using special patterns. For example: what is the best position to put a $\diamond$ in an n-character word to maximize its probability of appearing?
- Using the program that search for pattern $\alpha$ and $\beta$, we could search for $\alpha$ and $\alpha$ divided by the cases of appearing only one $\alpha$, obtaining the probability of multiple appearance of $\alpha$ and relate this probability with the form of generating functions of pattern $\alpha$.
- The presentation of the study done by Josep Peya and Antoni Sánchez of Aula Escola Europea about Markov Chains [12], inspired a very powerful combination of those objects with automata and therefore with generating functions. Since for each state we know the last character received and the arrow represents the future character, we can calculate the probability of the arrow using the idea behind Markov Chains.

**Final Note** The author continued its research after presenting the work to his school. The key elements of this research are the following:

- Proving formally the complexity of all the automata and algorithms shown in the paper and some other theorems mentioned there.
- Designing a program to work with Markov Chains applied to Generating Functions. This program and this tool in general can be very useful to solve problems of probabilities or mathematical expectation even if iterations go to infinity such as the problem shown in [26].
- The start of an extension of this research was attempted by analyzing other algorithms such as Knuth-Morris-Pratt or Aho-Corasick in a very similar way to the one done with automata theory, but using post-college math, making this research even more challenging and interesting.
- Using a previous experience in fractals, the author studied a stock exchange value in short and long term periods to see if it satisfied the same properties. Although properties were similar, some differences were found such as that smaller periods of time tended to have more random outcomes. This could lead to a very interesting study of stock exchange and currencies using generating functions and multifractal theory.

# Appendix

## Proofs

**Found in Chapter 1:**

**Theorem .0.1**

$$\sum_{k=0}^{n} \binom{2n}{2k} = \sum_{k=0}^{n-1} \binom{2n}{2k+1}$$

**Proof:** We start by simply doing:

$$(-1+1)^n = 0^n$$

Again, the Binomial Theorem implies:

$$(-1+1)^n = \sum_{k=0}^{n} \binom{2n}{2k} - \sum_{k=0}^{n-1} \binom{2n}{2k+1}$$

thus,

$$\sum_{k=0}^{n} \binom{2n}{2k} - \sum_{k=0}^{n-1} \binom{2n}{2k+1} = 0.$$

$$\sum_{k=0}^{n} \binom{2n}{2k} = \sum_{k=0}^{n-1} \binom{2n}{2k+1}.$$

□

**Theorem .0.2**

$$\sum_{k=0}^{n} \binom{2n}{2k} = 2^{2n-1}$$

**Proof:** For Theorem 1.2.1 we have:

$$\sum_{k=0}^{2n} \binom{2n}{k} = 2^{2n}$$

We can split it into odd and even $k$'s getting:

$$\sum_{k=0}^{n} \binom{2n}{2k} + \sum_{k=0}^{n-1} \binom{2n}{2k+1} = 2^{2n}$$

Now we can apply Theorem .0.1 getting:

$$\sum_{k=0}^{n} \binom{2n}{2k} = \frac{2^{2n}}{2} = 2^{2n-1}$$

□

**Found in Chapter 2:**

**Theorem .0.3** $sin'(x) = cos(x)$

**Proof:** $sin'(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot 2n}{(1+2n)!} x^{2n-1} =$
$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = cos(x).$ □

**Theorem .0.4** $cos'(x) = -sin(x)$

**Proof:** $sin'(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot 2n}{(2n)!} x^{2n-1} =$
$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n-1)!} x^{2n-1}$
We substitute $n$ for $n+1$.
$\sum_{n=0}^{\infty} \frac{-(-1)^n}{(2n+1)!} x^{(2n+1)} = -sin(x).$ □

## Further Analysis

### Perfect powers

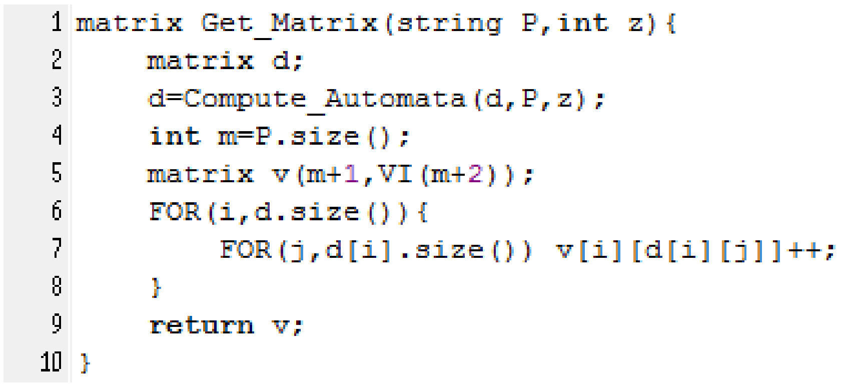
Starting with our basic generating function: $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$ there is an easy and logical way of getting the natural numbers: differentiating. $(\frac{1}{1-x})' = 1 + 2x + 3x^2 + 4x^3 + ....$ However, this does not match coefficient with degree since the coefficient n is matched with the degree $n-1$. Therefore, we have to multiply the result by x getting $x + 2x^2 + 3x^3 + 4x^4 + ... = \sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$. Iti s pretty logical that we can get the squares buy doing the same process, first differentiating and then multiplying by x. We get the following for the $n^{th}$ powers:

1. $\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + ...$
2. $\frac{x(x+1)}{(1-x)^3} = x + 4x^2 + 9x^3 + 16x^4 + ...$
3. $\frac{-x(x(x+4)+1)}{(1-x)^4} = x + 8x^2 + 27x^3 + 64x^4 + ...$
4. $\frac{x(x+1)(x(x+10)+1)}{(1-x)^5} = x + 16x^2 + 81x^3 + 256x^4 + ...$
5. $\frac{x(x(x(x+26)+66)+26)+1)}{(1-x)^6} = x + 32x^2 + 243x^3 + 1024x^4 + ...$
6. $\frac{x(x+1)(x(x(x+56)+246)+56)+1)}{(1-x)^7} = x + 64x^2 + 3^6x^3 + 4^6x^4 + ...$

As you can see it gets terribly complicated so this time no clear pattern appeared, probably due to the complexity of alternating multiplication with differentiation.

## Explained Programs

### Getting the Matrix



```
matrix Get_Matrix(string P,int z){
    matrix d;
    d=Compute_Automata(d,P,z);
    int m=P.size();
    matrix v(m+1,VI(m+2));
    FOR(i,d.size()){
        FOR(j,d[i].size()) v[i][d[i][j]]++;
    }
    return v;
}
```

Figure 1: Getting the Matrix

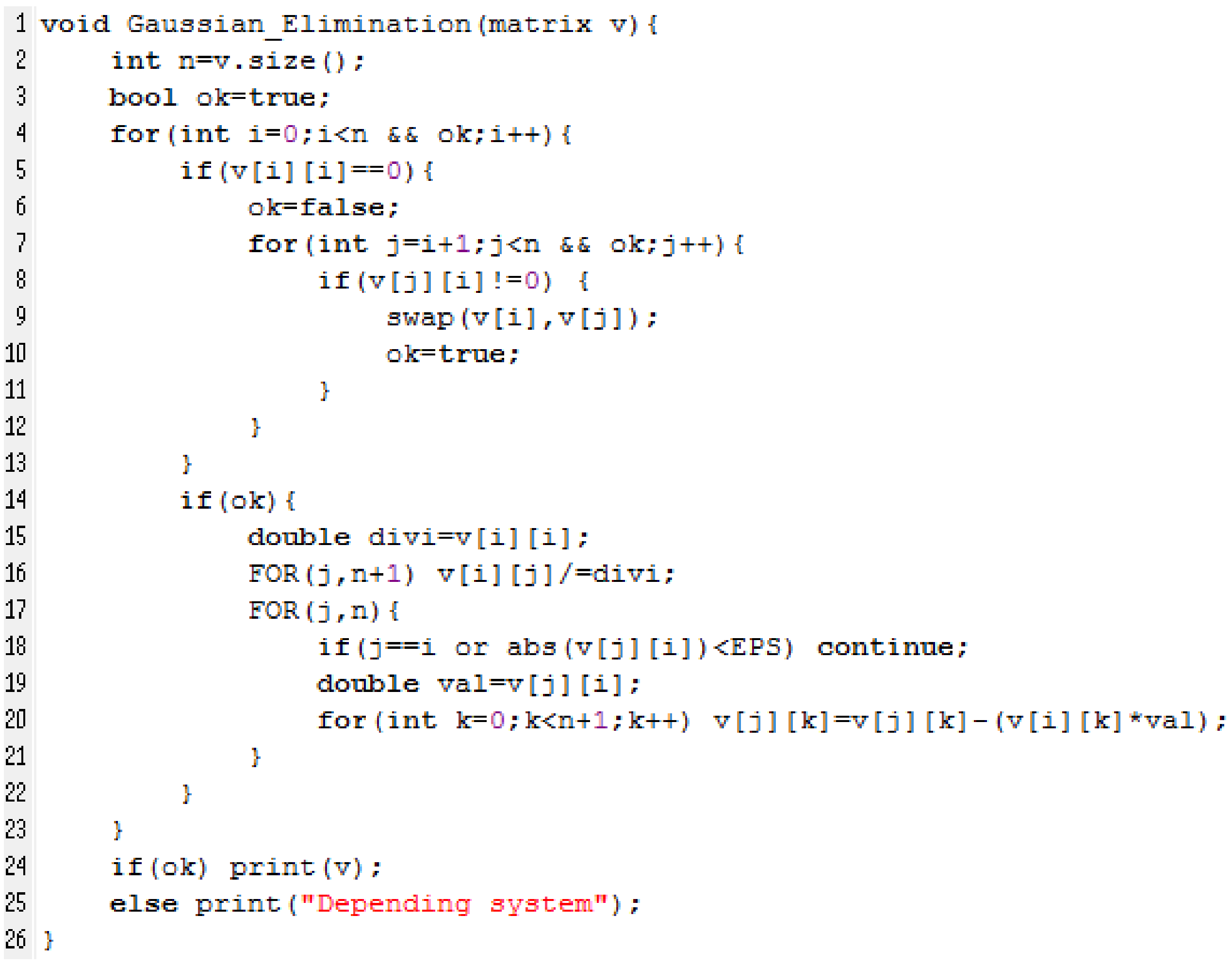
**Explanation line by line of the algorithm for getting the matrix**

