INTRODUCTION

Transportation, traffic, communication and energy networks form the backbone of our modern society. To deal with the uncertainty, variation, unpredictability, size and complexity inherent in these networks, we need to develop radically new ways of thinking. The ultimate goal is to build self-organising and intelligent networks. The NWO-funded Gravitation programme NETWORKS started in the Summer of 2014 and covers a broad range of topics dealing with stochastic and algorithmic aspects of networks.

In March 2022 the fourth "NETWORKS goes to school" event was organised. The aim of the event is to provide secondary education students and teachers a first mathematical introduction on network science. This book collects the material realised for the "NETWORKS goes to school" event.

The content of this book is intended for secondary education students and teachers, and aims to provide a first mathematical introduction on network science. In Chapter 1 all the necessary background material that is required for Chapter 2 is presented. In Section 2.1, we introduce the concept of computational complexity and we show how to analyse the complexity of an algorithm. In Section 2.2 we discuss one of the most fascinating questions in modern mathematics, whether **P**$\neq$**NP**, and we show a concrete example of a so-called **NP**-complete problem from graph theory. In Section 2.3.1 we discuss the First Mover Advantage, a phenomenon occurring in sports events, for example in the penalty series when a football match ends with a draw. We show that, in the sudden death phase, when one team always shoots first they have an advantage against the second team. An alternative method to make sudden death more honest is also shown. Chapter 3 contains exercises on these two topics and in Chapter 4 we provide the corresponding solutions. Chapters 2, 3 and 4 were written with the help of dr. Giulia Bernardini (University of Trieste) and Roel Lampers (Eindhoven University of Technology).

For more information and the books of the first two masterclasses "NETWORKS goes to school", please visit www.networkpages.nl/category/networks-goes-to-school/

On behalf of the NETWORKS programme,

the organising committee of "NETWORKS goes to school"
Jan-Pieter Dorsman (University of Amsterdam)
Nicos Starreveld (University of Amsterdam)

Contents

1.1. Basic notation

We start by introducing some notation we will use in the sequel:

(1) $\mathbb{N}$ for the set of natural numbers, that is $\mathbb{N} = \{1, 2, 3, \cdots\}$;
(2) $\mathbb{Z}$ for the set of integer numbers, that is $\mathbb{Z} = \{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\}$.
(3) $\mathbb{R}$ for the set of real numbers, that is all integer numbers and all the decimal numbers between them.
(4) We will use the symbol $\leq$ when we want to say "less or equal to". For example, $a \leq b$ means that a is less or equal to b.
(5) We will often work with subsets of the natural numbers. For a set I, we write I^c or $\mathbb{N}/I$ for it's complement. For example, if $I = \{2, 4, 6, \ldots\}$ contains all even numbers, then I^c contains all odd numbers.

1.2. Algorithms

Mobile phones, ATMs, modern cars, televisions, e-readers: none of them would work without software. The heart of software is formed by algorithms: step-by-step procedures to perform given tasks. Algorithms can be executed by computers, but also by persons.

Algorithm

An **algorithm** is a step-by-step procedure to perform a given task. Algorithms can be executed by computers, but also by persons.

For instance, suppose you hold twenty cards in your hand, each with a different number on them. Your task is to create a stack containing the cards in sorted order, with the lowest numbered card at the bottom and the highest numbered card on top of the stack. A simple algorithm for this task is the following. Go over the cards in your hand one by one, to find the lowest numbered card. Put this card as the bottom card on the stack. Then go over the remaining cards in your hand, to find the next lowest-numbered card, and put it on top of the previously selected card. Repeat this procedure until all cards are on the stack.

In order for an algorithm to be executed by a computer it has to be implemented in computer code, typically in a computer language like Java or C++. When the description of the algorithm is sufficiently precise, implementing it is not very difficult for a skilled programmer. Thus, when you want a certain task to be performed by computer the challenge lies in coming up with a good algorithm.

1.2.1. Describing algorithms: pseudocode

As mentioned before an algorithm is a precisely described step-by-step plan to perform a certain task. Generally an algorithm will eventually be converted into software to be to be run on a computer. It is therefore important that the description of the algorithm is precise enough so that a programmer can convert it into the desired programming language. We will therefore describe algorithms in **pseudocode**.

Let's look at an example. Suppose we need an algorithm that searches for a given number in a large set of numbers. This large set of numbers is stored in a so-called **array**: a row of numbers, just like in a table. The numbers are numbered from 1 up to n. If we call the array A, then we denote the first number with $A[1]$, the second number with $A[2]$, and so on. So the i-th number is in $A[i]$; the value i becomes the **index** of the number called. The whole set of numbers, $A[1], \ldots, A[n]$ is simply written as $A[1 \ldots n]$.

Index

The **index** of an element in an array is the position it holds in the array from left to right.

To see if the number q occurs in the given set of numbers, we can now simply run through the array and at each number check if it is equal to q. If **YES**, then we can stop, otherwise we will continue to the next number. If we checked all numbers without finding q, then we can conclude that q does not occur in the given set of numbers.

To the array A is checked by using an **auxiliary variable**. This variable, which we call j here, gives the index of the number we are looking at at that moment. Iterating through the array happens so by increasing j each time by 1. Assigning a value to a variable becomes notated as with the symbol $\leftarrow$. For example, if we want to give j the value 1, then we write $j \leftarrow 1$ (pronounced: "j becomes 1"), and if we want to increase j by 1 we write $j \leftarrow j + 1$ (pronounce: "j becomes j + 1"). In pseudocode, our search algorithm now looks like this.

Algorithm 1 Linear Search

1: **INPUT:** An array $A[1, \ldots, n]$ of numbers and a number q.
2: **OUTPUT:** An index i such that $A[i] = q$; or **FAIL** if no such index exists.
3: $j \leftarrow 1$;
4: **while** $j \leq n$ **do**
5: **if** $A[j] = q$ **then**
6: **return** j;
7: **else**
8: $j \leftarrow j + 1$;
9: **if** $j = n + 1$ **then return** : **FAIL**;

The input of the algorithm is indicated in parentheses, just like a math function: just as you write $f(x)$ for a function f where you put a number x, so you write *LinearSearch*(A, q) for a LinearSearch-algorithm where you put an array A and a number q as inputs. Because it's not immediately clear what the input represents and what the result of the algorithm should be, this is indicated at the beginning of the algorithm.

The description of the algorithm itself consists of simple commands (e.g. "$i \leftarrow i + 1$" or "return i") and tests (e.g. "$i \leq n$"). Moreover are there repetition commands of the form

while: some condition
do:...

or selection commands like

if: some condition
then:...
else:...

If nothing needs to be done in the *else* part, this part is omitted.

1.3. Graph theory

An intuitive definition of a network would be a 'collection of objects that are interconnected in some way'. Think for example of a collection of people, who can be interconnected by friendships; or a collection of cities, which can be interconnected by roads. To make this idea precise, we turn to graph theory.

Graph

A **graph** is a pair $G = (V, E)$, where
- V is the set of nodes or vertices;
- E is the set of edges, connecting the nodes.

Typically, we number the nodes from $\{1, 2, 3, \ldots, \}$. We denote an edge between two nodes i and j by $\{i, j\}$. To define a graph, we can write down the sets V and E.

EXAMPLE 1.3.1. Consider

$$V = \{1, 2, 3, 4, 5, 6\}, \quad E = \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}.$$

Then $G = (V, E)$ is a graph with six nodes and seven edges.

It may be very useful to have a graphical representation of a graph. We do this by typically drawing nodes as a circle with a label in it, and edges as a line between nodes. However,

you are free to choose any representation you may like! In fact, the location of the nodes is also arbitrary, it only matters the way in which the edges connect the nodes together.

EXAMPLE 1.3.1 (Continued). In Figure 1.3.1 we see two ways in which the graph G can be drawn.

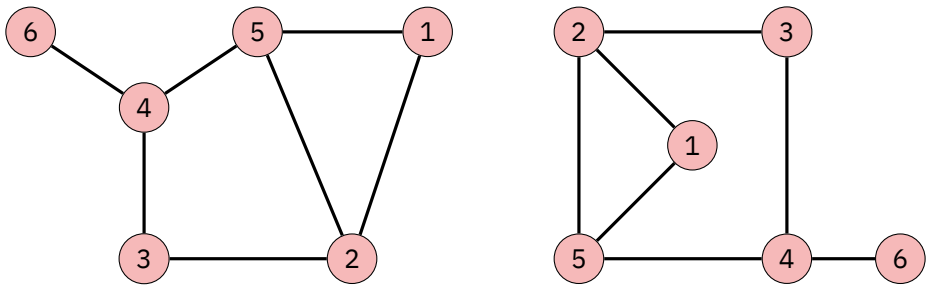


Figure 1.3.1. Two different representations of the graph in Example 1.3.1.

