Ministry of Higher Education and Scientific Research
University of Saida - Dr. Tahar Moulay
Faculty of Sciences

Computer Science department

Course - Algorithms and complexity

Edited by:

Dr. MEKKAOUI Kheireddine

Juin 2024

Thanks

I would like to extend my deepest gratitude to Dr. TEGGAR Hamza, Associate Professor at the University of Mascara, and Dr. MANKOUR Mansour, Associate Professor at the University of Saida, for generously agreeing to review this course booklet. I am sincerely appreciative of their interest in this manuscript and for the considerable time they dedicated to its review. Their invaluable feedback and insights have greatly enhanced the quality of this work.

I am also indebted to the numerous individuals whose contributions, whether direct or indirect, have been instrumental in the completion of this project. Your support, guidance, and constructive input have been invaluable throughout this endeavor.

This course material is a culmination of collective effort, and I extend my heartfelt thanks to everyone who has contributed to its development.

Foreword

This course serves as a fundamental introduction to algorithms and complexity, essential fields in computer science. Structured into five chapters, it aims to provide students with a solid and practical foundation in these crucial areas.

The first chapter, "A Review of Algorithmics and Complexity," lays the groundwork by presenting precise definitions and essential concepts. This chapter is designed to help students understand the fundamental principles underlying the analysis and design of algorithms, as well as various aspects of algorithmic complexity.

The second chapter, "Sorting Algorithms," delves deeply into different sorting techniques. These algorithms are critical tools for any computer scientist, enabling efficient data manipulation by organizing data into specific orders. The chapter covers classical algorithms such as insertion sort, selection sort, bubble sort, as well as more advanced approaches like quicksort and mergesort.

The third chapter focuses on trees and tree structures. Trees are fundamental data structures in computer science used to efficiently organize and hierarchically structure data. This chapter explores different types of trees, their properties, and their use in applications such as advanced data structures and search and traversal algorithms.

In the fourth chapter, we delve into the field of graphs. This chapter introduces essential methods for graph representation, such as adjacency lists and adjacency matrices, as well as fundamental algorithms for solving complex graph problems. Special attention is given to shortest path algorithms, a crucial issue in practical applications such as transportation networks and social networks.

Contents

1
General Introduction

What's a computer and what can be found in it?

A keyboard, a screen, a motherboard, a processor, memory, a hard drive, peripherals (floppy disk drive, CD, DVD, printers, etc.). In short, lots of things, but the only truly essential ones are the processor (which performs basic arithmetic operations), the memory (where the processor can store intermediate results), and the piece of wire between the two (also called the "bus") which is found on the motherboard.

How is it possible to accomplish such complex tasks with a processor that essentially only knows how to perform basic arithmetic operations?

At a slightly higher level, we will focus in this course on designing algorithms (i.e., methods) expressed in an elementary and "understandable" language for a computer. Therefore, we will use a high-level pseudo-language. Once this pseudo-language is mastered, we will translate our algorithms into a programming language and perform a compilation.

What is compilation and how is it that the same language can be used on so many different processors (i386, PowerPc, Atari, Sparc, Mips, etc.)?

As we mentioned, each processor is only capable of performing elementary tasks. To complicate matters, each processor is used with a machine language, and there are different types of machine languages. A compiler transforms a program written in a programming language (such as C) into machine language. Therefore, there are as many compilers as there are families of processors, languages, and operating systems.

But before programming, it is necessary to define an algorithm. An algorithm is a well-defined computational procedure that takes an input value (or set of values) and produces an output value or set of values. Therefore, an algorithm is a sequence of computational steps that allows one to move from the input value to the output value. An algorithm is used to solve a combinatorial problem. The problem statement specifies in general terms the desired relationship between the input and output. The algorithm describes a computational procedure to establish this relationship.

What is an Algorithm?

Here are several definitions found in the literature to define the word "algorithm":

1. An algorithm is a method for solving a given problem in a finite amount of time.
2. According to Le Petit Robert: Set of operational rules specific to a calculation. Calculation, sequence of actions necessary to accomplish a task.
3. According to the Grand Dictionnaire Terminologique (www.granddictionnaire.com): Set of operational rules that allow the resolution of a problem by the application of a finite number of sequential computing operations.
4. According to www.dico.com: A set of well-defined rules or procedures that must be followed to obtain the solution to a problem in a finite number of steps. An algorithm can be simple or complex, but it must achieve a solution in a finite number of steps.

Note

An important characteristic of an algorithm is that its execution must terminate after a finite number of steps. This is not true for all programs.

What is the purpose of algorithm analysis?

Primarily, algorithm analysis serves two main purposes:

1. To realize that an algorithm does not perform well (excessive complexity even on small data sets), or does not work at all (incorrect or non-conforming algorithm). This helps avoid unnecessary programming efforts.
2. To demonstrate the existence of discrete objects or structures that satisfy certain properties by proposing a construction algorithm and proving its correctness.

Attention

Some algorithms cannot be analyzed easily. In fact, some very simple algorithms can be extremely difficult to analyze.

Example 1. *Christian GOLDBACH said:*
Every even number greater than two is the sum of two prime numbers. This problem is straightforward to understand, and it appears that Christian Goldbach's assertion is true, but to this day, no one has been able to prove it.
Algorithme Goldbach(n)
Input : an even integer n>2
Output : boolean

```
Begin
Read($n$);
$i := 1$ to $\frac{n}{2} - 1$ Do :
        If $2i + 1$ and $n - (2i + 1)$ are two prime numbers, return True;
        Else return false;
End
```

Analyse

Thus, for $n = 30$, the algorithm will test the primality of:

- 3 and 27? NO;
- 5 and 25? NO;
- 7 and 23? YES.

This algorithm checks whether the even number $n > 2$ is the sum of two prime numbers.

Is it true that: $\forall n > 2$ even; Goldbach(n) = True?
This analysis appears to be very difficult in advance because it is an unsolved problem kn wn as the Goldbach Conjecture. It has been verified for all integers $n < 1.1 \times 10^{18}$ (February 2008)

Example 2. Algorithme Syracuse(n);

```
Begin
Lire(n);
While n > 1 Do :
        if $n$ is an even integer, then $n := \frac{n}{2}$ else $n := 3n + 1$;
End
```

Is it true that :$\forall n \in N$, Syracuse(n) halts?
It is an unresolved problem (also known as the "$3x + 1$" problem). It holds true for every integer $n < 2^{62}$ (January 2008 - T. Oliveira e Silva), which is greater than four billion billion!

In general, it is undecidable to determine whether a given program or algorithm halts or loops indefinitely. Therefore, one should not expect to find a systematic technique (an algorithm, hence) that analyzes every algorithm: it can be proven that such a method does not exist!

Algorithm and Program

An algorithm is not a program. An algorithm describes a method that will then be implemented in a programming language. The development of a program is divided into several phases, and algorithmics is at the conceptual level.

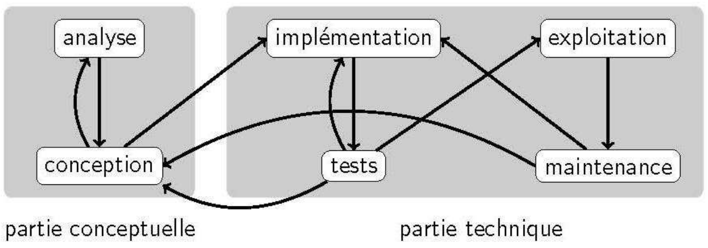


Figure 1.1: Phase de conception des programmes

In practice, things are often more complicated.