1. The beginning of the function: we return a matrix and the input are a string of characters ( the pattern P) and the size of the alphabet (an integer z). Note that it is not the same generating function for the pattern ”001” in a binary alphabet that in a ternary alphabet, therefore we also need z.
2. We declare d, the matrix that will store the automata. The first parameter (the row) describes the state, and the second parameter (the column) describes the character we are introducing.
3. We use the automata algorithm shown in the previous chapter to calculate d.
4. Let m be the size of the pattern P.
5. Declaration of the matrix v, the future result. Note that the dimensions are m+1 rows and m+2 columns since we have m+1 states and it is an extended matrix (so m+1+1=m+2 columns).
6. This means: ”for each row (state) in the transition function d”
7. Given the row ( determined by the previous for ), check every column (possible character in the alphabet ). These two parameters describe an arrow. The start state is clearly the row we are analyzing, and the end state is saved in d[i][j]. Therefore we have to add a z (z++) to v[i][d[i][j]].
8. End of for
9. Return the resultant matrix.
10. End of the function

## Gaussian Elimination

Again, let us analyze it line by line:

- 1. void means that we will not return anything, as the result will be in printed form. The input is the matrix we want to solve.
- 2. Define n as the number of rows of matrix v. ( Note that this is equal to the number of columns -1 )
- 3. Let ok be a boolean that shows if the matrix has a unique solution, i.e. all the equations are linearly independent.
- 4. For each row in the matrix and while the system is described as linearly independent:



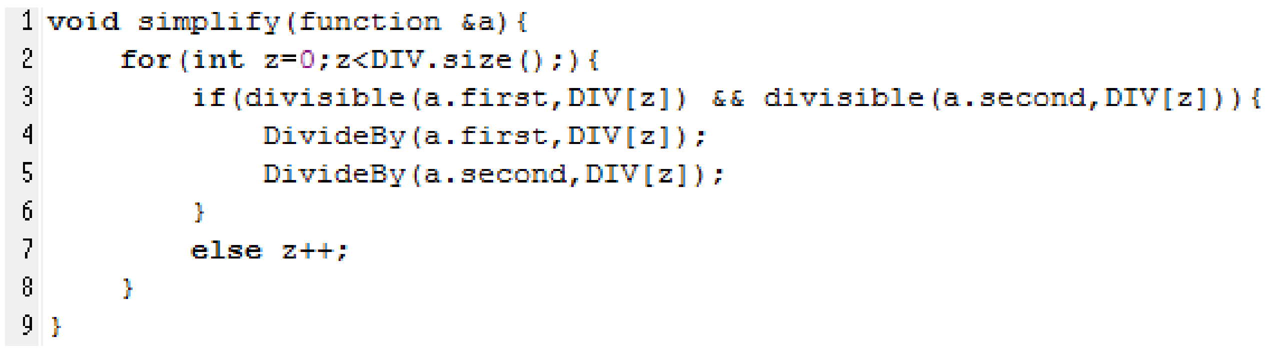
```
1  void Gaussian_Elimination(matrix v){
2      int n=v.size();
3      bool ok=true;
4      for(int i=0;i<n && ok;i++){
5          if(v[i][i]==0){
6              ok=false;
7              for(int j=i+1;j<n && ok;j++){
8                  if(v[j][i]!=0) {
9                      swap(v[i],v[j]);
10                     ok=true;
11                 }
12             }
13         }
14         if(ok){
15             double divi=v[i][i];
16             FOR(j,n+1) v[i][j]/=divi;
17             FOR(j,n){
18                 if(j==i or abs(v[j][i])<EPS) continue;
19                 double val=v[j][i];
20                 for(int k=0;k<n+1;k++) v[j][k]=v[j][k]-(v[i][k]*val);
21             }
22         }
23     }
24     if(ok) print(v);
25     else print("Depending system");
26 }
```

Figure 2: Gaussian Elimination

- 5. If the leading coefficient is 0
- 6. Describe momentarily the system as linearly dependent
- 7-12. Look for the following rows and see if any has coefficient different from 0 in that column. In case there exists, describe the system again as linearly independent and swap rows. All this ensures that either the system is described as linearly dependent or we have a leading coefficient in the diagonal.
- 14. In case we have an existing leading coefficient in the row:
- 15-16. Put the leading coefficient to 1 by dividing the row by the previous leading coefficient.
- 17-18 For every row with a non-zero coefficient in that column (except row i, where we want it to be 1):
- 19-20 Subtract v[j][i] times row i to row j so that we get a 0 in the position v[j][i].

- 21-23 Closing all the opened if's and loops.
- 24-25 Print the solution in case it has one; otherwise print "Depending system"
- 26 End of the function.

### Simplifying fractions of polynomials

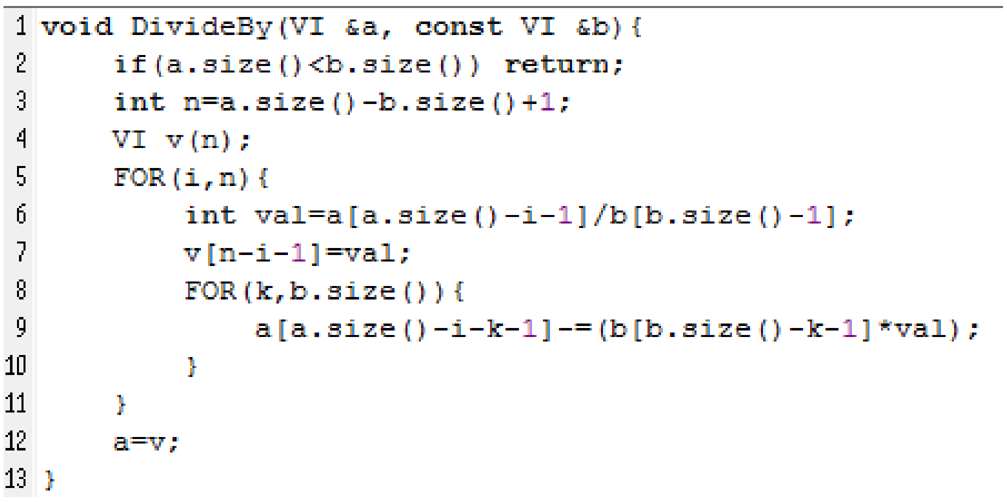


```
void simplify(function &a){
    for(int z=0;z<DIV.size();){
        if(divisible(a.first,DIV[z]) && divisible(a.second,DIV[z])){
            DivideBy(a.first,DIV[z]);
            DivideBy(a.second,DIV[z]);
        }
        else z++;
    }
}
```

Figure 3: Simplification of 2 polynomials

**Explanation of the algorithm:** Given the list of possible divisors (DIV), we check every one of them, if both numerator (a.first) and denominator (a.second) are divisible. Then we divide both of them by this divisor, in case one or both of them are not divisible, we continue to look through the list of divisors. Note that if both of them are divisible in the next loop we will continue to look at the same divisor since it is possible that they share this divisor more than once.
Now let us look the function DivideBy($a, b$), which is very similar to divisible(a,b). Therefore we will only show one of them, also note that the algorithm is exactly the one we perform when doing the division by hand:
**Explanation of the algorithm**

- 1. The function divides a by b, so nothing is returned.
- 2-4. In case the degree of a is smaller than the degree of b, it is already divided; otherwise, create a vector to store the result: a polynomial of degree= deg(a) – deg(b).
- 5. For each coefficient of a
- 6-7. Divide it by the leading coefficient of b and store this value. Note that the degree of this coefficient is $n - i - 1$ because it starts with $n - 1$ and then it goes down 1 for each iteration.
- 8-10. Once known the value multiply each coefficient of b and subtract this multiplication from a.



```
1  void DivideBy(VI &a, const VI &b){
2      if(a.size()<b.size()) return;
3      int n=a.size()-b.size()+1;
4      VI v(n);
5      FOR(i,n){
6          int val=a[a.size()-i-1]/b[b.size()-1];
7          v[n-i-1]=val;
8          FOR(k,b.size()){
9              a[a.size()-i-k-1]-=(b[b.size()-k-1]*val);
10         }
11     }
12     a=v;
13 }
```