In the following two sections we will discuss two problems from graph theory, the first one is finding an Euleriean path and the second one is finding a Hamiltonian path. These two problems may seem similar to each other at first sight, but later on you will see that one is very easy to solve while the other is very difficult!

1.3.1. Eulerian path

In the Eulerian path problem we want, given a graph $G(V, E)$, to find out whether there is a way of visiting all the edges of G exactly once with an unbroken path. In such a path vertices may be visited more than once.

EXAMPLE 1.3.1. In Figure 1.3.2 below you can see an Eulerian path in a graph on 6 vertices.

You can thus see that an Eulerian path depends on the starting point. For example, if you start from vertex 4 in the graph above, you will quickly notice that it is not possible to find a path using all edges exactly once! In Section 3.1 you will find more exercises to work on.

1.3.2. Hamiltonian path

In the Hamiltonian path problem we want, given a graph $G(V, E)$, to find out whether there is a way of visiting all the vertices of G exactly once with an unbroken path. The difference with the Eulerian path problem is that now not all edges are necessarily used, moreover in the Eulerian path problem some vertices could be used twice.

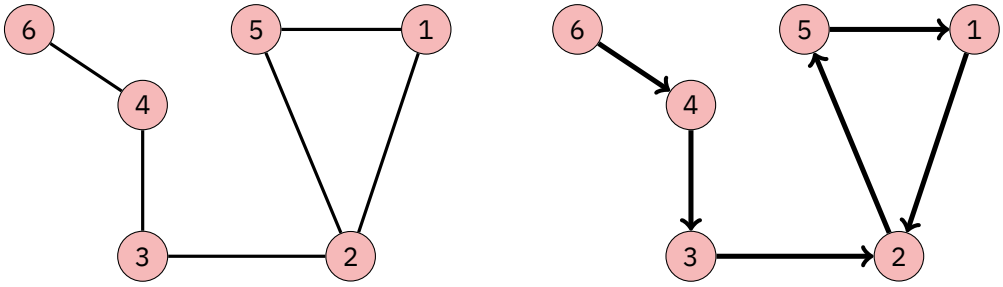


Figure 1.3.2. (Left) A graph with 6 vertices. (Right) An Eulerian path in the graph starting from vertex 6.

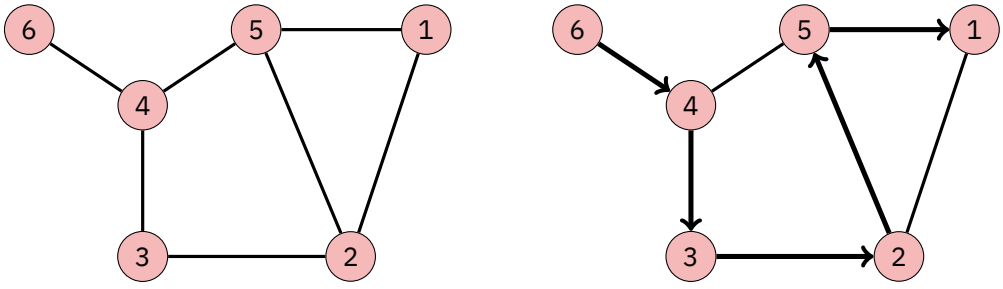


Figure 1.3.3. (Left) A graph with 6 vertices. (Right) A Hamiltonian path in the graph starting from vertex 6.

EXAMPLE 1.3.1. In Figure 1.3.3 below you can see a Hamiltonian path in a graph on 6 vertices.

A Hamiltonian path thus also depends on the starting vertex. For example, if you start from vertex 4 in the graph above you will quickly notice that it is not possible to find a path visiting all vertices exactly once! In order to visit vertex 6 the path needs to visit vertex 4 twice. In Section 3.1 you will find more exercises to work on.

1.4. About models, sums and products

In this section we discuss some concepts will be needed later in Section 2.3 on the mathematics behind penalty shooting. In this section we will discuss what is a mathematical model, and how to compactly write down sums and products.

1.4.1. Mathematical Models

To study any kind of system or real-life situation we first have to construct a mathematical model. To illustrate what a mathematical model is, you can think of a toy car of a Ferrari (which is also called a *model car*). Such a toy car is not precisely like the Ferrari, since it does not contain a working engine, is made of different material, is much smaller, and so on. However, it does give you a good idea of the shape, how it looks when it is driving, and how it compares to other toy cars. As another example, architects make models of their buildings on a small scale (also called a *scale model*) to study how they would look, how much light will enter the building, how much material is needed, and so on. In a similar way, mathematical models describe a real-life phenomenon, using mathematical concepts and language. The model will not resemble reality perfectly, but can be used to learn from.

1.4.2. On sums and products

Mathematicians always want to write down mathematics as compactly as possible. But the notation used should also be clear and representative of what it describes. Typical notation you encounter when doing mathematics involves:

(1) the symbol used for summation: $\sum$;
(2) the symbol used for a product: $\prod$.

Below we show how these symbols are used to describe a sum and a product respectively. Suppose you want to use the summation symbol to describe the sum $1+2+3+4+5+6$, then you can write this down compactly as

$$\sum_{k=1}^{6} k = 1+2+3+4+5+6 = 21.$$

The advantage of this notation is that you can write down large sums very compactly. For example, if you want the sum of the first 100 natural numbers, instead of only up to 6, then you can write this down as

$$\sum_{k=1}^{100} k.$$

You can imagine that similar notation can be used also for products of factors. Suppose you want to use the product symbol to describe the product $3 \cdot 4 \cdot 5 \cdot 6$, then you can write this down compactly as

$$\prod_{k=3}^{6} k = 3 \cdot 4 \cdot 5 \cdot 6 = 360.$$

By playing with the value at which the sum (or product) starts or ends you see that you represent many sums or products using this notation. The general notation is the following:

$$\sum_{k=m}^{n} a_k \qquad \text{and} \qquad \prod_{k=\ell}^{r} b_k,$$

where k is the index of summation or of the product; a_k, b_k are indexed variables representing each term of the sum and each factor of the product; m, ℓ are the lower bounds of summation and of the product, and n, r are the upper bounds of summation and of the product.

In all the expressions above, either of summations or products, you can remark that the index k in every step increases by one, it starts from a number m, then takes the value $m + 1$, $m + 2$ until it reaches the number n. It is also possible to choose the indices from some set of values. Say for example that you want to compute the sum of the squares of all even numbers greater or equal to 4 and less or equal to 20. You can define the set of indexes you want to sum over, in this case

$$I = \{N \in \mathbb{N} : 4 \leq N \leq 20 \text{ and } N \text{ is even}\} = \{4, 6, 8, 10, 12, 14, 16, 18, 20\}.$$

Then the desired sum can be written as

$$\sum_{k \in I} k^2 = 16 + 36 + 64 + 100 + 144 + 296 + 324 + 400 = 1380.$$

This sum could also be written compactly as follows

$$\sum_{k \in I} k^2 = \sum_{4 \leq k \leq 20,\ k \text{ even}} k^2.$$

Similar rules hold also for products.

We have paid so much attention on writing out sums and products in order to get used with this notation. Being skilful with the notation and being able to compactly write down mathematics is a very important skill that will turn out to be important in Section 2.3.1.

In what follows we present an important concept in arithmetic, that of a geometric progression and the geometric sum.