An algorithm must be translated into a programming language to produce a compilable and executable program. However, this does not necessarily mean that a program defines an algorithm.

- A system is said to be ***reactive*** when it maintains constant interaction with its environment and its behavior is event-driven. Events are linked to either internal or external stimuli, or constraints related to the passage of time. The objective of a reactive system is thus not to produce a final ultimate result but to interact with its environment. Telecommunication networks, operating systems, process control systems, embedded systems, human-machine interfaces, etc., are examples of reactive systems.

- A system is said to be ***functional or transformational*** (in English, *functional or transformational*) when it generates a set of outputs from input data and then terminates its execution. Such systems are also termed input-output driven. Batch processing systems are examples of functional systems.

From these definitions and examples, we can understand that a reactive program should never terminate its execution and therefore does not implement an algorithm. However, this does not prevent such a program from using and implementing various algorithms within its tasks.

From Reasoning to Algorithm and then to Code

Example task: deciding if a list L is sorted.

L is sorted if all its elements are in ascending order. More formally:

$$L \text{ is sorted if } \forall\ 0 \leq i \leq \|L\| - 1 : L[i] \leq L[i+1]$$

From this, we derive the following algorithm: assume the list is sorted initially and look for a contradiction.

```
def is_sorted(L):
    for i in range(len(L) - 1):
        if L[i] > L[i + 1]:
            return False
    return True
```

Programming Paradigms and Architectural Model

An algorithm must be translated into a programming language to effectively solve problems using computer programs and systems.

A program is a set of components written in one or more programming languages and executable on a particular type of machine. A program involves many details and is not as abstract as an algorithm.

A typical characteristic of an algorithm is that it is a solution expressed relatively abstractly, independent of a particular language or compiler, without reference to a specific machine. However, when considering various approaches to solving algorithmic problems, it becomes apparent that an algorithm cannot be independent:

- from the choice of a programming style (paradigm)
- from the choice of an architectural model (to which the algorithm will ultimately be translated and executed)

In other words, to paraphrase the saying "*when the only tool you have is a hammer, every problem looks like a nail*," if the only architectural model you know is the von Neumann architecture, then you see sequential and imperative algorithms everywhere.

The Importance of Developing Efficient Algorithms

For many years, machines have become faster and more powerful, with increasingly more memory. However, the problems we tackle are often more complex, thus more demanding in terms of time and resources. The choice of an efficient algorithm remains, and will always remain, crucial!

1. Algorithm: set of operational rules whose application solves the problem through a finite number of operations.
    - The major concern to consider is execution time, thus minimizing processing.
2. Memory space: size of data used in processing and problem representation.
    - The major issue to address is minimizing memory space occupied.

It is evident that the concepts of processing and memory space are interconnected. These two criteria should guide the choice of a data structure.

2
Algorithmic complexity

2.1 Introduction

2.1.1 What is an Algorithm?

Which to choose: a fast algorithm or a fast machine?

For many years, machines have become faster and more powerful, with increasing memory capacities. However, the problems we tackle are often becoming more complex, hence more demanding in terms of time and resources. Therefore, the choice of an efficient algorithm remains, and will always remain, crucial!

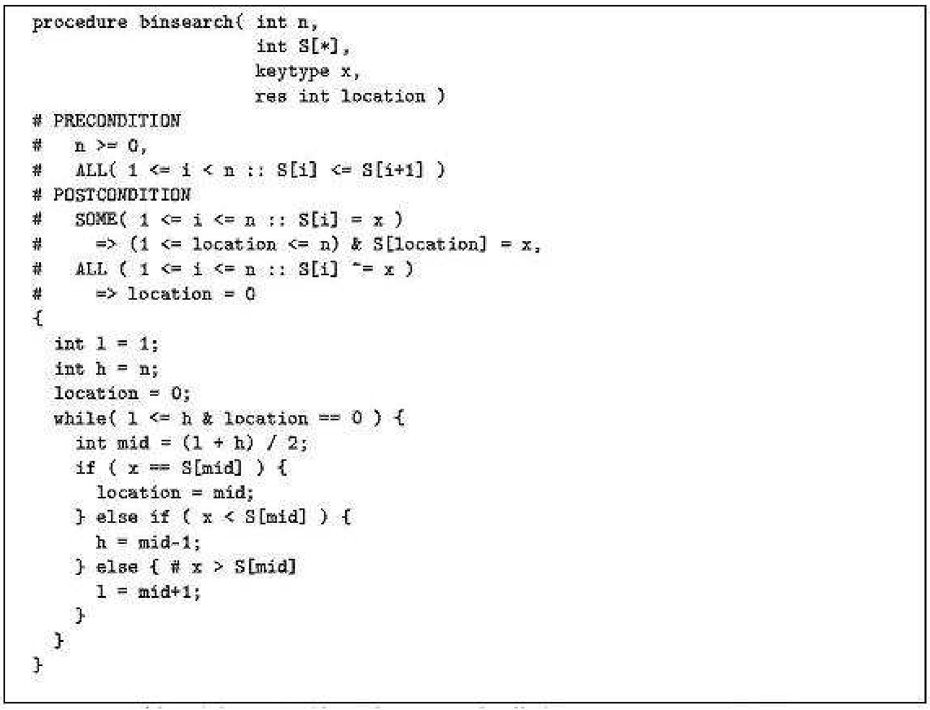
1. Algorithm: a set of operational rules whose application solves the problem through a finite number of operations.
    - The major concern to consider is execution time, thus minimizing processing.
2. Memory space: size of data used in processing and problem representation.
    - The major issue to address is minimizing occupied memory space.

It is clear that processing and memory space are interconnected. These two criteria should guide the choice of data structure and comparison between algorithms.

Example 3. Sequential Search Vs Binary Search

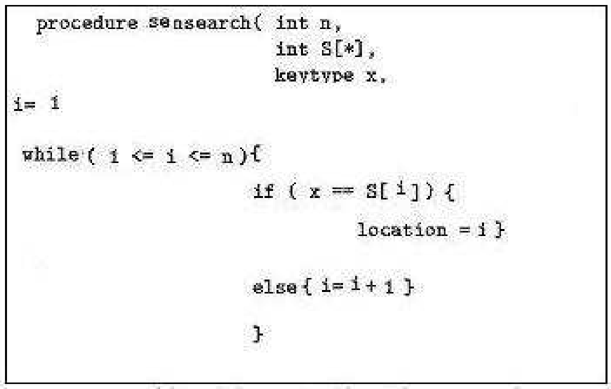
Here is a comparison between two algorithms performing the same task, which is searching for an element in an array:

The number of comparisons performed (in the worst case) by sequential search compared to binary search is presented in Table 2.1:



```
procedure binsearch( int n,
                     int S[*],
                     keytype x,
                     res int location )
# PRECONDITION
#   n >= 0,
#   ALL( 1 <= i < n :: S[i] <= S[i+1] )
# POSTCONDITION
#   SOME( 1 <= i <= n :: S[i] = x )
#     => (1 <= location <= n) & S[location] = x,
#   ALL ( 1 <= i <= n :: S[i] ~= x )
#     => location = 0
{
  int l = 1;
  int h = n;
  location = 0;
  while( l <= h & location == 0 ) {
    int mid = (l + h) / 2;
    if ( x == S[mid] ) {
      location = mid;
    } else if ( x < S[mid] ) {
      h = mid-1;
    } else { # x > S[mid]
      l = mid+1;
    }
  }
}
```

Figure 2.1: Binary Search Algorithm



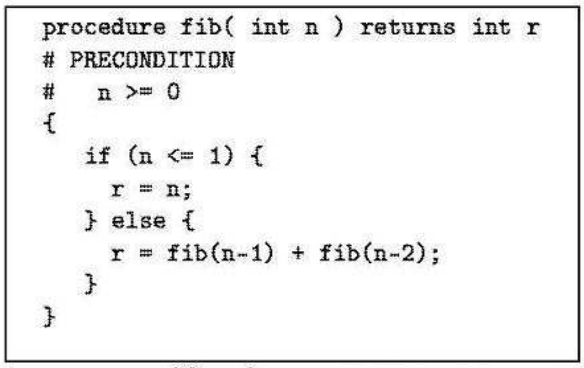
```
  procedure sensearch( int n,
                       int S[*],
                       keytype x,
i= 1

 while ( 1 <= i <= n ){
                      if ( x == S[ i ]) {
                               location = i }

                      else{ i= i + 1 }

                      }
```

Figure 2.2: Sequential Search Algorithm

Array Size $S[*]$	Sequential Search	Binary Search
128	128	8
2014	1024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

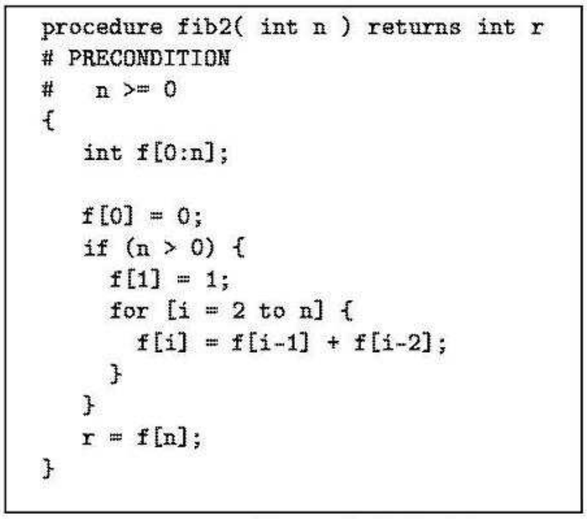
Table 2.1: Sequential Search Vs Binary Search

Example 4. Fibonacci Numbers:



```
procedure fib( int n ) returns int r
# PRECONDITION
#   n >= 0
{
   if (n <= 1) {
     r = n;
   } else {
     r = fib(n-1) + fib(n-2);
   }
}
```

Figure 2.3: Recursive Algorithm for Computing the nth Fibonacci Number



```
procedure fib2( int n ) returns int r
# PRECONDITION
#   n >= 0
{
   int f[0:n];

   f[0] = 0;
   if (n > 0) {
     f[1] = 1;
     for [i = 2 to n] {
       f[i] = f[i-1] + f[i-2];
     }
   }
   r = f[n];
}
```

Figure 2.4: Iterative Algorithm for Computing the nth Fibonacci Number

Execution times of the recursive and iterative versions for different values of n, assuming each term can be computed in 1 ns (10^{-9} sec), are presented in Table 2.2:

n	fib	fib2
40	1,048,000 ns	41 ns
80	18 minutes	81 ns
160	38,000,000 years	161 ns

Table 2.2: Recursive Fibonacci Vs Iterative Fibonacci

2.2 Analysis of Algorithms

2.2.1 Definition

Analyzing an algorithm involves determining, in a relatively abstract manner (i.e., independent of a specific language or machine), its efficiency in terms of time and space.

Analyzing an algorithm means predicting the resources (i.e., time and memory) required by the algorithm and measuring its execution time, potential for parallelism, and the mathematical tools used. Generally, when analyzing multiple candidate algorithms for a given problem, it is easy to identify the most efficient one. This type of analysis can reveal several viable candidates and help eliminate others.

Two types of analysis are considered:

- **Empirical Analysis:** This involves studying the complexity of an algorithm, in terms of time and space, experimentally. The algorithm must be implemented and tested on a set of inputs of varying sizes and compositions. However, this method only allows comparison of algorithms under the same computational environment. Additionally, the results obtained may not be representative for all inputs.

- **Theoretical Analysis:** This evaluates the pseudo-code of the algorithm, the execution time of an algorithm on an instance (i.e., a specific set of data), and the number of elementary operations performed, known as instructions. The memory space of an algorithm on an instance is the number of elements manipulated in memory at a given point during resolution. The execution time for any input is estimated by an upper bound: the worst-case execution time, or in other words, the longest execution time for any input. Execution time can sometimes be estimated using average (or expected) execution time.

2.2.2 Analysis of Time Complexity

- Analyzing the time complexity of an algorithm $\neq$ exact analysis of the execution time of the associated program because it depends too much on the language, compiler, and machine used.
- Time complexity analysis = determining, independently of language and machine, the number of elementary operations (basic operations) needed to solve a problem based on its size.
- The choice of elementary operations to analyze or count and the parameter determining the problem size to be resolved depend on the problem being tackled.

Example 5. :

1. Sorting an array of arbitrary integers:
   - Elementary operations: comparisons between two elements (because the total number of operations will roughly be proportional to the number of comparisons).
   - Size: number of elements in the array.
2. Matrix multiplication:
   - Elementary operations: multiplications and additions.
   - Size: size of matrices (number of rows/columns).

According to **Brassard** and **Bratley** (1996, p. 54), "*an elementary operation is one whose execution time can be bounded by a constant that depends only on the machine, programming language, etc. Thus, this constant does not depend on the size of the problem or any other instance parameters*".

Some authors discuss the use of a benchmark operation to analyze an algorithm: "*A benchmark operation is an operation that is executed at least as often as any other instruction in the algorithm*". Since we are only interested in obtaining the complexity order (magnitude), there is therefore no problem if a non-benchmark operation is executed a constant number of times. In other words, using a benchmark operation avoids explicitly manipulating multiplicative constants in the complexity order definition.

In most algorithms, but not all, the number of operations performed depends not only on the size of the data but also on the data itself. For example:

- Sum of elements in an array of size n: number of additions $= (n-1)$ operations regardless of the content of the array.
- Sequential search:
  - The searched element is first at the beginning of the list: 1 comparison.
  - The searched element is last at the end of the list: n comparisons.