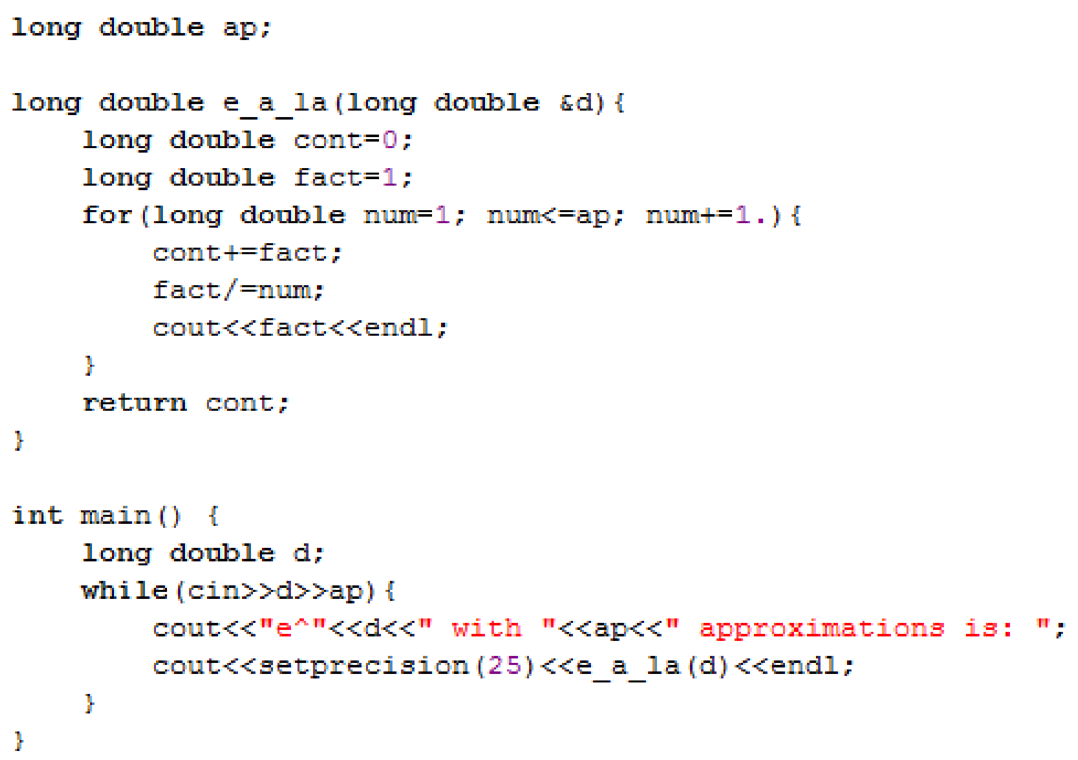
Figure 4: Divide Function

- 11. End of the for started in line 5
- 12. Let a be v, which is $\frac{a}{b}$.
- 13. End of function

## Other Programs

## Axioms of $\mathbb{Z}$

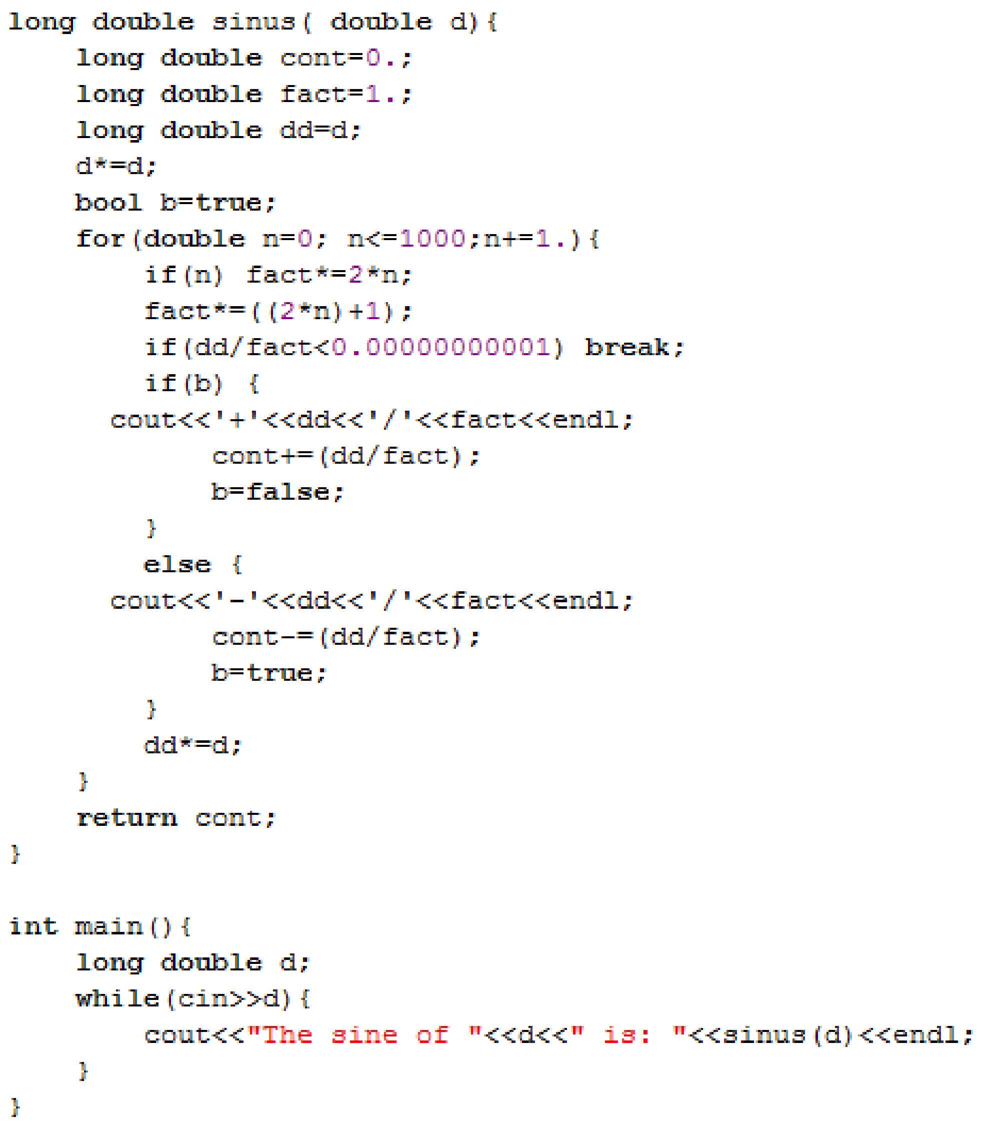
1. If a,b $\in \mathbb{Z}$ then $a + b$ and $a \cdot b$ are $\in \mathbb{Z}$
2. There exists an element $0 \in \mathbb{Z}$ for which $a + 0 = a$ for every $\mathbb{Z}$
3. There exists an element $1 \in \mathbb{Z}$ for which $a \cdot 1 = a$ for every $\mathbb{Z}$
4. $a \cdot (b + c) = a \cdot b + a \cdot c$
5. $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
6. For every $a \in \mathbb{Z}$ there exists a $' - a' \in \mathbb{Z}$ such that $a + (-a) = 0$
7. There are only 3 types of $\mathbb{Z}$: $\mathbb{Z}^+$, 0 and $\mathbb{Z}^-$
8. $-1 \in \mathbb{Z}^-, 1 \in \mathbb{Z}^+$
9. Well Ordering Principle, In a set $S \neq \emptyset$ there exists a smallest element
10. $a = b$ and $b = c \rightarrow a = c$



```
long double ap;

long double e_a_la(long double &d){
     long double cont=0;
     long double fact=1;
     for(long double num=1; num<=ap; num+=1.){
          cont+=fact;
          fact/=num;
          cout<<fact<<endl;
     }
     return cont;
}

int main() {
     long double d;
     while(cin>>d>>ap){
          cout<<"e^"<<d<<" with "<<ap<<" approximations is: ";
          cout<<setprecision(25)<<e_a_la(d)<<endl;
     }
}
```