Geometric Progression

A geometric progression, also known as a geometric sequence, is a sequence of non-zero numbers where each term after the first is found by multiplying the previous one by a fixed, non-zero number, called the common ratio. For example, the sequence $2, 6, 18, 54, \ldots$ is a geometric progression with common ratio 3. Similarly $10, 5, 2.5, 1.25, \ldots$ is a geometric sequence with common ratio 0.5. If the common ratio is denoted by ω, and the first number in the progression by A_0, then the whole progression can be written as

$$A_0, A_0 \cdot \omega, A_0 \cdot \omega^2, A_0 \cdot \omega^3, \ldots.$$

Suppose we want to sum the first n terms of a geometric progression with parameters A_0, ω. Then we are looking for the following sum

$$\sum_{k=0}^{n} A_0 \cdot \omega^k = A_0 + A_0 \cdot \omega + A_0 \cdot \omega^2 + \ldots + A_0 \cdot \omega^n = A_0 \cdot \sum_{k=0}^{n} \omega^k.$$

We call such a sum a *geometric sum*. After some (non-standard) computations we can find that

$$\sum_{k=0}^{n} \omega^k = \frac{1 - \omega^{n+1}}{1 - \omega}. \tag{1.4.1}$$

Try yourselves to prove this result, it is fun! This compact notation of sums and products allows us to consider sums and products with infinitely many terms and factors! And the amazing thing, very often such infinite sums and products are finite! Lets see an example using the geometric progression above. It is simple to see that for $\omega > 1$ the infinite sum

$$\sum_{k=0}^{n} \omega^k$$

cannot be finite, simply because you keep adding growing numbers to each other. For $\omega = 1$ the same, you just add one infinitely many times. But what happens for $\omega < 1$? In this case this infinite sum is always finite! We can see this from the expression on the right hand side of (1.4.1). The term ω^{n+1} will get smaller and smaller as n grows larger, in the end it will converge to zero. This means that

$$\sum_{k=0}^{\infty} \omega^k = \frac{1}{1 - \omega}, \qquad \text{for } \omega > 1. \tag{1.4.2}$$

This infinite sum appears often in mathematics and will be used in Chapter 2 later when we will study the First Mover Advantage! Being skilful with summations will allow you to grasp everything about the First Mover Advantage in Section 2.3.1.

Acknowledgements

Section 1.2 of this chapter is heavily based on Chapter 1 from the module “Algoritmiek” written by Prof. Mark de Berg from the Eindhoven University of Technology (win.tue.nl/wiskunded/vakken.shtml#beslissen).

2.1. Computational Complexity – How complex is an algorithm?

Often there are different strategies to perform a given task - or "to solve a given problem", as it is usually called - leading to different algorithms. These algorithms typically differ in efficiency: some are fast while others are slow. Of course the actual time a computer needs to solve a problem depends on the speed of the computer. Moreover, it depends on the size of the problem at hand: sorting a thousand numbers will take less time than sorting a million numbers. It is nevertheless possible to compare the efficiency of algorithms in a mathematical way. To this end the number of "steps" of the algorithm is analyzed as a function of the number of input elements.

Algorithmic Complexity

Algorithms typically differ in efficiency: some are fast while others are slow. It is possible to compare the efficiency of algorithms in a mathematical way. To this end the number of "steps" of the algorithm is analyzed as a function of the number of input elements.

2.1.1. How do we Analyse the Running Time of Algorithms?

The most common way to analyse the running time of an algorithm is the so-called *worst-case* analysis. As the name suggests, this is the longest running time for *any* input of size n: this means that, whenever we will analyse the time complexity of an algorithm, we will be very pessimistic and assume that our input is the worse possible. The reason for this strategy is to provide strong **guarantees** that the algorithm will never take any longer, not even in the most unfortunate cases... that, for some algorithms, occur quite often.

Some simplifying abstractions are luckily allowed when analysing algorithms. In fact, we do not need to determine the exact computational cost of each step of an algorithm, but we are rather interested in its **order of growth**. This means, intuitively speaking, that we are e.g. required to determine if the cost of a certain procedure is some constant c, but we do not need to compute the actual value of c.

Likewise, if we know that the cost of a task can be expressed as a polynomial in the size of the input, e.g., cost(input of size n) $= an^2 + bn + c$, we are only interested in its highest-order term (in this case an^2) and we can even ignore this term's constant coefficient: we will just say that the cost of this task is *quadratic*. This is because, for large enough inputs, the multiplicative constants and the terms of lower order are dominated by the effects of the highest-order term; and we are only concerned with large inputs, as this is when things can go sideways if an algorithm is not efficient enough.

Order of growth

if we know that the cost of a task can be expressed as a second order polynomial in the size of the input, e.g., cost(input of size n) $= an^2 + bn + c$, we will say that the cost of this task is **quadratic**. If the cost of the task can be expressed as a first order polynomial in the size of the input, e.g., cost(input of size n) $= dn + e$, we will say that the cost of this task is **linear**.

Let us do a bit of algebra to understand why this is all that matters. Suppose you have an incredibly super-fast computer S, which can execute 10 billion (10^{10}) instructions per second, and a more modest computer M, which can execute 10 million (10^7) instructions per second: thus computer S is 1000 times faster than computer M.

Now suppose you want to solve some task for which you have two possible algorithms: Algorithm 1 requires $2n^2$ (quadratic) instructions for an input of size n, while Algorithm 2 requires $120n$ (linear) instructions. You run the quadratic Algorithm 1 on the super-fast computer S, and the linear-time Algorithm 2 on computer M, both for the same large input of size $n = 10^7$ (10 millions). The super-fast computer S will take

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20000 \text{ seconds};$$

while the modest computer M will take

$$\frac{120 \cdot 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} = 120 \text{ seconds}.$$

So an incredibly fast computer takes more than 5.5 hours to solve a task that a 1000 times slower computer can solve in 2 minutes, just because the slower computer uses a linear-time algorithm in place of a quadratic one!

Notice that the multiplicative constants play a very marginal role here: even if the linear-time algorithm had a multiplicative constant of 1200 instead of 120, the slower computer could solve the task in 20 minutes, still way faster than the 5.5 hours required by the fast computer. Lower-order terms would also not change the results significantly, as they could only add a bunch of seconds or milliseconds to each algorithm (you can try to do the math yourself).

These concepts will be made more precise in the next section, in which we introduce the mathematical notation we use for this analysis. The example in Section 2.1.3 will also be helpful in understanding and clarifying these matters.

2.1.2. The $\mathcal{O}$-Notation

We are interested in studying how the running time of an algorithm increases with the size of the input in the limit, that is, when the size of the input grows unbounded. The asymptotic notation we are going to use is defined in terms of functions from $\mathbb{N}$ to $\mathbb{N}$. Although this notation is primarily used to describe the running times of algorithms, it can be applied as well to describe other aspect of algorithms (for example the amount of space they require) or even to functions that have nothing to do with algorithms.

We say that a function $f(n)$ is "big-oh" of another function $g(n)$, and we write $f(n) = \mathcal{O}(g(n))$, if there exist a positive constant c and a natural number n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. The $\mathcal{O}$-notation is used to give an upper bound on a function, to within a constant factor: $\mathcal{O}(g(n))$ actually denotes the *whole set* of functions that are asymptotically smaller or equal to $g(n)$ up to constant factors. For example, any function $an + b$ with $a > 0$ is in $\mathcal{O}(n)$. In fact, this is true if there exist a constant c and an n_0 such that

$$cn \geq an + b, \ \forall n \geq n_0.$$

If we take, for example, $c = 2a$ and $n_0 = 2|b|/a$, you can verify that the inequality is always satisfied for $n \geq n_0$. Of course there are other possible choices for c and n_0, but this is not important: the only important thing is that *some* values for c and n_0 exist. It is handy to know that, in general, any polynomial $P(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_0$ of degree d with $a_d > 0$ is in $\mathcal{O}(n^d)$, as well as any polynomial of degree $d' < d$. In particular, any constant function is in $\mathcal{O}(1)$.

The $\mathcal{O}$-notation is extremely useful in the analysis of the running time of algorithms because, by bounding the function of its cost from above, it offers guarantees on the behaviour of the algorithm for arbitrarily large inputs, which is precisely our goal. In the next section we will analyse the time complexity of different algorithms that solve a concrete problem.

Bounding the complexity of an algorithm

A bound of the function of the cost of algorithm from above offers guarantees on the behaviour of the algorithm for arbitrarily large inputs.

2.1.3. An Example: Searching on a Sorted Data

In this section we consider a problem similar to the one discussed before in Section 1.2.1. In Section 1.2.1 we presented a Linear Search algorithm (Algorithm 1) which could find an element q in a given set of numbers. In this section we consider a slightly different problem.

We consider the following problem: you have a **sorted** sequence of n items and you want to search for a certain item x in this sequence. Real-life examples may be searching for a book

in a library where the books are sorted in increasing lexicographical order, or searching for a name in an address book, or a registration number in a sorted list.