2.3 Worst, Average, and Best Case Complexity

Let n be the size of a problem and $T(n)$ the exact number of elementary operations performed by the algorithm for a problem of size n. Various types of time complexity analysis are as follows:

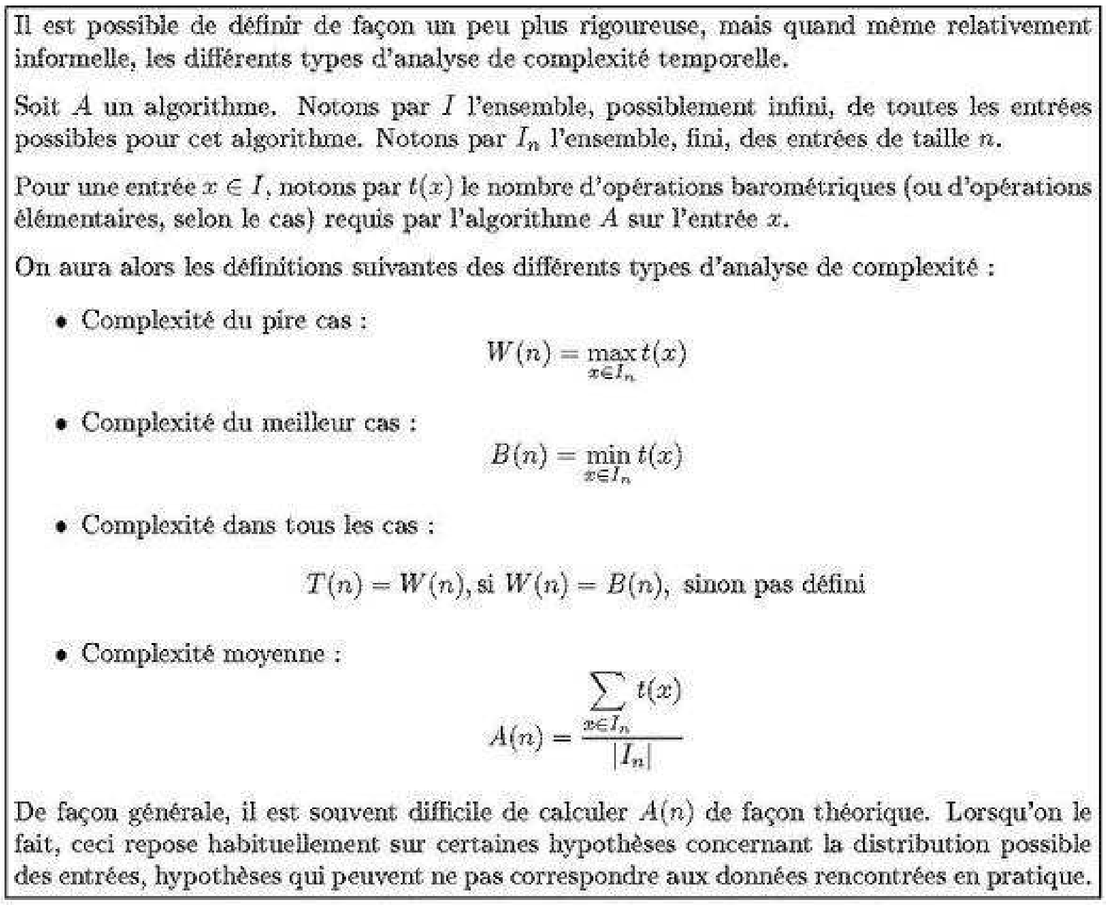
1. Every case complexity: $T(n)$ for any n.
2. Worst-case complexity $W(n)$: number of operations in the worst-case scenario (i.e., maximum number of operations required).
3. Best-case complexity $B(n)$: opposite of worst case.
4. Average complexity $A(n)$: average (expected) number of elementary operations required for a problem of size n.

Example 6. 1. Every case complexity for matrix multiplication (no worst or best case).

2. Worst-case complexity for sequential search.
3. Average-case complexity for sequential search.

Types of Analysis Most Commonly Used:

1. Worst-case: generally the easiest to determine.
2. Average time: more complex to determine because we must associate a probability distribution with the various possible input data.



Il est possible de définir de façon un peu plus rigoureuse, mais quand même relativement informelle, les différents types d'analyse de complexité temporelle.

Soit A un algorithme. Notons par I l'ensemble, possiblement infini, de toutes les entrées possibles pour cet algorithme. Notons par I_n l'ensemble, fini, des entrées de taille n.

Pour une entrée $x \in I$, notons par $t(x)$ le nombre d'opérations barométriques (ou d'opérations élémentaires, selon le cas) requis par l'algorithme A sur l'entrée x.

On aura alors les définitions suivantes des différents types d'analyse de complexité :

- Complexité du pire cas :
$$W(n) = \max_{x \in I_n} t(x)$$
- Complexité du meilleur cas :
$$B(n) = \min_{x \in I_n} t(x)$$
- Complexité dans tous les cas :
$$T(n) = W(n), \text{si } W(n) = B(n), \text{ sinon pas défini}$$
- Complexité moyenne :
$$A(n) = \frac{\sum_{x \in I_n} t(x)}{|I_n|}$$

De façon générale, il est souvent difficile de calculer $A(n)$ de façon théorique. Lorsqu'on le fait, ceci repose habituellement sur certaines hypothèses concernant la distribution possible des entrées, hypothèses qui peuvent ne pas correspondre aux données rencontrées en pratique.

2.4 Asymptotic Notation

Analyzing an algorithm, even a simple one, can prove to be difficult. Therefore, it is necessary to provide ourselves with mathematical tools to achieve our goals.

The execution time of an algorithm A can be expressed as a function $f(n)$: $\aleph \to \Re^+$ where $f(n)$ represents the maximum time taken by A on an input of length n.

If we want to compare the execution times of multiple algorithms, it is first necessary to compare the functions themselves. To achieve this goal, we will define a notation that will be very useful later.

2.4.1 Big-O Notation

The Big-O notation (*Big-O*) was first introduced in the book "Analytische Zahlentheorie" by P. Bachmann in 1892. This notation is widely used to compare the efficiency of algorithms.

Let f and g be two functions from $\aleph \rightarrow \Re^+$ where $\Re^+$ denotes the set of positive real numbers. Indeed, since we will use this notation to compare the asymptotic behaviors of computing times, we can assume without loss of generality that the functions f and g take positive values.

Definition 1. We say that $f(n)$ is $\mathcal{O}(g(n))$, or $f(n) = \mathcal{O}(g(n))$, and sometimes we use the notation $f(n) \in \mathcal{O}(g(n))$, if f is eventually bounded by a multiple of g (see Figure 2.5), that is, if there exists a positive real constant $c > 0$ and a non-negative integer $n_0 \geq 1$, such that:

$$\exists c > 0, n_0 \geq 1, \text{ such that for all } n \geq n_0, \quad f(n) \leq cg(n)$$

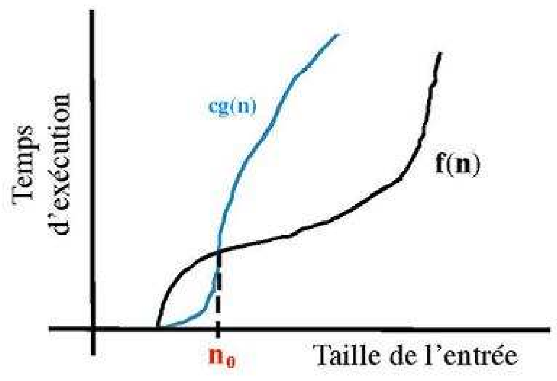


Figure 2.5: Big-O Notion

Example 7. Consider the functions f and g defined by $f(n) = 10n$ and $g(n) = n^2$. We will demonstrate that the relation $f = \mathcal{O}(g)$ is indeed verified.
Here is a table showing the initial values of the functions f and g.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$f(n)$	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150
$g(n)$	0	1	4	9	16	25	36	49	64	81	100	121	144	169	196	225

We observe that $f(n) \leq g(n)$ for all integers $n \geq 10$. Indeed, we have $f(n) = 10n \leq n^2 = g(n)$ for $n \geq 10$. Thus, we can take the constant c equal to 1 and the integer $n_0 = 10$. Note that if we take the constant $c = 1$, the inequality $f(n) \leq cg(n)$ is satisfied for all integers $n \geq 0$. In this case, we can take the integer $n_0 = 1$.