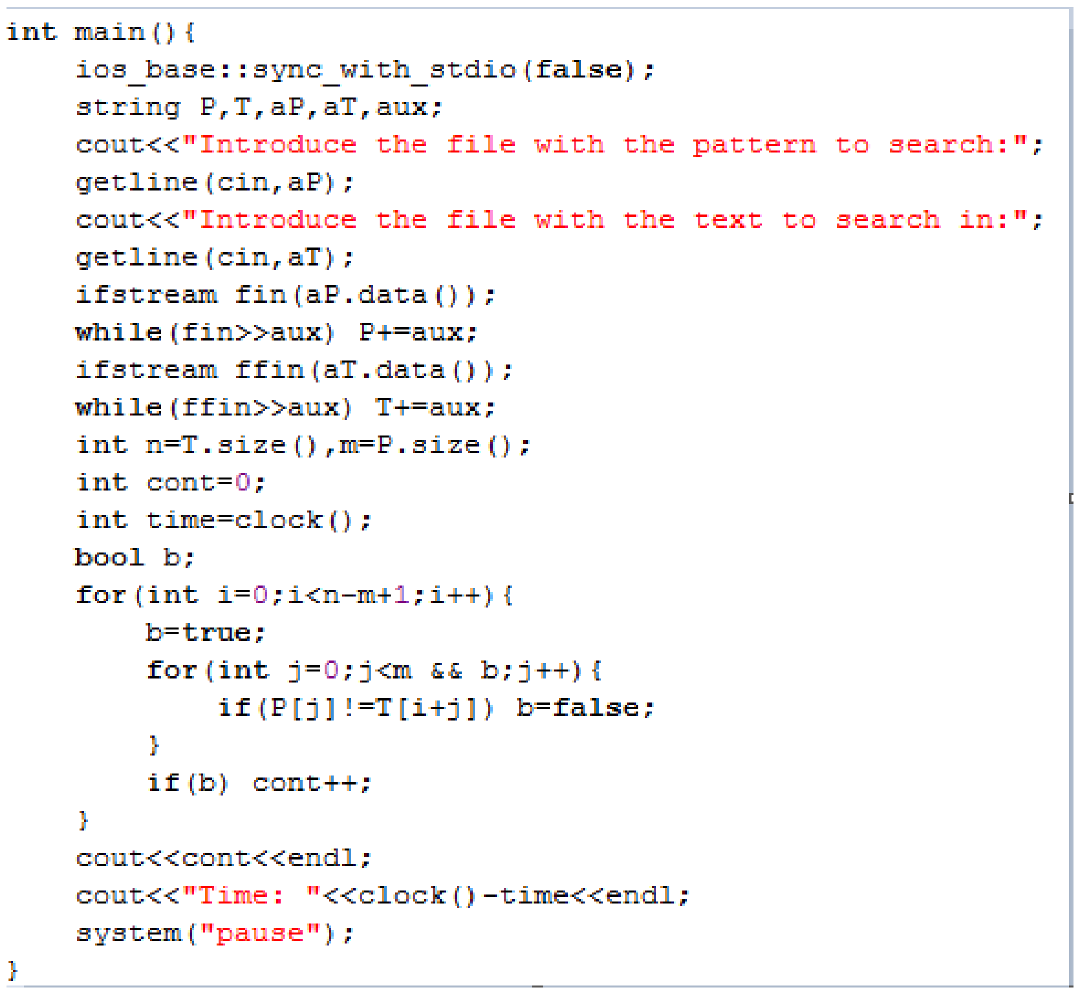
Figure 5: Program for $e^x$

11. $1 \neq 0$



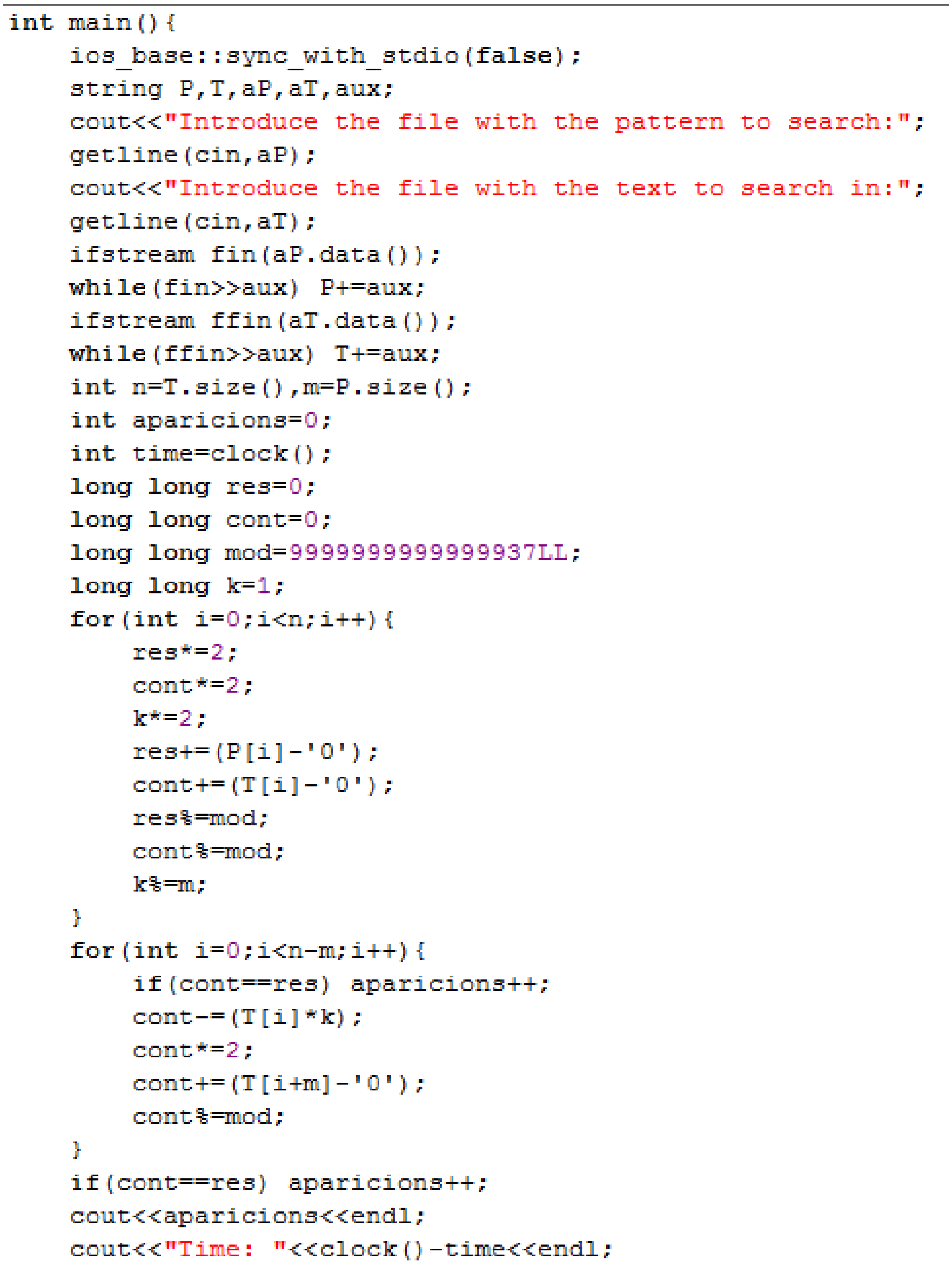
```
long double sinus( double d){
    long double cont=0.;
    long double fact=1.;
    long double dd=d;
    d*=d;
    bool b=true;
    for(double n=0; n<=1000;n+=1.){
        if(n) fact*=2*n;
        fact*=((2*n)+1);
        if(dd/fact<0.00000000001) break;
        if(b) {
      cout<<'+'<<dd<<'/'<<fact<<endl;
            cont+=(dd/fact);
            b=false;
        }
        else {
      cout<<'-'<<dd<<'/'<<fact<<endl;
            cont-=(dd/fact);
            b=true;
        }
        dd*=d;
    }
    return cont;
}

int main(){
    long double d;
    while(cin>>d){
        cout<<"The sine of "<<d<<" is: "<<sinus(d)<<endl;
    }
}
```