The first strategy that comes to mind is to simply scan the sequence until either the desired item is found or the search fails, if the item is not part of the sequence. A pseudocode for this simple algorithm is as follows.

Algorithm 2 Exhaustive Search

```
1: INPUT: A sorted sequence s = s[1]s[2]...s[n] of items from a set X and an
   item x ∈ X.
2: OUTPUT: An index i ∈ [1, n] such that s[i] = x; or FAIL if no such index exists.
3: i ← 1;
4: while i ≤ n do
5:     if s[i] = x then
6:         return : i;
7:     else
8:         i ← i + 1;
9: if i = n + 1 then return : FAIL;
```

What is the running time of this algorithm? The instructions of Line 3 (initialise index i to 1), Line 4 (check whether the index is still at most n), Line 5 (compare x with the item at position i), Lines 6 and 9 (return the index or FAIL if the item was not found) and Line 8 (increase index i by 1) require a constant time c_1, c_2, c_3, c_4, c_5, respectively.

How many times do we execute the loop of Lines 4-8? If we find x at position j, we execute it j times, each requiring constant time. Thus the cost of the algorithm is given by $c_1 + j(c_2 + c_3 + c_5) + c_4$. The worst-case scenario is when x is either the last element of s or it is not in the sequence, because in this case we execute the loop n times. Since we are performing a worst-case analysis, and since we ignore the values of the constants, the running time of this algorithm is in $\mathcal{O}(n)$.

Can we design a more efficient algorithm? Notice that we have not used at all the information that our sequence s is sorted: the same algorithm with the same running time would also work on an unsorted sequence. Remember that Algorithm 1 in Section 1.2.1 had exactly the same structure as Algorithm 2.

The algorithm we describe next uses the so-called **divide and conquer** technique to exploit the fact that the items of the sequence are sorted and achieve a much better running time.

What does Algorithm 3 do? The first time it executes the while loop, mid is the middle position of the sequence (Line 5). The item at this position is compared with x: if we are lucky and they are equal, it means we have found x and we return its position mid.

Algorithm 3 Binary Search

```
1: INPUT: A sorted sequence s = s[1]s[2]...s[n] of items from a set X and an
   item x ∈ X.
2: OUTPUT: An index i ∈ [1, n] such that s[i] = x; or FAIL if no such index exists.
3: start ← 1, end ← n;
4: while start ≤ end do
5:     mid ← ⌊(start + end)/2⌋;
6:     if s[mid] = x then
7:         return : mid;
8:     else if s[mid] < x then
9:         start ← mid + 1;
10:    else if s[mid] > x then
11:        end ← mid − 1;
12: return : FAIL;
```

Otherwise, if x is greater than the element in the middle, it means that it should be in the second half of the sequence, because the sequence is ordered! This is the crux that makes this algorithm faster!

This is why, at Line 9, we set start to be the position immediately to the right of mid: because that is the smallest position where we could find x. The symmetric situation is when x is smaller than the middle element: in this case it could only be found in the first half of the sequence and therefore we set end to be the position to its left (Line 11).

In both cases, the next time the loop is executed it will repeat the same procedure in the half of the sequence where x could be found. The instructions in the loop are repeated until either x has been found or end becomes smaller than start, which only happens if x is not in the sequence (you can easily verify that): in this case, we return FAIL (Line 12).

The reason why this strategy is intuitively more efficient than Algorithm 2 is that we never check all the elements in the sequence: in fact, at each iteration we completely disregard half of the remaining elements, and we will never need to check them!

Let us now analyze the asymptotic running time of Binary Search in the worst case. The worst case is when x is not in the sequence, and thus we have to execute the while loop until start $=$ end. How many times do we execute it in this case?

To find it out, let us focus on the number of elements between start and end at each iteration. At the first iteration they are all the elements of s, thus they are n. At the second iteration they are $n/2$, because we only focus on one half of the sequence; at the third iteration they are half of half, thus $n/4 = n/2^2$... and so on. In general, at iteration i we have $n/2^{i-1}$ elements. At the last iteration we know that we have just one element, because start $=$ end.

Thus we have

$$1 = n/2^{\text{last}-1} \implies 2^{\text{last}-1} = n \implies \text{last} = \log_2(n) + 1.$$

We found out that the loop is executed $\log_2(n) + 1$ times in the worst case! Since each instruction in the loop requires a constant time, and since we ignore these constants, we conclude that the running time of binary search is in $\mathcal{O}(\log_2(n))$. By doing Exercise 11 you will realise yourself what a great difference there is between an $\mathcal{O}(\log_2(n))$-time algorithm and an $\mathcal{O}(n)$-time algorithm!

2.2. P vs NP

Both Algorithms 2 and 3 of the previous section have a *polynomial* running time. In general, a polynomial-time algorithm has a worst-case running time of $\mathcal{O}(n^k)$ for some constant $k > 0$.

Polynomial time algorithms

We say that an algorithm runs in polynomial time if the function of the cost of algorithm given the input n has a worst-case running time of $\mathcal{O}(n^k)$ for some constant $k > 0$.

An important question that arises is the following. Does there always exist a polynomial-time algorithm for solving any problem? The answer is no, and we will see an example of problem for which no polynomial-time algorithm is known shortly.

We call the class of problems for which we know at least one algorithm with polynomial worst-case running time **P**. The problems in **P** are commonly regarded as easy problems, while we consider hard the problems for which no polynomial-time algorithm has been discovered yet.
There is an interesting class of problems, called the "**NP**-complete" problems, for which no polynomial-time algorithm has been discovered yet, but for which no proof that such an algorithm cannot exist has been found either. We say that a problem is in **NP** if it is *verifiable* in polynomial time. This means that, if someone gives us an alleged solution to the problem (also called a *certificate*), we are able to verify whether this is an actual solution or not in polynomial time.
Clearly this is true for all the problems in **P** (for which we can even *produce* a solution in polynomial time), so we know that **P** $\subseteq$ **NP**. Whether or not **P** $\neq$ **NP**, though, is an open question since 1971, when it was first posed: there is even a million-dollar prize for whoever can prove one of the two options! Deep down, most computer scientists think that **P** $\neq$ **NP** is the most reasonable hypothesis (...maybe we think that we are so clever that,

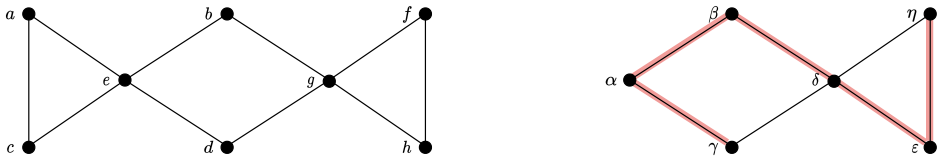


Figure 2.2.1. A graph with no Hamiltonian paths (left) and a graph with an Hamiltonian path (right).

if a polynomial-time algorithm for those problems existed, we would have found it by now), but it is surprising how several problems in **NP** for which no polynomial-time algorithm is known are apparently very similar to problems in **P**.

An example is the following: given a graph $G(V, E)$ (see Section 1.3), we want to find out whether

(a) there is a way of visiting all the edges of G exactly once with an unbroken path (an Eulerian path thus), and
(b) there is a way of visiting all the vertices of G exactly once with an unbroken path (a Hamiltonian path thus).

The two problems seem very similar, right? However, while there is an algorithm that solves problem (a) for a graph with m edges in $\mathcal{O}(m)$ time (and thus problem (a) is known to be in **P**), no polynomial-time algorithm is known for problem (b). We will discuss the Hamiltonian path problem in detail in the next section.

2.2.1. An Example: Hamiltonian Path

Let us start by refreshing the Hamiltonian Path problem from Section 1.3.2 and discuss it a bit more formally.

In input we are given a graph $G(V, E)$ with n vertices and m edges; the output is either YES or NO, depending on whether or not there exists a continuous path in G that touches each of its vertices exactly once. A *certificate* for Hamiltonian Path is a sequence $v_1 v_2 \dots v_n$ of n vertices of G, and it is a valid solution if and only if

(i) $v_i \neq v_j$ for any two distinct $i, j \in [1, n]$ and 2
(ii) there is an edge between any two consecutive vertices in the sequence, that is, $\{v_i, v_{i+1}\} \in E$ for all $i \in [1, n-1]$.