- Intuitively, this means that f does not grow faster than g. We say that f is dominated by g.

- The bound n_0, called the threshold, allows us to ignore the behavior of the functions for small input sizes (in these cases, the algorithm's complexity is often dominated by initialization operations that become negligible for larger data).

- The constant c, called the factor, allows us to abstract away from the speed of the machine used.

- The numbers n_0 and c are called witnesses of the relation $f(n) \in \mathcal{O}(g(n))$.

- The *big-$\mathcal{O}$* notation gives an upper bound on the growth rate of a function.

Example 8.
- $10n + 10n^2 \in \mathcal{O}(n^2)$ because for every $n \geq 1$, we have: $10n + 10n^2 \leq 11n^2$, hence with $c = 11$ and $n_0 = 1$.
- $10n + 10n^2 \in \mathcal{O}(n^2)$ because for every $n \geq 10$, we have: $10n + 10n^2 \leq 2n^2$, hence with $c = 2$ and $n_0 = 10$.

Remark

There is no requirement for optimization (finding the smallest c or n_0 that works), just to provide values that work; Therefore, to prove that $f(n)$ is $\mathcal{O}(g(n))$, it suffices to find a pair of witnesses among infinitely many possible choices. Intuitively, we can distinguish two possible cases:

1. First case (Figure 2.6): From a certain point n_0, $\forall n \geq n_0 : f(n) \leq g(n)$, in this case, the appropriate c is $c = 1$.

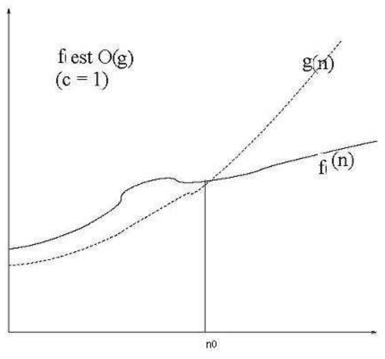


Figure 2.6: $f(n) \in \mathcal{O}(g(n))$: f is smaller than g from a point n_0

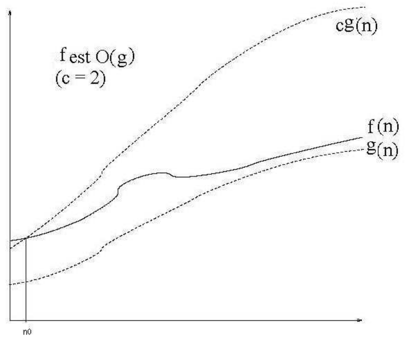


Figure 2.7: $f(n) \in \mathcal{O}(g(n))$: f is not initially smaller than g, but becomes so if g is inflated by a constant factor

2. Second case (Figure 2.7): From a certain point n_0, $g(n)$ is never more than c times smaller than $f(n)$; in other words, if we inflate $g(n)$ by a constant factor c, then from a certain point n_0: $f(n)$ becomes smaller or equal to $c \times g(n)$.

Example 9.

- $n^2 + 10n \in \mathcal{O}(n^2)$, because for $n > 1$ we have: $n^2 + 10n \leq n^2 + 10n^2 = 11n^2$, so $c = 11$ and $n_0 = 1$.
- $n^2 + 10n \in \mathcal{O}(n^2)$, because for $n > 10$ we have: $n^2 + 10n \leq n^2 + n^2 = 2n^2$, so $c = 2$ and $n_0 = 10$.
- $n \in \mathcal{O}(n^2)$, because for $n > 1$ we have: $n \leq n^2 \times 1$, so $c = 1$ and $n_0 = 1$.

Example 10. Let's prove that the function $f(n) = 6n^2 + 2n - 8$ is in $\mathcal{O}(n^2)$:

1. First, let's try to find the constant c; $c = 6$ doesn't work, so let's try $c = 7$;

2. Then we must find a threshold $n_0 \in N$ as of which: $6n^2+2n-8 \leq 7n^2 \;\; \forall n \geq n_0$;
3. A simple calculation gives us $n^2-2n+8 \geq 0 \;\; \forall n \geq 0$ So we can take $n_0 = 1$;
4. In conclusion, $c = 7$ and $n_0 = 1$ give us the desired result;

Example 11. To prove that $f(n) = 5n^2+6n+9$ is $\mathcal{O}(n^2)$, we can provide witnesses such as $c = 6$ and $n_0 = 8$ or $c = 10$ and $n_0 = 2$ as shown in figure 2.8.

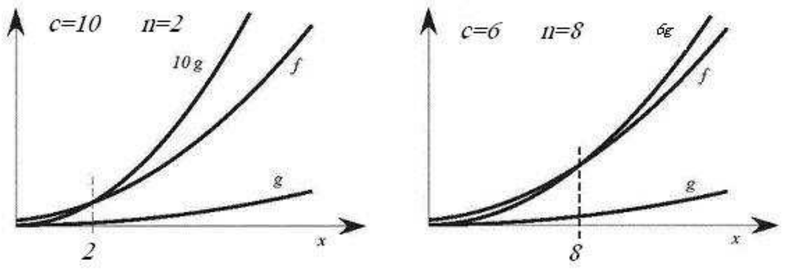


Figure 2.8: Example of choice of witnesses

Example 12. Show that $7n^2$ is $\mathcal{O}(n^3)$ by providing the witnesses.

Response

For $7n^2$ to be $\mathcal{O}(n^3)$ we can choose $c = 1$ and $n_0 = 7$ It can be difficult to find the witnesses n_0 and c in a big $\mathcal{O}$ relation. In practice, the value of these witnesses is not important. What matters is knowing whether $f(n)$ is $\mathcal{O}(g(n))$.

2.4.2 Properties

Here are some theorems that avoid the need to find witnesses:

Property 1. The notation $\mathcal{O}(g(n))$ describes the asymptotic behavior of a function. In other words, when n is sufficiently large, the ratio $\frac{f(n)}{g(n)}$ always remains bounded by a constant.

theorem 1. $f(n) \in \mathcal{O}(g(n))$ means that for sufficiently large values of n, the ratio $\frac{f(n)}{g(n)}$ is always bounded.

Property 2. If: $\lim n \to \infty \frac{f(n)}{g(n)} = b$ and $b > 0$ then $f(n) \in \mathcal{O}(g(n))$ and $g(n) \in \mathcal{O}(f(n))$.

Property 3. If: $\lim n \to \infty \frac{f(n)}{g(n)} = 0$ then $f(n) \in \mathcal{O}(g(n))$ and $g(n) \notin \mathcal{O}(f(n))$.

Example 13. Let $f(n) = 7n^2$ and $g(n) = n^3$.