Figure 6: Program for $\sin(x)$



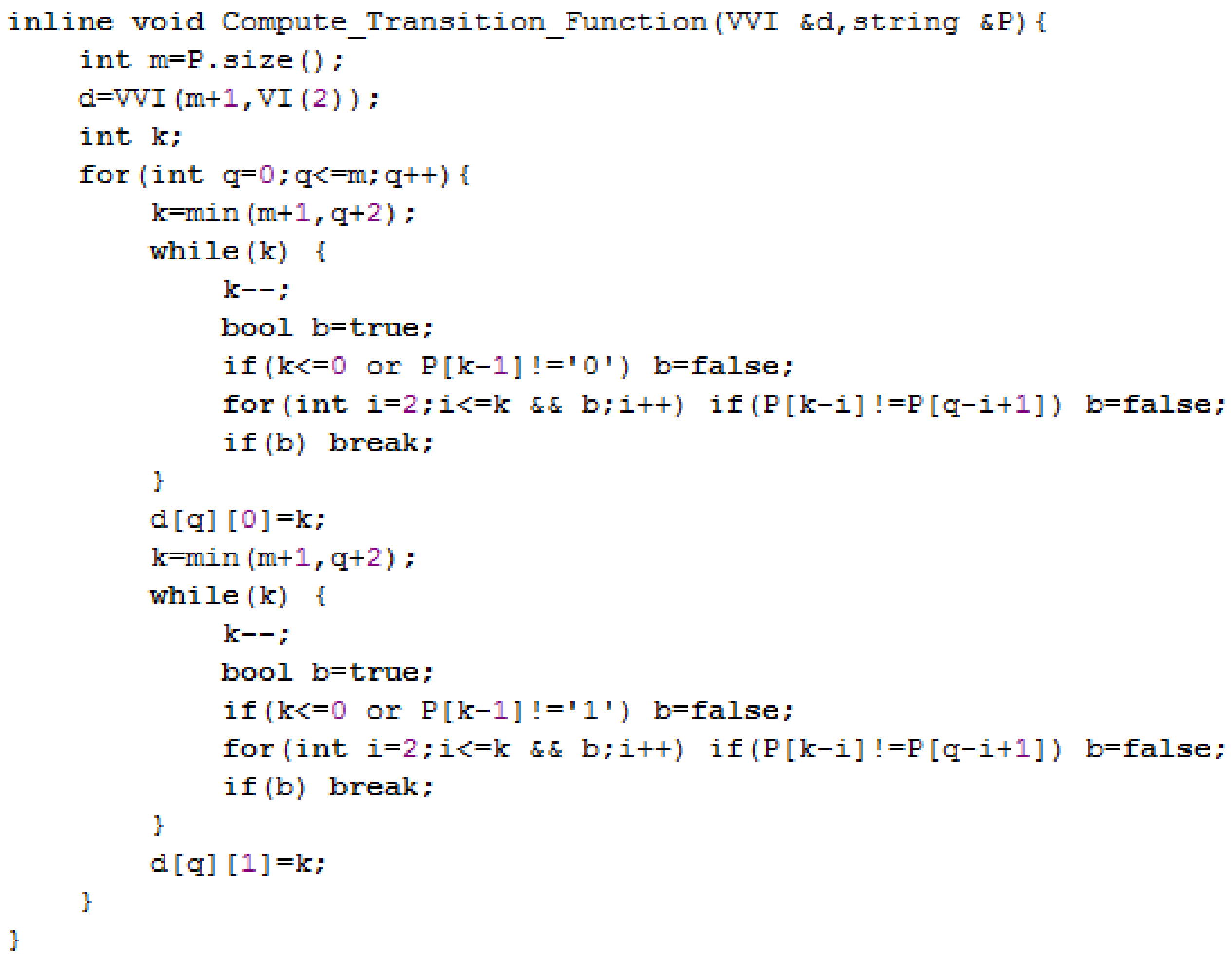
```
int main(){
    ios_base::sync_with_stdio(false);
    string P,T,aP,aT,aux;
    cout<<"Introduce the file with the pattern to search:";
    getline(cin,aP);
    cout<<"Introduce the file with the text to search in:";
    getline(cin,aT);
    ifstream fin(aP.data());
    while(fin>>aux) P+=aux;
    ifstream ffin(aT.data());
    while(ffin>>aux) T+=aux;
    int n=T.size(),m=P.size();
    int cont=0;
    int time=clock();
    bool b;
    for(int i=0;i<n-m+1;i++){
        b=true;
        for(int j=0;j<m && b;j++){
            if(P[j]!=T[i+j]) b=false;
        }
        if(b) cont++;
    }
    cout<<cont<<endl;
    cout<<"Time: "<<clock()-time<<endl;
    system("pause");
}
```

Figure 7: Naive string matcher



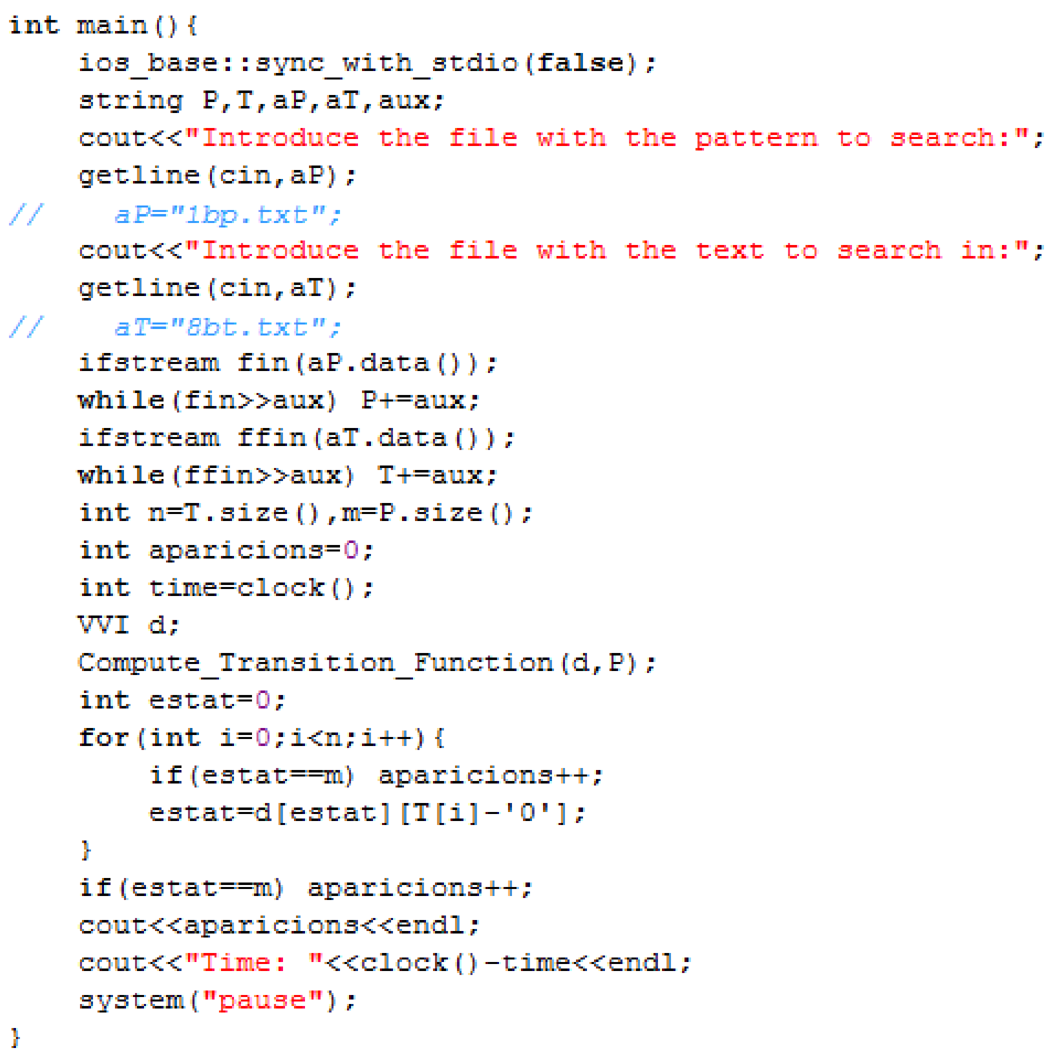
```
int main(){
    ios_base::sync_with_stdio(false);
    string P,T,aP,aT,aux;
    cout<<"Introduce the file with the pattern to search:";
    getline(cin,aP);
    cout<<"Introduce the file with the text to search in:";
    getline(cin,aT);
    ifstream fin(aP.data());
    while(fin>>aux) P+=aux;
    ifstream ffin(aT.data());
    while(ffin>>aux) T+=aux;
    int n=T.size(),m=P.size();
    int aparicions=0;
    int time=clock();
    long long res=0;
    long long cont=0;
    long long mod=9999999999999937LL;
    long long k=1;
    for(int i=0;i<n;i++){
        res*=2;
        cont*=2;
        k*=2;
        res+=(P[i]-'0');
        cont+=(T[i]-'0');
        res%=mod;
        cont%=mod;
        k%=m;
    }
    for(int i=0;i<n-m;i++){
        if(cont==res) aparicions++;
        cont-=(T[i]*k);
        cont*=2;
        cont+=(T[i+m]-'0');
        cont%=mod;
    }
    if(cont==res) aparicions++;
    cout<<aparicions<<endl;
    cout<<"Time: "<<clock()-time<<endl;
```