Inspect Fig. 2.2.1. The vertex set of the graph on the right is $V = \{\alpha, \beta, \gamma, \delta, \varepsilon, \eta\}$; its edge set is $E = \{\{\alpha, \beta\}, \{\alpha, \gamma\}, \{\beta, \delta\}, \{\gamma, \delta\}, \{\eta, \delta\}, \{\eta, \varepsilon\}, \{\varepsilon, \delta\}\}$. A Hamiltonian path in the graph is highlighted in red and it is given by the sequence of vertices $\eta\varepsilon\delta\beta\alpha\gamma$: indeed, each vertex appears exactly once in the sequence, and there is an edge for each pair of consecutive vertices. The graph on the left has no Hamiltonian path: in fact, you can easily see that it

is not possible to visit all vertices without traversing e or g at least twice.

The Hamiltonian Path problem is in **NP**. To prove it, it suffices to give a polynomial-time algorithm that, given a sequence $v_1 v_2 \dots v_n$ of n vertices of a graph, checks whether it corresponds to an Hamiltonian path or not. A possible algorithm is the following, that we call VerifyHP.

First read the sequence from left to right. When we read vertex v_i, we mark it as *visited* in the graph. If we read a vertex that is already marked as visited, we know that it is not a Hamiltonian path, thus we can stop. Otherwise, if we read the whole sequence and no vertex was visited twice, we proceed to check if each pair of consecutive vertices corresponds to an edge of the graph.

To do so, we start again from the beginning of the sequence, we read the first two vertices v_1, v_2 and we check if $\{v_1, v_2\} \in E$. If so, we discard v_1, read the next vertex v_3 and check if $\{v_2, v_3\} \in E$. We proceed like this until either we find a pair that is not an edge, and thus the sequence does not give a Hamiltonian Path, or we arrive at the end of the sequence, and thus the sequence corresponds to a Hamiltonian path.

Exercise 12 asks you to decide what is the worst-case running time of VerifyHP, to confirm that it always runs in polynomial time and thus that the Hamiltonian Path problem is in **NP**.

Algorithm VerifyHP can be used to find a solution to the Hamiltonian Path problem via the following algorithm, that we call FindHP. Consider all possible permutations of the vertices of the graph (that is, all possible sequences that contain each vertex exactly once). Then run VerifyHP with each such permutation as input. If the graph has an Hamiltonian Path, this procedure guarantees to find it, because it tries all possibilities. Likewise, if no Hamiltonian Path is found after verifying all permutations, we are guaranteed that the graph does not have any.

What is the time complexity of FindHP? Let VHP be the time complexity of VerifyHP, which must be run on every possible permutation. Since there are $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = \mathcal{O}(n^n)$ distinct permutations, the worst-case running time is $\mathcal{O}(n^n \cdot \text{VHP})$, which is not polynomial because the input size (n) (which is not a constant!) appears as exponent.

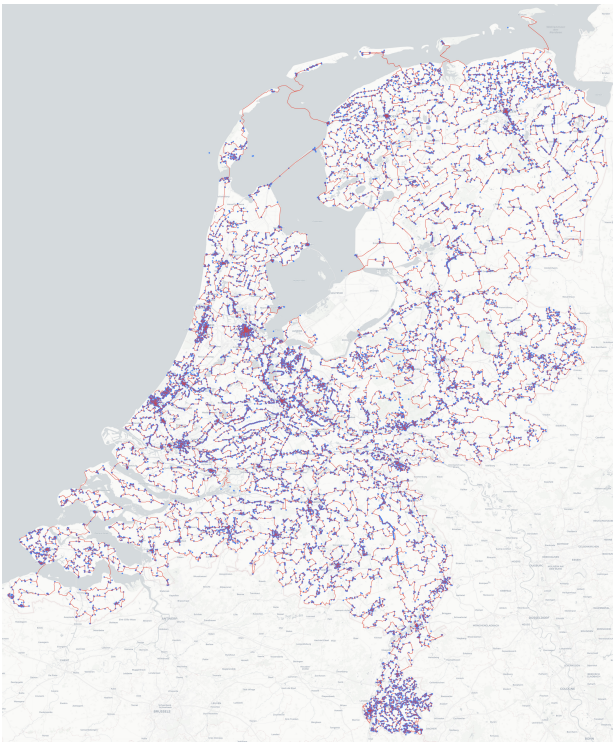


Figure 2.2.2. An optimal biking route among all nodes in the graph made of 57,912 national monuments in the Netherlands. Taken from the Travelling Salesman Problem website (http://www.math.uwaterloo.ca/tsp/nl/index.html)

On the Network Pages

For further reading on algorithms, their complexity, their history, and their applications you can read the following articles.

(1) *Enigma: a complexity titan* by Stijn Maatje,

networkpages.nl/enigma-a-complexity-titan /.

(2) *Cycling to 57,912 National Monuments in the Netherlands* by Nicos Starreveld,

networkpages.nl/cycling-to-57912-national-monuments-in-the-netherlands/.

(3) *Attacking complex problems using preprocessing* by Astrid Pieterse,

networkpages.nl/attacking-complex-problems-using-preprocessing/.

(4) *A multi-objective fight against prostate cancer* by Stef Maree,

networkpages.nl/a-multi-objective-fight-against-prostate-cancer/.

(5) *Google PageRank: how search engines 'bring order to the Web'* by Nelly Litvak

networkpages.nl/google-pagerank-how-search-engines-bring-order-to-the-web/.

2.3. Mathematics to make sports more honest

As the final minutes of the football European Championships final passed by, tensions grew, not only with the spectators on the couch, but also in the stadium and on the pitch. None of the teams wanted to be that team that made a fatal mistake with mere seconds left to play, and see the other lift the trophy. Thus, the game faded, with everyone waiting anxiously until the moment the referee blew the final whistle. At which point, the nerve-wrecking excitement only just started.

The penalties. In a best-of-5 series, with sudden death if that resulted in a tie, both teams had to take penalties to decide a winner. The Italians knew what they had to do, as they had won the semi-finals after penalties.

It is an odd thing that some of the most influential moments at major football tournaments have little to do with the regular game. Penalty series are a discipline on its own, or a lottery, as some less-gifted penalty takers have called it at times.

They are no lottery, you could and should practice them to increase your chances, but they are meant to give both teams an equal and fair chance of progressing to the next round or even winning the tournament. And why wouldn't they be fair? Both teams get the same task - convert as many penalties as possible - and thus have the same chance of winning. Right?

There is a catch. Both teams indeed have the same objective, but they don't have the same path. In a usual penalty series, there is one team that starts the series and the other one has to catch-up all the time. Surprisingly, statistical research has suggested that this actually gives an advantage to the team starting first! This knowledge and feeling is in fact so widely accepted that most trainers will have their team shoot first if they win the toss to determine the starting team.

In this chapter we will look at a way to model this advantage, called First Mover Advantage (FMA), and try to find different shooting orders for penalties that are fair.

2.3.1. Modelling First Mover Advantage

We want to model the above phenomenon, FMA. First we assume that both teams are equally skilled. That way, any advantage or disadvantage that we may find, directly indicates an unfairness.

Next, we have to look at a penalty series. A series consists of rounds, in every round both teams take one penalty. It starts with 5 rounds in a Best-of-5 series (later, we might replace the 5 with a general number n) and, in the case of a tie, a sudden death phase starts - as soon as one of the teams misses in a round and the other team scores, the series end.

A way to model First Mover Advantage is given below:

Model 1

We call this first model the Simple(p, q) model.

- A team shooting first in a round scores with probability p.
- A team shooting second in a round scores with probability q.

Any choice of $p > q$ could (and will) lead to unfairness. An alternative modelling approach is the following:

Model 2

We call this first model the Pressure(p, q) model.

- When a team takes a penalty, while being at least equal in the score, they score with probability p.
- When a team takes a penalty, while being behind, they score with probability q.

The idea behind Model 2, is that a player that shoots while they are behind, feels the pressure to catch up, and is thus more likely to miss. This is the model that we will mostly focus on.

Fairness

We say a shoot-out is **fair** with respect to a model with parameters p, q, if:

$$\mathbb{P}(\text{Team A wins}) = \mathbb{P}(\text{Team B wins}) = \frac{1}{2}.$$

The notation $\mathbb{P}(A)$ denotes the **probability** of some event A.