- Is $f(n) \in \mathcal{O}(g(n))$?
- Is $g(n) \in \mathcal{O}(f(n))$?

$$\begin{aligned} \lim n \to \infty \frac{f(n)}{g(n)} &= \lim n \to \infty \frac{7n^2}{n^3} \\ &= \lim n \to \infty \frac{7}{n} \\ &= 0 \end{aligned}$$

thus $7n^2 \in \mathcal{O}(n^3)$ but $n^3 \notin \mathcal{O}(7n^2)$.

Example 14. Let $f(n) = \log_2(n)$ and $g(n) = n$.

$$\begin{aligned} \lim n \to \infty \frac{f(n)}{g(n)} &= \lim n \to \infty \frac{\log_2(n)}{n} \\ &= \lim n \to \infty \frac{\frac{\ln n}{ln 2}}{n} \text{ by the property : } \log_a(n) = \frac{\ln n}{\ln a} \\ &= \lim n \to \infty \frac{\ln n}{n \ln 2} \\ &= \lim n \to \infty \frac{1}{n \ln 2} \text{ by l'Hopital's rule } \lim n \to \infty \frac{f(n)}{g(n)} = \lim n \to \infty \frac{f'(n)}{g'(n)} = \\ &= 0. \end{aligned}$$

thus: $\log_2(n) \in \mathcal{O}(n)$ but $n \notin \mathcal{O}(\log_2(n))$.

Example 15. Let $f(n) = 2^n$ and $g(n) = 3^n$.

$$\begin{aligned} \lim n \to \infty \frac{f(n)}{g(n)} &= \lim n \to \infty \frac{2^n}{3^n} \\ &= \lim n \to \infty (\frac{2}{3})^n \\ &= 0. \end{aligned}$$

thus: $2^n \in \mathcal{O}(3^n)$ but $3^n \notin \mathcal{O}(2^n)$.

Property 4. In the list of functions below, each function is big-$\mathcal{O}$ of the functions to its right but is not big-$\mathcal{O}$ of the functions to its left:

$$1,\ \log(n),\ n,\ n\log(n),\ n^2,\ n^3,\ n^4,\ \ldots,\ 2^n,\ 3^n,\ 4^n,\ \ldots,\ n!,\ n^n.$$

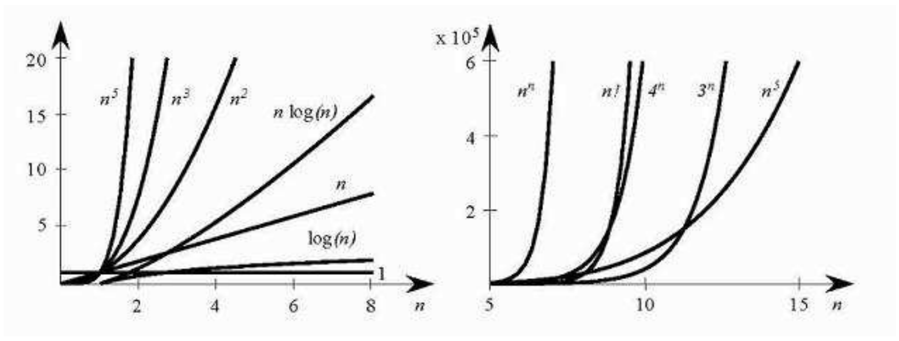


Figure 2.9: Comparison of functions in the list

Figures 2.9 and 2.10 illustrate the growth of the functions in the list.

An exponential with base b greater than 1 will always eventually exceed a polynomial $p(n)$, even if the base is very close to 1 and the degree of the polynomial is very large (see figure 2.10):

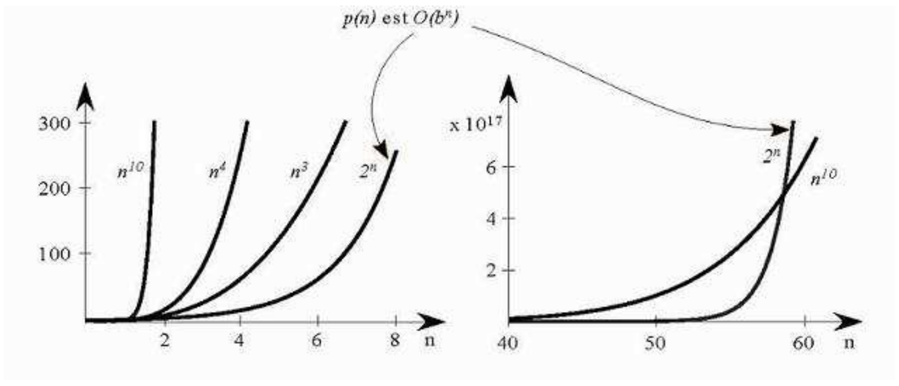


Figure 2.10: Comparison between $p(n)$ and b^n

Property 5. Scale of comparison
Here is a list of categories of functions commonly used in analysis. The functions are ranked in order of growth from slowest to fastest, as the variable $n \to +\infty$. c is an arbitrary constant.

Notation	**Growth**
$\mathcal{O}(1)$	constant
$\mathcal{O}(\log(n))$	logarithmic
$\mathcal{O}((\log(n))^c)$	polylogarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n\log(n))$	sometimes called linearithmic, or quasilinear
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(n^c)$	polynomial, sometimes geometric
$\mathcal{O}(c^n)$	exponential
$\mathcal{O}(n!)$	factorial

This list is useful because of the following property: if a function f is a sum of functions, and if one of the functions in the sum grows faster than the others, then the fastest-growing function determines the order of $f(n)$.

Property 6. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $(f_1 + f_2)(n) \in \mathcal{O}(g_1(n) + g_2(n))$

Property 7. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $(f_1 + f_2)(n) \in \mathcal{O}(max(g_1(n), g_2(n)))$

Property 8. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $(f_1 \times f_2)(n) \in \mathcal{O}(g_1(n) \times g_2(n))$

Property 9. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ and if $g_1(n) \in \mathcal{O}(g_2(n))$ then $(f_1 + f_2)(n) \in \mathcal{O}(g_2(n))$

Property 10. If $p(n)$ is a polynomial of degree d then $p(n) \in \mathcal{O}(n^d)$

Property 11. Let $f, g, h : \aleph \to \Re^+$ be three positive functions. If $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(h)$ then $f \in \mathcal{O}(h)$.

Property 12. Let $f, g : \aleph \to \Re^+$. If $f \in \mathcal{O}(g)$ then $\mathcal{O}(f) \subseteq \mathcal{O}(g)$.

Property 13. Let $p(n)$ and $q(n)$ be two polynomials of degree $d \geq 0$. Then $\mathcal{O}(p(n)) = \mathcal{O}(q(n))$.

Example 16. Let $f(n) = n^2 + 5^n$.

Find a function $g(n)$, the simplest possible, such that $f(n) \in \mathcal{O}(g(n))$.

According to the list, it is the exponential 5^n that dominates the power n^2, in the sense that $n^2 \in \mathcal{O}(5^n)$. Thus: $n^2 + 5^n \in \mathcal{O}(5^n)$.

The function sought is therefore $g(n) = 5^n$.