Figure 8: Rabin-Karp algorithm



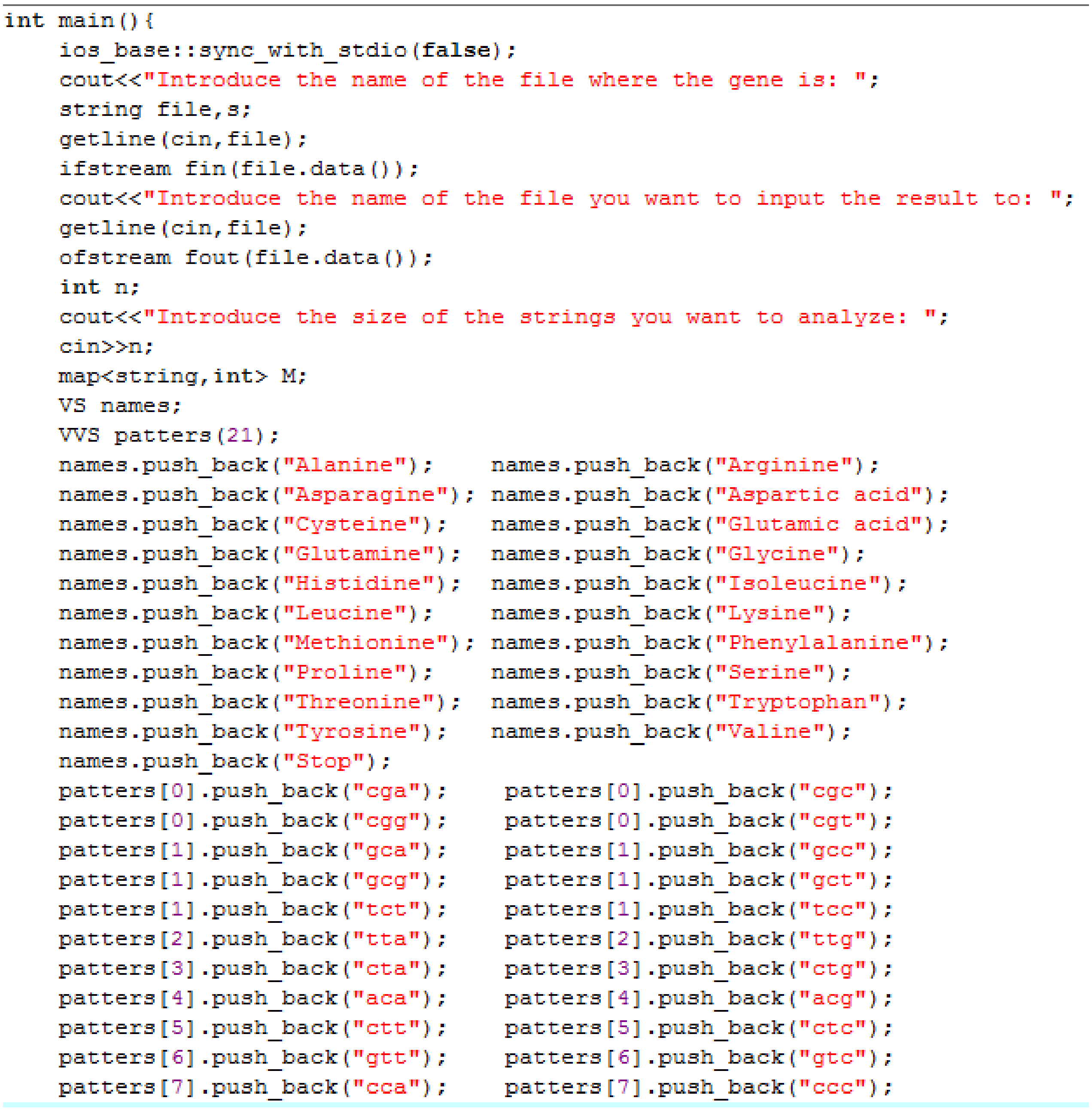
```
inline void Compute_Transition_Function(VVI &d,string &P){
    int m=P.size();
    d=VVI(m+1,VI(2));
    int k;
    for(int q=0;q<=m;q++){
        k=min(m+1,q+2);
        while(k) {
            k--;
            bool b=true;
            if(k<=0 or P[k-1]!='0') b=false;
            for(int i=2;i<=k && b;i++) if(P[k-i]!=P[q-i+1]) b=false;
            if(b) break;
        }
        d[q][0]=k;
        k=min(m+1,q+2);
        while(k) {
            k--;
            bool b=true;
            if(k<=0 or P[k-1]!='1') b=false;
            for(int i=2;i<=k && b;i++) if(P[k-i]!=P[q-i+1]) b=false;
            if(b) break;
        }
        d[q][1]=k;
    }
}
```

Figure 9: Automata Transition Function



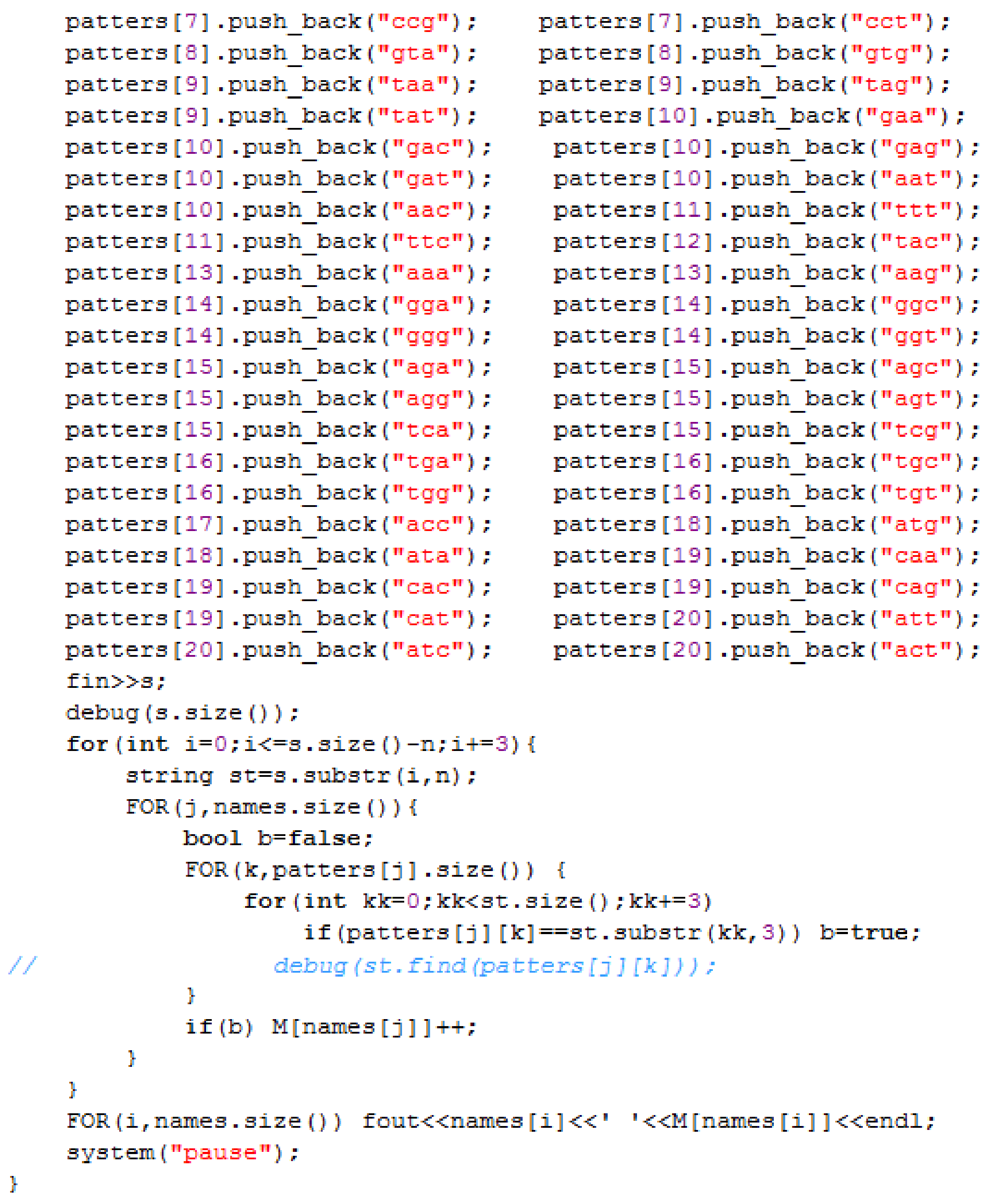
```
int main(){
    ios_base::sync_with_stdio(false);
    string P,T,aP,aT,aux;
    cout<<"Introduce the file with the pattern to search:";
    getline(cin,aP);
//    aP="1bp.txt";
    cout<<"Introduce the file with the text to search in:";
    getline(cin,aT);
//    aT="8bt.txt";
    ifstream fin(aP.data());
    while(fin>>aux) P+=aux;
    ifstream ffin(aT.data());
    while(ffin>>aux) T+=aux;
    int n=T.size(),m=P.size();
    int aparicions=0;
    int time=clock();
    VVI d;
    Compute_Transition_Function(d,P);
    int estat=0;
    for(int i=0;i<n;i++){
        if(estat==m) aparicions++;
        estat=d[estat][T[i]-'0'];
    }
    if(estat==m) aparicions++;
    cout<<aparicions<<endl;
    cout<<"Time: "<<clock()-time<<endl;
    system("pause");
}
```