This doesn't look too exciting, but it is crucial to almost everything of our analysis.

2.3.2. Sudden death

Common penalty shoot-out have a sudden death phase, where every miss can be definitive. We will only analyse this part, as it is the most interesting from a mathematical point of view - there is no arbitrary cut-off moment as we have with the first phase, which is a best-of-5.

A sudden death consists of possibly infinitely many rounds, and all rounds start with an equal standing between both the teams - if the standings are not equal, the sudden death would have finished already.

Thus, we can define the probability of Team A winning a sudden death as:

$$\mathbb{P}(\text{Team A wins}) = \sum_{r=1}^{\infty} \mathbb{P}(\text{Team A wins in round } r). \tag{2.3.1}$$

And similarly for Team B. It is important to notice that a team can only win in a specific round, if the sudden death reaches that round. Thus, we can write:

$$\mathbb{P}(\text{Team A wins in round } r) = \mathbb{P}(\text{The first } r-1 \text{ rounds are tied})\mathbb{P}(\text{Team A wins in round } r)$$

Any round is treated equally by both models 1 (Simple(p,q)) and 2 (Pressure(p,q)) , thus this can be rewritten again as:

$$\mathbb{P}(\text{Team A wins in round r}) = \mathbb{P}(\text{A round is tied})^{r-1}\mathbb{P}(\text{Team A wins in round r})$$

We can calculate the probability that a round is drawn per model. This happens if either both teams score, or both teams miss:

$$\begin{aligned} \text{Simple}(p,q): &\quad \mathbb{P}(\text{Round is drawn}) = p\cdot q + (1-p)(1-q). \\ \text{Pressure}(p,q): &\quad \mathbb{P}(\text{Round is drawn}) = p\cdot q + (1-p)(1-p). \end{aligned}$$

Before proceeding try to understand why these formulas hold. When is a round a draw in each of these models?

We can also calculate, for both models, the probability that the first shooting, denoted by P_+ from now on, (resp. second shooting, denoted by P_- from now on) team wins the round. The probability of a tie will be denoted from now on by $P_\pm$. These probabilities are shown in Table 2.3.1.

Model	First wins	Second wins	Tie
Simple(p,q)	$p(1-q)$	$(1-p)q$	$pq+(1-p)(1-q)$
Pressure(p,q)	$p(1-q)$	$(1-p)p$	$pq+(1-p)^2$

Table 2.3.1. Probabilities per model

EXERCISE 1. Given parameters p,q and model Simple(p,q), find - if possible - values p',q' such that Pressure(p',q') is equivalent - that is, both models have the same probabilities for all three outcomes.

From here on we only consider the Pressure model. The reason why we do this is made clear in Exercise 1, where it is asked to show that these models are equivalent.

Now that we determined the probability of the first shooting team winning, i.e. P_+, second shooting team winning, i.e. P_-, and the probability of a tie, i.e. $P_\pm$, we can further specify

the probability of a team winning the sudden death given in (2.3.1). To do this, we need to know in which rounds Team A actually shoots first. We denote these rounds by index set $I \subset \mathbb{N}$. Automatically, team B shoots first in all other rounds, I^c.

A normal penalty sequence has a single team always shooting first - the AAA-sequence, referring to team A shooting first. The index set I belonging to this sequence is given by $I_{AAA} = \mathbb{N}$, simply all the rounds. Another popular alternative that is used in some sports and is trialled in football, is the $ABBA$-sequence, which should be read as $A(B)B(A)$. Team A and B alternate shooting first. When using this sequence, I equals all the odd numbers and I^c all the even numbers.

Without choosing any set I yet, we get to the following expression for the winning probability of team A:

$$\begin{aligned}\mathbb{P}(\text{Team A wins}) &= \sum_{r=1}^{\infty} P_{\pm}^{r-1}\mathbb{P}(\text{Team A wins in round } r)\\ &= \sum_{r\in I} P_{\pm}^{r-1}P_{+} + \sum_{r\in I^c} P_{\pm}^{r-1}P_{-}.\end{aligned}$$

Using Table 2.3.1, we see that $P_+ - P_- = p(p-q) =: p\lambda$, where $\lambda = p - q$ - as we assumed $p > q$, $\lambda > 0$. This means we can replace $P_+ = P_- + p\lambda$, and using that $\mathbb{N} = I \cup I^c$, leads to:

$$\mathbb{P}(\text{Team A wins}) = P_- \sum_{r\in\mathbb{N}} P_{\pm}^{r-1} + p\lambda \sum_{r\in I} P_{\pm}^{r-1}.$$

EXERCISE 2. When we define $f(I, P_{\pm}) = \sum_{r\in I} P_{\pm}^{r-1}$, show that:

$$\begin{aligned}\mathbb{P}(\text{Team A wins}) &= \frac{P_-}{1-P_{\pm}} + p\lambda f(I, P_{\pm}).\\ \mathbb{P}(\text{Team B wins}) &= \frac{P_-}{1-P_{\pm}} + p\lambda f(I^c, P_{\pm}).\end{aligned}$$

Hint: Use the expression for the sum of the terms in a geometric regression in (1.4.2).

Thus, for any shoot-out with $f_A := f(I, P_{\pm})$ and $f_B = f(I^c, P_{\pm})$, it is fair if $f_A = f_B$.

Criterium for fairness

Models 1 and 2 and the compact notation used above to write down the probabilities that each team wins allow us now to state a criterion for fairness, namely, any shoot-out with $f_A := f(I, P_{\pm})$ and $f_B = f(I^c, P_{\pm})$, it is fair if $f_A = f_B$.

It is time to put the result from Exercise 2 to use.

EXAMPLE 2.3.1 (Unfair sequences). We check two mentioned shootout-series, the common AAAA and the ABBA order. We know that $I_{AAA} = \mathbb{N}$ is the index set of the common penalty series, that is team A always shoots first. Then:

$$f_A := f(I_{AAA}, P_\pm) = \frac{1}{1-P_\pm} \qquad f_B := f(I^c_{AAA}, P_\pm) = 0$$

Clearly, we can conclude that the regular series is unfair, as $f_A > 0$.

For the ABBA-series, we know that $I_{AB} = \{1, 3, 5, \dots\}$ is the index set belonging to Team A in this series. Hence the index set belonging to Team B in this series is $I^c_{AB} = \{2, 4, 6, \dots\}$. Then:

$$f_A = \sum_{r\in I_{AB}} P_\pm^{r-1} = \frac{1}{P_\pm} \sum_{r\in I_{AB}} P_\pm^{r} = \frac{1}{P_\pm} \sum_{r'\in I^c_{AB}} P_\pm^{r'-1} = \frac{f_B}{P_\pm}.$$

As $P_\pm < 1$, we see that $f_A > f_B$, thus we can conclude that even the ABBA-series is unfair.

We found that both these sequences are unfair. Why, and should we be surprised? No, we shouldn't be surprised at all - as will become clear in the next paragraph. Both the AAAA- as the ABBA-sequence, are examples of repetitive sequences, meaning that after a fixed amount of rounds, resp. 1 and 2, the sequence repeats it self. Every repetitive shootout $\mathcal{S}$ is determined by the part S that gets repeated (obviously). If we denote k as the length of the repeated part S and I_S as Team A's index set in these k rounds, we see that:

$$f_A = \sum_{r\in I} P_\pm^r = \left(\sum_{r\in I_S} P_\pm^{r-1}\right) + \left(\sum_{r\in I_S} P_\pm^{r+k-1}\right) + \ldots = \frac{\sum_{r\in I_S} P_\pm^{r-1}}{1-P_\pm^k}.$$

Do you see why the last equality holds? Try to spot the geometric regression hidden in the sum in the middle!

With $I^c_S = \mathbb{N}_{\leq k} \setminus I_S$ the index set of rounds where Team B shoots first leading to:

$$f_A = f_B \iff \sum_{r\in I_S} P_\pm^{r-1} = \sum_{r\in I^c_S} P_\pm^{r-1}.$$

This formula leads to many conclusions, but most importantly, if we take p, q to be any rational number, i.e. a fraction, it can be shown that no finite repeated sequence will ever be fair!

2.3.3. Looking for fair sequences

In the previous section, we analysed the set-up of a sudden death shootout, and showing unfairness of the most common sequences. However, we would like to find nice sequences that are fair. The idea of the ABBA-sequence, is that having Team B should first in all the

even rounds, cancels the advantage of Team A shooting first in odds rounds. This does not work, unfortunately, as the fact that Team A consistently has its advantage first, is in itself an advantage. But, we might expand on this idea.