Example 17. Let $f(n) = \sqrt{7}n^6 + 7n^5 + \pi n^3 - 194n^2 - 2112$.
The function f is a polynomial of degree 6. So:
$\sqrt{7}n^6 + 7n^5 + \pi n^3 - 194n^2 - 2112 \in \mathcal{O}(n^6)$.

Example 18. Let $f(n) = 4n^2 + 3n + 7 + 6 \times 3^n + 5\log(n)$.

Find a function $g(n)$, the simplest possible, such that $f(n) \in \mathcal{O}(g(n))$.

We know that:

- $4n^2 + 3n + 7 \in \mathcal{O}(n^2)$: because it is a polynomial of degree 2.

- $6 \times 3^n \in (3^n)$: because the limit of the quotient is 6, or by using the witnesses $c = 6$ and $n_0 = 0$.

- $5\log(n) \in \mathcal{O}(\log(n))$: because the limit of the quotient is 5, or by using the witnesses $c = 5$ and $n_0 = 0$.

According to the list in theorem 2, it is 3^n that dominates. Thus, according to the sum theorem: $4n^2 + 3n + 7 + 6 \times 3^n + 5\log(n) \in \mathcal{O}(3^n)$.

The function sought is: $g(n) = 3^n$.

Example 19. Let $f(n) = (14n + 3)\log(n) + 3n^2$.

Find a function $g(n)$, the simplest possible, such that $f(n) \in \mathcal{O}(g(n))$.

We know that:

- $14n + 3 \in \mathcal{O}(n)$: because it is a polynomial of degree 1.
- Therefore, by the product theorem: $(14n + 3)\log(n) \in \mathcal{O}(n\log(n))$
- We have $3n^2 \in \mathcal{O}(n^2)$, $(14n+3)\log(n) \in \mathcal{O}(n\log(n))$ and, looking at the list in theorem 3, we see that n^2 dominates $n\log(n)$.

Thus, according to the sum theorem, $(14n + 3)\log(n) + 3n^2 \in \mathcal{O}(n^2)$.

The function sought is: $g(n) = n^2$.

Example 20.
- Show that: $5n^2 - 3n - 4 \in \mathcal{O}(n^2)$
- Show that: $n^2 \in \mathcal{O}(5n^2 - 3n - 4)$
- What can we deduce?

2.4.3 Samll-o Notation

Definition 2. We say that $f(n)$ is $o(g(n))$, or $f(n) = o(g(n))$, and sometimes use the notation $f(n) \in o(g(n))$, if for any positive real constant $c > 0$, there exists an integer $n_0 \geq 1$ such that:

$$\forall c > 0, \quad \exists n_0 \geq 1, \quad \text{such that} \quad \forall n \geq n_0 \qquad f(n) < cg(n)$$
$$\text{or if and only if} \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- This means that f grows more slowly than g when n is very large.
- $f(n)$ is negligible compared to $g(n)$, since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Example 21.
- x^2 is $o(x^5)$
- $\sin x$ is $o(x)$
- $14.709\sqrt{x}$ is $o\left(\frac{x}{2} + 7\cos x\right)$
- $23 \log x$ is $o(x^{0.002})$

2.4.4 Big-Ω Notation

Definition 3. We say that $f(n)$ is $\Omega(g(n))$, or $f(n) = \Omega(g(n))$, and sometimes use the notation $f(n) \in \Omega(g(n))$, if and only if there exists a positive real constant $c > 0$ and an integer $n_0 \geq 1$ such that:

$$\exists c > 0, n_0 \geq 1, \text{ such that} \quad \forall n \geq n_0 \qquad f(n) \geq cg(n)$$
$$\text{or if and only if} \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Example 22.
- $\sqrt{n} = \Omega(\log n)$.

2.4.5 Small-ω Notation

Definition 4. We say that $f(n)$ is $\omega(g(n))$, or $f(n) = \omega(g(n))$, and sometimes use the notation $f(n) \in \omega(g(n))$, if for any positive real constant $c > 0$, there exists

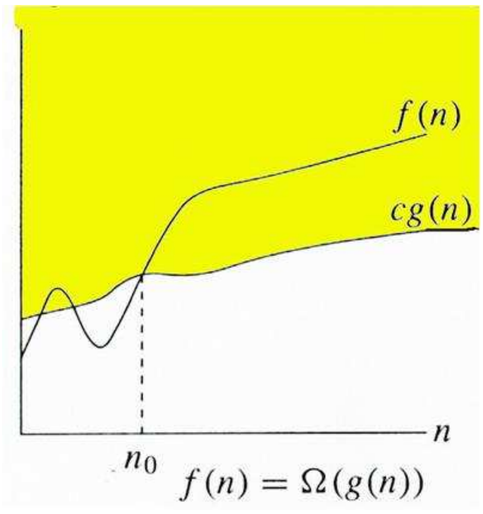


Figure 2.11: $f(n)$ is $\Omega(g(n))$

an integer $n_0 \geq 1$ such that:

$$\forall c > 0, \exists n_0 \geq 1, \text{ such that: } \forall n \geq n_0 \qquad f(n) > cg(n)$$
$$\text{or if and only if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

The graph 2.12 should help us visualize the relationships between these notations: These definitions have more similarities than differences.

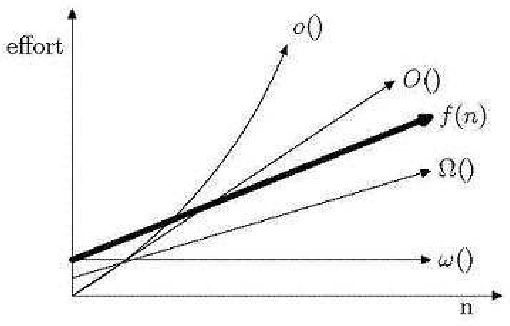


Figure 2.12: The relationship between Big-O notations

2.4.6 Big-Θ Notation

When we say an algorithm $f(n)$ is $\Theta(g(n))$, it is equivalent to saying that $g(n)$ is a strict upper and lower bound on the growth of the effort of $f(n)$.

Definition 5. We say that f is $\Theta(g)$, or $f = \Theta(g)$, and sometimes use the notation $f(n) \in \Theta(g(n))$, if and only if:

$\exists c_1 > 0, c_2 > 0, n_0 \geq 1,$ such that: $\forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$
Or if:
$f(n)$ is $\mathcal{O}(g(n))$ and $f(n)$ is $\Omega(g(n))$.
Or if:
$f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(f(n))$.
Or if:
$0 < \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

Example 23.

- $2n \in \Theta(n)$ but $2n \notin \Theta(n^2)$

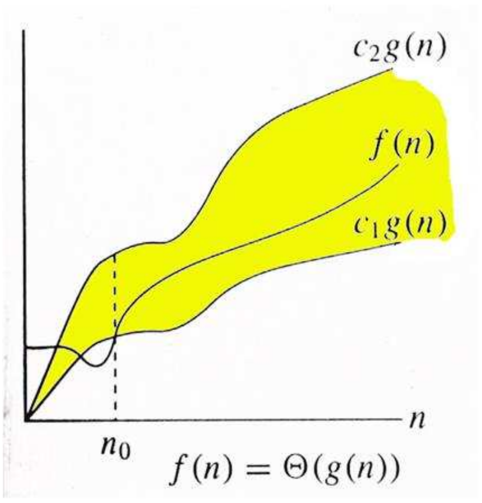


Figure 2.13: $f(n) = \Theta(g(n))$

Example 24.