Figure 10: Automata simulation



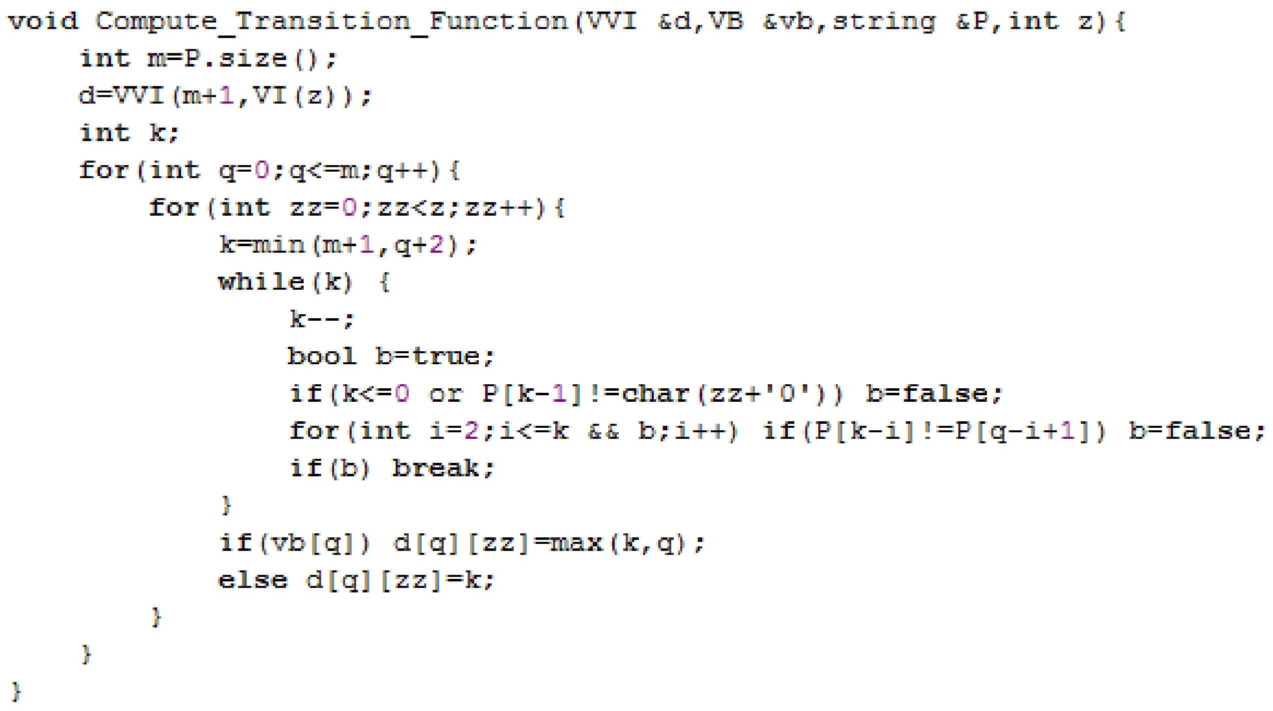
```
int main(){
    ios_base::sync_with_stdio(false);
    cout<<"Introduce the name of the file where the gene is: ";
    string file,s;
    getline(cin,file);
    ifstream fin(file.data());
    cout<<"Introduce the name of the file you want to input the result to: ";
    getline(cin,file);
    ofstream fout(file.data());
    int n;
    cout<<"Introduce the size of the strings you want to analyze: ";
    cin>>n;
    map<string,int> M;
    VS names;
    VVS patters(21);
    names.push_back("Alanine");    names.push_back("Arginine");
    names.push_back("Asparagine"); names.push_back("Aspartic acid");
    names.push_back("Cysteine");   names.push_back("Glutamic acid");
    names.push_back("Glutamine");  names.push_back("Glycine");
    names.push_back("Histidine");  names.push_back("Isoleucine");
    names.push_back("Leucine");    names.push_back("Lysine");
    names.push_back("Methionine"); names.push_back("Phenylalanine");
    names.push_back("Proline");    names.push_back("Serine");
    names.push_back("Threonine");  names.push_back("Tryptophan");
    names.push_back("Tyrosine");   names.push_back("Valine");
    names.push_back("Stop");
    patters[0].push_back("cga");    patters[0].push_back("cgc");
    patters[0].push_back("cgg");    patters[0].push_back("cgt");
    patters[1].push_back("gca");    patters[1].push_back("gcc");
    patters[1].push_back("gcg");    patters[1].push_back("gct");
    patters[1].push_back("tct");    patters[1].push_back("tcc");
    patters[2].push_back("tta");    patters[2].push_back("ttg");
    patters[3].push_back("cta");    patters[3].push_back("ctg");
    patters[4].push_back("aca");    patters[4].push_back("acg");
    patters[5].push_back("ctt");    patters[5].push_back("ctc");
    patters[6].push_back("gtt");    patters[6].push_back("gtc");
    patters[7].push_back("cca");    patters[7].push_back("ccc");
```

Figure 11: Searcher in genes part 1



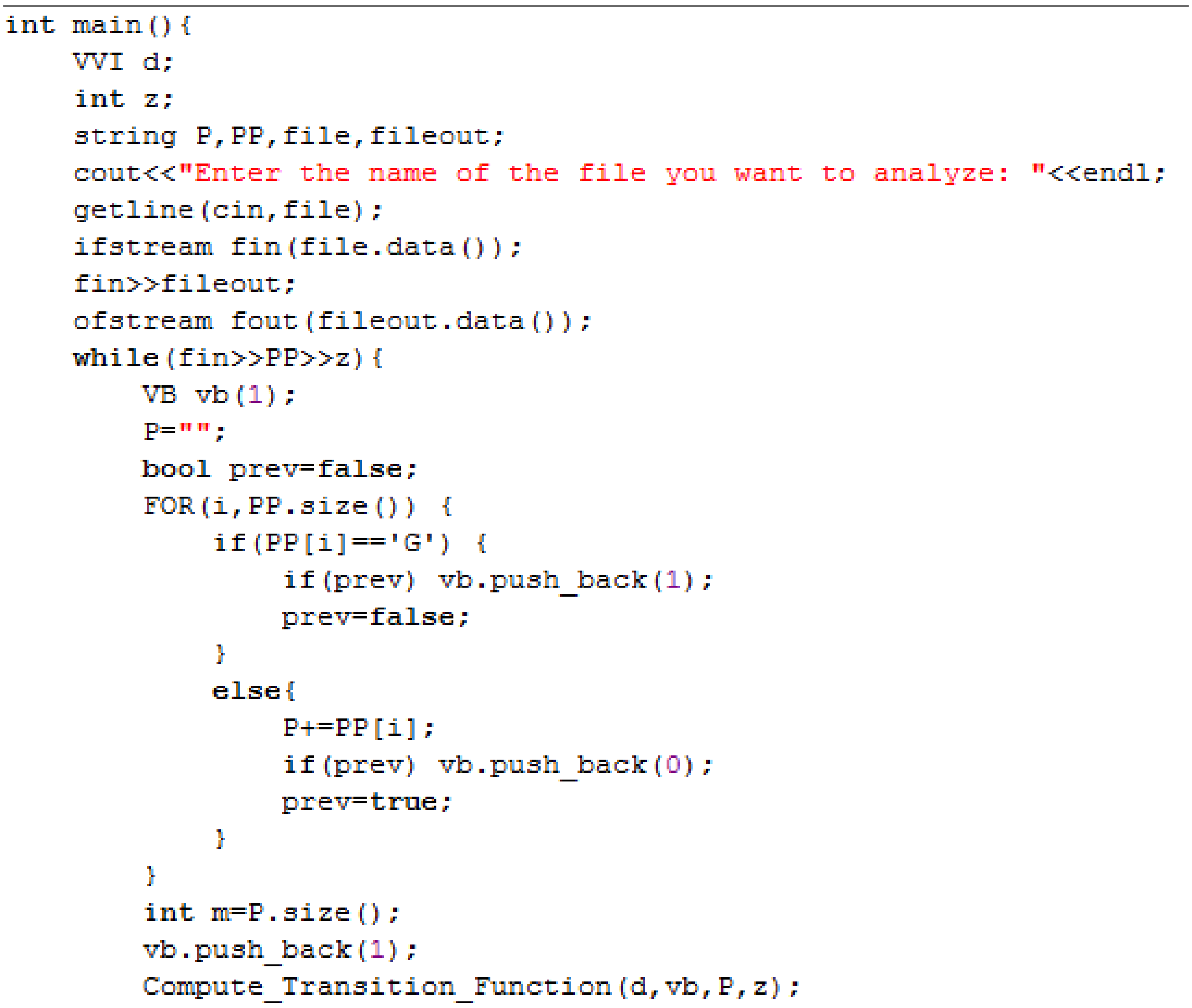
```
    patters[7].push_back("ccg");    patters[7].push_back("cct");
    patters[8].push_back("gta");    patters[8].push_back("gtg");
    patters[9].push_back("taa");    patters[9].push_back("tag");
    patters[9].push_back("tat");    patters[10].push_back("gaa");
    patters[10].push_back("gac");    patters[10].push_back("gag");
    patters[10].push_back("gat");    patters[10].push_back("aat");
    patters[10].push_back("aac");    patters[11].push_back("ttt");
    patters[11].push_back("ttc");    patters[12].push_back("tac");
    patters[13].push_back("aaa");    patters[13].push_back("aag");
    patters[14].push_back("gga");    patters[14].push_back("ggc");
    patters[14].push_back("ggg");    patters[14].push_back("ggt");
    patters[15].push_back("aga");    patters[15].push_back("agc");
    patters[15].push_back("agg");    patters[15].push_back("agt");
    patters[15].push_back("tca");    patters[15].push_back("tcg");
    patters[16].push_back("tga");    patters[16].push_back("tgc");
    patters[16].push_back("tgg");    patters[16].push_back("tgt");
    patters[17].push_back("acc");    patters[18].push_back("atg");
    patters[18].push_back("ata");    patters[19].push_back("caa");
    patters[19].push_back("cac");    patters[19].push_back("cag");
    patters[19].push_back("cat");    patters[20].push_back("att");
    patters[20].push_back("atc");    patters[20].push_back("act");
    fin>>s;
    debug(s.size());
    for(int i=0;i<=s.size()-n;i+=3){
        string st=s.substr(i,n);
        FOR(j,names.size()){
            bool b=false;
            FOR(k,patters[j].size()) {
                for(int kk=0;kk<st.size();kk+=3)
                    if(patters[j][k]==st.substr(kk,3)) b=true;
//                    debug(st.find(patters[j][k]));
            }
            if(b) M[names[j]]++;
        }
    }
    FOR(i,names.size()) fout<<names[i]<<' '<<M[names[i]]<<endl;
    system("pause");
}
```