What if we flip the order of penalties after the second round again for the two rounds after that? And, for the following four rounds, flip them again? And so on and so on. Wouldn't that cancel out any of the sequences?

The Prohuet-Thue-Morse sequence

The Prohuet-Thue-Morse sequence is created like this:

- The first two terms are given by 01.
- The second two terms are given the inverse, 10, thus leading to 0110.
- The next four terms are given by its inverse, leading to 01101001.
- etc.

We can apply this sequence, hypothetically, as an order for the penalties in a shootout.

Theorem 2.3.1. *The PTM-sequence S_{PTM} as a penalty shoot-out in the sudden death, is still unfair, for any value of $p, q, P_{\pm}$.*

Proof. To prove this theorem, we need to find a way to express f_A, f_B of the given sequence S_{PTM}. We will not specifically look at f_A, f_B, but at their difference, $f_A - f_B$:

$$f_A - f_B = \sum_{r \in I_{PTM}} P_{\pm}^{r-1} - \sum_{r \in I_{PTM}^c} P_{\pm}^{r-1} = 1 - P_{\pm} - P_{\pm}^2 + P_{\pm}^3 + \dots$$

EXERCISE 3. Argue that, as the order of the shootout is inverted after 2^k rounds, we can write:

$$\Delta := f_A - f_B = (1 - P_{\pm})\left(1 - P_{\pm}^2\right)\left(1 - P_{\pm}^4\right)\cdots = \prod_{k=1}^{\infty}\left(1 - P_{\pm}^{2^k}\right)$$

If we want to prove unfairness in all situations, we need to prove $\Delta > 0$ whenever $P_{\pm} > 0$. But is it? From exercise 3, we have that we can write Δ as a product of terms. And, one might think that, as all the individual terms $(1 - P_{\pm}^{2^k})$ are non-zero (> 0), the product itself is non-zero. Except, we have an infinite number of terms, which makes things complicated. To see why, see the following product:

$$Z = \left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{4}\right)\cdots = \prod_{k=2}^{\infty} 1 - \frac{1}{k}$$

This only has non-zero terms multiplied, so one could think that $Z > 0$. In fact, $Z = 0$, as:

$$Z = \prod_{k=2}^{\infty} 1 - \frac{1}{k} = \prod_{k=2}^{\infty} \frac{k-1}{k} = \lim_{k\to\infty} \frac{1}{2}\cdot\frac{2}{3}\cdot\frac{3}{4}\cdots\frac{k-1}{k} = \lim_{k\to\infty} \frac{1}{k} = 0$$

Thus, the question remains whether (and for which $P_\pm$) the corresponding Δ is bigger than 0. There are several ways to approach this - one is to look for the right theorems in the literature. We will do it more intuitively.
Lets write $\Delta(x) = \prod_{k=0}^{\infty}(1 - x^{2^k})$, the original expression for Δ with the term $P_\pm$ replaced by variable x. Notice that the following holds:

$$\Delta(x^2) = (1 - x)\Delta(x)$$

As we want to prove that $\Delta(x) > 0$ for $0 < x < 1$, we immediately see that $\Delta(x) = 0$ would imply that $\Delta(x^2) = 0$ (*). And this holds for general x. It is not difficult to see that $\Delta(x) \geq \Delta(y)$ when $x \leq y$ (**). Combining these two observations (*) and (**), we find that:

$$\text{If there is } 0 < x < 1 \text{ such that } \Delta(x) = 0 \text{then } \Delta(x) = 0 \text{ for all } 0 < x < 1$$

From this statement, the only thing that is left to prove that $\Delta(x) > 0$ for all $0 < x < 1$, is proving that there exists an x for which $\Delta(x) > 0$. This is asked in one of the exercises in 3.5.

□

Finding a fair sequence

So far, we have not found any fair sequences. However, it is very well possible to find a fair sequence for any given value $p, q \in \mathbb{Q}$.

The following algorithm does so:

Algorithm to find a fair sequence

- Fix p, q as values that resemble reality. For instance, $p = \frac{3}{4}, q = \frac{2}{3}$. Let I, I^c be the rounds where A, B shoot first respectively.
- Assign Team A to shoot first in the first round.
- In any round r, calculate
$$f_A^r = \sum_{i \in I, i<r} P_\pm^{i-1}$$
and
$$f_B^r = \sum_{i \in I^c, i<r} P_\pm^{i-1}.$$
Let A shoot first in round r if $f_A^r \geq f_B^r$, and B otherwise.
- Do until infinity.

This algorithm produces a fair sequence for any p, q as long as $P_\pm \geq \frac{1}{2}$. Why? When $P_\pm < \frac{1}{2}$, after the advantage gained by shooting first in the first round, it is very unlikely that more rounds will be played (as $P_\pm < \frac{1}{2}$) so it is impossible to make up for the disadvantage.

When $P_{\pm} \geq \frac{1}{2}$, the advantage to either of the sides will be less then some constant times $P_{\pm}^{r-1}$ in round r, thus in the long run, the sequence will be fair.

EXERCISE 4. For $p = \frac{3}{4}, q = \frac{2}{3}$, determine the order of the first 10 penalties according to the algorithm above.

Figure 2.3.1. For an honest, nice, and successful tournament it is important to make a schedule which is as balanced as possible. What would you think if we told you that a balanced schedule for the Premier League of Darts exists... and is not used...

On the Network Pages

For further reading on applications of mathematics in sports you can read the following articles.

(1) *A balanced schedule for the Premier League of Darts exists... and is not used...* by Roel Lambers, Rudi Pendavingh and Frits Spieksma,

networkpages.nl/a-balanced-schedule-for-the-premier-league-of-darts-exists-and-is-not-used/ .

(2) *How the schedule in the TATA Steel Chess Championship forced Carlsen to help Caruana win* by Roel Lambers, Jasper Nederlof and Frits Spieksma ,

networkpages.nl/how-the-schedule-in-the-tata-steel-chess-championship-forced-carlsen-to-help-caruana-win/.

3.1. Exercises on algorithms

EXERCISE 5. What does Algorithm 4 below give as output?

Algorithm 4 An Algorithm A

```
1: INPUT: An array of numbers A[1,...,n].
2: OUTPUT: ????
3: i ← 1, amount ← 0;
4: while i ≤ n do
5:     if A[i] > 0 then
6:         amount←amount+1
7:     i←i+1;
8: return amount;
```

Remark

Note that the command "$i \leftarrow i + 1$" is indented a bit, and straight under **if** and **then**. This indicates that this command belongs at the while-command. The command "return amount" is not indented, and is therefore only executed after the while-command has been completed.

EXERCISE 6. (Be careful! Trick question)
What does Algorithm 5 below give as output?

Algorithm 5 A crazy algorithm ∞

```
1: INPUT: An array of numbers A[1,...,n].
2: OUTPUT: ????
3: i ← 1, total ← 0;
4: while i ≤ n do
5:     total←total+A[i]
6: return total;
```

EXERCISE 7. What does Algorithm 6 below give as output?

EXERCISE 8. Write pseudocode for an algorithm which, for a given array $A[1, \ldots, n]$ of n numbers, computes the largest number in the array.

EXERCISE 9. Write pseudocode for an algorithm which, for a given array $A[1, \ldots, n]$ of n numbers and a number q, gives **True** if there are indices $i < j$ such that $A[i] + A[j] = q$.

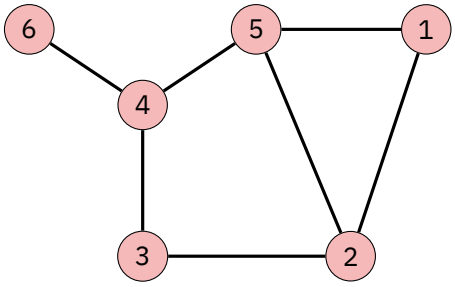
Algorithm 6 Algorithm B

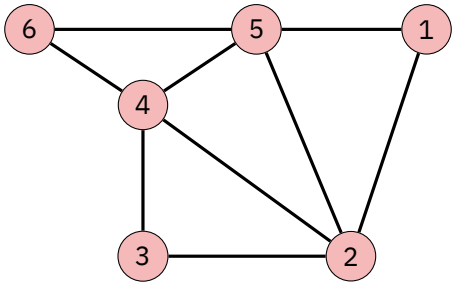
```
1: INPUT: An array of numbers A[1, ..., n].
2: OUTPUT: ????
3: i ← 1, total ← 0;
4: while i ≤ n do
5:       total←total+A[i]
6:      i←i+1
7: return total;
```

3.2. Exercises on graph theory

3.2.1. Finding Eulerian and Hamiltonian paths

EXERCISE 10. Find Eulerian and Hamiltonian paths in each of the following graphs (if possible). You need to try different starting points in order to find a path.