Figure 12: Searcher in genes part 2



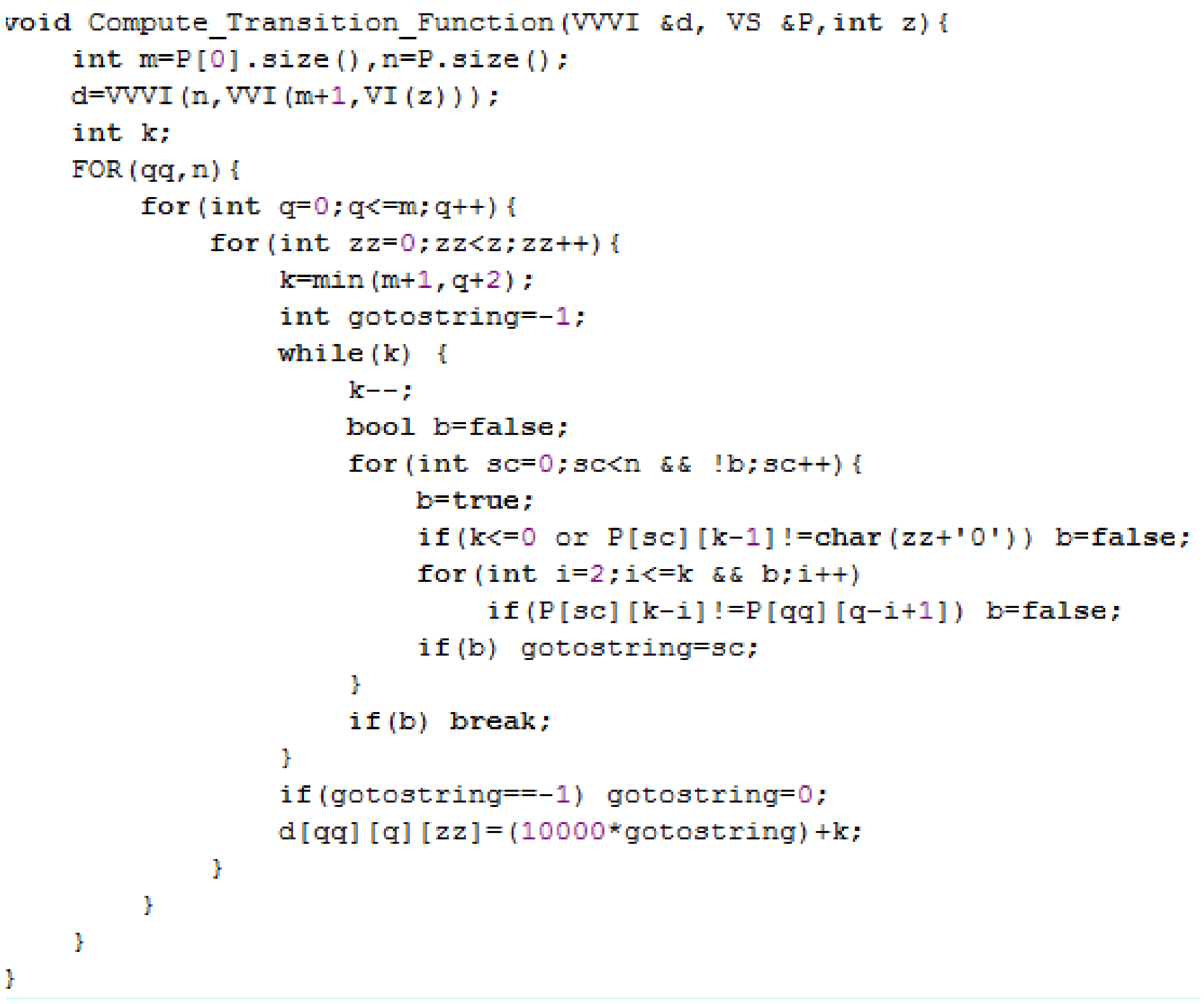
```
void Compute_Transition_Function(VVI &d,VB &vb,string &P,int z){
    int m=P.size();
    d=VVI(m+1,VI(z));
    int k;
    for(int q=0;q<=m;q++){
        for(int zz=0;zz<z;zz++){
            k=min(m+1,q+2);
            while(k) {
                k--;
                bool b=true;
                if(k<=0 or P[k-1]!=char(zz+'0')) b=false;
                for(int i=2;i<=k && b;i++) if(P[k-i]!=P[q-i+1]) b=false;
                if(b) break;
            }
            if(vb[q]) d[q][zz]=max(k,q);
            else d[q][zz]=k;
        }
    }
}
```

Figure 13: Transition function for patterns with gaps



```
int main(){
    VVI d;
    int z;
    string P,PP,file,fileout;
    cout<<"Enter the name of the file you want to analyze: "<<endl;
    getline(cin,file);
    ifstream fin(file.data());
    fin>>fileout;
    ofstream fout(fileout.data());
    while(fin>>PP>>z){
        VB vb(1);
        P="";
        bool prev=false;
        FOR(i,PP.size()) {
            if(PP[i]=='G') {
                if(prev) vb.push_back(1);
                prev=false;
            }
            else{
                P+=PP[i];
                if(prev) vb.push_back(0);
                prev=true;
            }
        }
        int m=P.size();
        vb.push_back(1);
        Compute_Transition_Function(d,vb,P,z);
```

Figure 14: Special reading for patterns with gaps



```
void Compute_Transition_Function(VVVI &d, VS &P,int z){
    int m=P[0].size(),n=P.size();
    d=VVVI(n,VVI(m+1,VI(z)));
    int k;
    FOR(qq,n){
        for(int q=0;q<=m;q++){
            for(int zz=0;zz<z;zz++){
                k=min(m+1,q+2);
                int gotostring=-1;
                while(k) {
                    k--;
                    bool b=false;
                    for(int sc=0;sc<n && !b;sc++){
                        b=true;
                        if(k<=0 or P[sc][k-1]!=char(zz+'0')) b=false;
                        for(int i=2;i<=k && b;i++)
                            if(P[sc][k-i]!=P[qq][q-i+1]) b=false;
                        if(b) gotostring=sc;
                    }
                    if(b) break;
                }
                if(gotostring==-1) gotostring=0;
                d[qq][q][zz]=(10000*gotostring)+k;
            }
        }
    }
}
```

Figure 15: Transition function for the *either* pattern problem

# Bibliography

[1] CARRERAS, Núria: *Aproximació informàtica a la genètica molecular i clàssica de la fibrosi quística* Aula Escola Europea 2009

[2] CORMEN, Thomas H.; LEISERON, Charles E. ; RIVEST, Ronald L.; STEIN, Clifford: *Introduction to Algorithms* Second Edition, MIT Press, 1180 pages.

[3] DEHMER, Matthias: *Structural Analysis of Complex Networks* Birkhäuser, 2011, Google Books Version (Consulted January 2011)

[4] DIOFANTO: *Aritmética* Spanish translation done by Javier García Sanz in *Dios creó los números* pages 209-253.

[5] EUCLIDES: *Elementos* Spanish translation and anotations by María Luisa Puertas Castaños. Found in *Dios creó los números* third edition pages 7-76.

[6] GAUSS, Carl Friedrich: *Disquisiciones aritméticas* spanish translation by Javier García Sanz in *Diós creó los números*, third edition, pages 501-553.

[7] GRANE, Josep and others; *Llibre de Preparació per a l'Olimpíada matemàtica* 339 pages; $www.iecat.net/institucio/societats/SCMatematiques/Publicacions/pubs_1/sessions_olimpiada.pdf$ (multiple visits)

[8] HERSTEIN, I.N.: *Abstract Algebra.*Third edition, John Wiley and sons, 249 pages

[9] KROB, D.; MIKHALEV, A.A.; MIKHALEV, A.V.: *Formal Power Series and Algebraic Combinatorics* Moscow State University, 2000

[10] LANG, Serge: *Algebra*

[11] LOZANO, Oriol: *Nombres primers: anàlisi de la complexitat* Aula Escola Europea 2007

[12] PEYA, Josep; SÁNCHEZ, Antoni: *Les cadenes de Markov* Aula Escola Europea 2010

[13] RAVEN; JOHNSON; MASON; LOSOS; SINGER: *Biology* , McGraw Hill, ninth edition

[14] STANLEY, Richard P.: *Enumerative Combinatorics Vol.1*

[15] STANLEY, Richard P.: *Enumerative Combinatorics Vol.2*

[16] STEWARD, James: *Calculus* Brooks/Cole,Third edition. 1138 pages.

[17] http://mathworld.wolfram.com/ ( multiple visits )

[18] http://www.wolframalpha.com/ ( multiple visits )

[19] $http : /www.ncbi.nlm.nih.govnuccoreNC_0 00007?report = genbank\&from = 117120017\&to = 117308719$

[20] http://www.britannica.com/EBchecked/topic/127341/combinatorics

[21] Gimp program; downloaded from: http://www.gimp.org/ (December 2010)

[22] Jean Bovet *Visual Automata Simulator* program(2004-06), used to draw the automata manually; http://www.cs.usfca.edu/ jbovet/vas.html

[23] http://www.google.com/finance

[24] *Part of mouse genome:* $http : //www.ncbi.nlm.nih.gov/nuccore/NC_0 00067.5$ (January 2011)

[25] *Part of human genome:* http://www.ncbi.nlm.nih.gov/nuccore/224183340?report=fasta

[26] *Problem that can be solved using Markov Chains and Generating Functions:* $http : /www.elpais.comvideossociedadhormigaamenazadaelpepusoc20110325elpepusoc_1 Ves$