3.3. Exercises on algorithmic complexity

EXERCISE 11. Suppose you have a computer that can execute 10 millions (10^7) instructions per second. You are provided two algorithm for a certain task: Algorithm A solves it with $60 \log_2(n)$ instructions for an input of size n, Algorithm B with $3n$ instructions. How long does each of the algorithms take to solve the task for an input of size 10^8? And for an input of size 10^{12}?

EXERCISE 12. Consider Algorithm VerifyHP described in Section 2.2.1. Can you tell what is its worst-case running time?

EXERCISE 13. Consider the graph on the left of Figure 2.2.1. Take any permutation of its 8 vertices (e.g. $acebghfd$) and verify using Algorithm VerifyHP, described in Section 2.2.1, that it does not correspond to an Hamiltonian path.

EXERCISE 14. Consider the graph on the right of Figure 2.2.1. Can you find an Hamiltonian path different from the one highlighted in the figure?

3.4. Exercises on sums and products

EXERCISE 15. Use the summation and product notation to write down in a compact manner the following sums and products.

$$\begin{aligned} S_1 =& 4+5+6+7+8+9+10 \\ S_2 =& 2+4+6+8+10 \\ S_3 =& 3\cdot 5\cdot 7\cdot 9\cdot 11\cdot 13 \\ S_4 =& 1+0.5+0.25+0.125+0.0625+\dots \\ S_5 =& 1+2^2+3^2+4^2+5^2+6^2. \end{aligned}$$

EXERCISE 16. Use the expressions in 1.4.1 and 1.4.2 to compute the following sums

$$\begin{aligned} \Sigma_1 =& \sum_{k=0}^{10} 3^k \\ \Sigma_2 =& \sum_{k=3}^{6} 0.3^k \\ \Sigma_3 =& \sum_{k=0}^{\infty} 0.7^k. \end{aligned}$$

3.5. Exercises First Mover Advantage

EXERCISE 17. Given parameters p, q and model Simple(p, q), find - if possible - values p', q' such that Pressure(p', q') is equivalent - that is, both models have the same probabilities for all three outcomes.

EXERCISE 18. When we define $f(I, P_\pm) = \sum_{r\in I} P_\pm^{r-1}$, show that:

$$\begin{aligned} \mathbb{P}(\text{Team A wins}) &= \frac{P_-}{1-P_\pm} + p\lambda f(I, P_\pm). \\ \mathbb{P}(\text{Team B wins}) &= \frac{P_-}{1-P_\pm} + p\lambda f(I^c, P_\pm). \end{aligned}$$

Hint: Use the expression for the sum of the terms in a geometric regression in (1.4.2).

Thus, for any shoot-out with $f_A := f(I, P_\pm)$ and $f_B = f(I^c, P_\pm)$, it is fair if $f_A = f_B$.

EXERCISE 19. Argue that, as the order of the shootout is inverted after 2^k rounds, we can write:

$$\Delta := f_A - f_B = (1 - P_\pm)\left(1 - P_\pm^2\right)\left(1 - P_\pm^4\right)\cdots = \prod_{k=1}^{\infty}\left(1 - P_\pm^{2^k}\right)$$

EXERCISE 20. Show that $\Delta(\frac{1}{2}) > 0$.

EXERCISE 21. For $p = \frac{3}{4}, q = \frac{2}{3}$, determine the order of the first 10 penalties according to the algorithm above.

4.1. Algorithms

1. Algorithm 4 gives as output the total number of positive numbers in the array $A[1, \ldots, n]$.
2. Algorithm 5 doesn't give any output, it never terminates since the while-loop never ends. In the while-loop the variable i always stays equal to 1.
3. Algorithm 6 gives as output the sum of the numbers in the array $A[1, \ldots, n]$.

4.2. Algorithmic complexity

1. For an input of size 10^8 Algorithm A uses in total $60 \log_2(10^8) = 480 \log_2(10) \approx 480 \cdot 3.321928 \approx 1595$ instructions. Algorithm B uses in total $3 \cdot 10^8 = 300.000.000$ instructions. A computer that can execute 10 millions instructions per second needs
   (a) $\frac{1595}{10^7} = 0.0001595$ seconds (0.1595 milliseconds) to solve the task using Algorithm A;
   (b) $\frac{3 \cdot 10^8}{10^7} = 30$ seconds to solve the task using Algorithm B.
2. The worst-case running time of VerifyHP occurs when the algorithm needs to check all n vertices in the sequence without terminating. This means that the given sequence has length n and no vertex is repeated. Afterwards the algorithm will control the edges, in the worst-case running time all edges in the given sequence (in total $n - 1$) will be controlled. Thus, the worst-case running time of VerifyHP is $n \cdot (n - 1)$, which is $O(n^2)$.
3. In all permutations of the 8 vertices no vertices will be repeated, hence the algorithm will proceed to check the edges. No matter which permutation you choose the algorithm will always find two vertices in the sequence for which there is no edge in the graph. Take for example the permutation $acebghfd$, you can see that edge fd does not exist in the graph in Figure 2.2.1.
4. Try the path $\gamma\alpha\beta\delta\eta\epsilon$.

4.3. First Mover Advantage

1. In order to find these parameters p', q' we need to solve a system of two equations with two unknowns. The two equations we need to solve are

$$\mathbb{P}(\text{1st shooter wins in Simple}(p, q)) = \mathbb{P}(\text{1st shooter wins in Pressure}(p', q'))$$

and

$$\mathbb{P}(\text{2nd shooter wins in Simple}(p, q)) = \mathbb{P}(\text{2nd shooter wins in Pressure}(p', q')).$$

In these two equations the variables p', q' are the unknowns. Using Table 2.3.1 we obtain the following equations

$$p(1-q) = p'(1-q') \text{ and } (1-p)q = (1-p')p'. \tag{4.3.1}$$

The second equation is a second order equation in p', hence we can use the abc-formula. This equation can be rewritten as

$$(p')^2 - p' + (1-p)q = 0,$$

which has the two solutions

$$p'_{12} = \frac{1 \pm \sqrt{1-4(1-p)q}}{2}.$$

We know that p' represents a probability, hence one of the two solutions can be disregarded. Hence we have that

$$p' = \frac{1 - \sqrt{1-4(1-p)q}}{2}.$$

Since $p > q$ we also have that $(1-p)q < (1-p)p$ which is less than 0.25 for all values of p between 0 and 1. Hence p' is well defined since the quantity in the square root is non-negative. Using the first equation in (4.3.1) we find

$$q' = 1 - \frac{p(1-q)}{p'} = 1 - \frac{2p(1-q)}{1-\sqrt{1-4(1-p)q}} = \frac{\sqrt{1-4(1-p)q} - 2p(1-q)}{1-\sqrt{1-4(1-p)q}}.$$

2. We show that $\Delta(\frac{1}{2}) > 0$. We know that

$$\Delta(x) = \prod_{k=0}^{\infty}(1-x^{2^k}),$$

and consequently

$$\Delta(\frac{1}{2}) = \prod_{k=0}^{\infty}\left(1 - \frac{1}{2^{2^k}}\right) = (1-\frac{1}{2})(1-\frac{1}{4})(1-\frac{1}{16})\cdots.$$

To show that $\Delta(\frac{1}{2}) > 0$ we use the property that

$$1 - \sum_{k=0}^{\infty}\frac{1}{2^{2^k}} \leq \prod_{k=0}^{\infty}(1-\frac{1}{2^{2^k}}).$$

The infinite sum on the left hand side converges to a number less than 1, since

$$\sum_{k=0}^{\infty}\frac{1}{2^{2^k}} < \sum_{k=1}^{\infty}\frac{1}{2^k} = 1.$$

This shows that

$$\prod_{k=0}^{\infty}(1-\frac{1}{2^{2^k}}) > 0.$$