1. Prove that:
   - $10n^2 - 3n = \Theta(n^2)$
   - $\frac{n^2}{2} - 3n = \Theta(n^2)$
2. Is it true that?
   - $3n^3 \in \Theta(n^4)$?
   - $2^{2n} \in \Theta(2^n)$?

Function Comparison

$f \leftrightarrow g \equiv a \leftrightarrow b$

- $f(n) = \mathcal{O}(g(n)) \equiv a \leq b$
- $f(n) = \Omega(g(n)) \equiv a \geq b$
- $f(n) = \Theta(g(n)) \equiv a = b$
- $f(n) = o(g(n)) \equiv a < b$
- $f(n) = \omega(g(n)) \equiv a > b$

Properties

1. $f(n) \in \mathcal{O}(g(n))$ if and only if $g(n) \in \Omega(f(n))$
2. $f(n) \in o(g(n))$ if and only if $g(n) \in \omega(f(n))$
3. $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$
4. Let the complexity functions be ordered as follows, where $k > j > 2$ and $b > a > 1$:

$$\Theta(1), \Theta(\log n), \Theta(n), \Theta(n \log n), \Theta(n^2), \Theta(n^j), \Theta(n^k), \Theta(a^n), \Theta(b^n), \Theta(n!)$$

Application Examples

Here are some examples that show how the definitions should be applied.

Example 25. Let two functions $f(n) = 7n + 8$ and $g(n) = n$.
Problem: $f(n) = O(g(n))$?
For $f(n) = O(g(n))$, we must find the existence of a number c and n_0 such that: $f(n) \leq cg(n)$ for all $n > n_0$.
Clearly, we can select $c = 8$.
Thus:

$$\begin{aligned} & f(n) \leq cg(n) \\ & \Rightarrow 7n + 8 \leq 8n \\ & \Rightarrow n \geq 8 \end{aligned}$$

So: $f(n) \leq cg(n)$ for $c = 8$ and for all $n \geq 8$.
Thus $f(n) = O(g(n))$.

Example 26. Let the two functions $f(n) = 7n + 8$ and $g(n) = n$.
Problem: $f(n) = o(g(n))$?
For $f(n) = o(g(n))$, we must find n_0 such that $f(n) < c(g(n))$ is asymptotically true for all $c > 0$.
Thus:

$$\begin{aligned}
&f(n) < cg(n)\\
&\Rightarrow 7n + 8 < cn\\
&\Rightarrow (c - 7)n > 8
\end{aligned}$$

The problem is: is the equation verified for all $c > 0$?
Absolutely not.
For instance, take $c = 1$. Then the inequality is not verified.
Thus $f(n) \neq o(g(n))$.

Example 27. Let the two functions $f(n) = 7n + 8$ and $g(n) = n^2$.
Problem: $f(n) = o(g(n))$
For $f(n) = o(g(n))$, we must find n_0 such that $f(n) < c(g(n))$ is asymptotically true for all $c > 0$.
Thus:

$$\begin{aligned}
&f(n) < cg(n)\\
&\Rightarrow 7n + 8 < cn^2\\
&\Rightarrow cn^2 - 7n - 8 > 0
\end{aligned}$$

For the inequality to be verified, we must prove that for all $c > 0$, there exists an n_0 such that:

$$\forall n \geq n_0, cn^2 - 7n - 8 > 0$$

If we calculate Δ, we find: $\Delta = 49 + 32c > 0 \ \forall c > 0$
Let's find n_0 for which the inequality is verified:
After solving the equation, we find: $n_1 = \frac{7-\sqrt{\Delta}}{2c}$ and $n_2 = \frac{7+\sqrt{\Delta}}{2c}$
and depending on the sign of the equation, we find that there exists $n_0 = \frac{7+\sqrt{\Delta}}{2c}$ for the inequality to be verified.
Thus $f(n) = o(g(n))$.

Example 28. We want to show that: $5n + 3\log n + 10n\log n + 7n^2 \in \Theta(n^2)$:

- $7n^2 \in \Theta(n^2)$
- $0n\log n + 7n^2 \in \Theta(n^2)$
- $3\log n + 10n\log n + 7n^2 \in \Theta(n^2)$

- $5n + 3\log n + 10n\log n + 7n^2 \in \Theta(n^2)$

Example 29.

$f(n)$	$g(n)$	$f = \mathcal{O}(g)$	$g = \mathcal{O}(f)$	$f = o(g)$	$g = o(f)$	$f = \Theta(g)$
$n+1$	$n^4 + 3n - 2$	True	False	False	True	False
$n+4$	$4n$	True	True	False	False	True
$\frac{1}{10}n^3 + 100n^2 + 10000$	n^3	True	True	False	False	True
2^{n+1}	2^n	True	True	False	False	True
2^{2n}	2^n	False	True	True	False	False
$(n+a)^b$	n^b	True	True	False	False	True
n^n	$n!$	False	True	True	False	False
2^n	$n!$	True	False	False	True	False
n if $n \equiv 0[2]$ and 1 otherwise	n	True	False	False	False	False

2.5 Time Complexity

The complexity of an algorithm is determined by dividing the algorithm into so-called elementary operations.

2.5.1 Elementary Operations

Basic instructions take constant time, denoted $O(1)$.

Example 30.
- Read
- Write
- Assignment
- ...

2.5.2 Sequence of Instructions:

The execution time of a sequence of instructions is the sum of the execution times of the elementary instructions in the sequence.

$$f(I_1, I_2, \ldots, I_n) = f(I_1) + f(I_2) + \ldots + f(I_n)$$
$$T(n) = \mathcal{O}(f(I_1)) + \mathcal{O}(f(I_2)) + \ldots + \mathcal{O}(f(I_n))$$

Let c_1 and c_2 denote the execution times of the assignment and addition instructions, respectively. Therefore, the execution time is: $T(n) = c_1 + c_1 + c_2 = c$, which is a constant time.

Algorithm 1 Sequence of Instructions

Example 31.
```
1: $x \leftarrow 1$;        $c_1$
2: $y \leftarrow 2$;        $c_1$
3: $z \leftarrow x + y$;    $c_2$
```

So $T(n) = c$ (c is a positive real constant).

Thus we have $T(n) = c = \mathcal{O}(1)$, which is very convenient notation because even if there are 1235 constants, they would all be denoted as $O(1)$.

Thus, Algorithm 1 has a complexity of $\mathcal{O}(1)$, or constant complexity.

2.5.3 Conditional Instructions

The execution time of a conditional instruction is the maximum of the two instructions (direct or indirect - IF ELSE).

$$T(f(\text{if...}\{A\}\ \text{else}\{B\})) = \mathcal{O}(\max(f(A), f(B)))$$

Example 32. Consider the following example:

Algorithm 2 Conditional Instructions
```
1: **if** condition **then**
2:     $A$;                 $t_1$
3: **else**
4:     $B$;                 $t_2$
5: **end if**
```

Let's calculate the complexity of Algorithm 2:

- $T(\text{if}) = T(A) = t_1$
- $T(\text{else}) = T(B) = t_2$

Thus, $T(n) = c + \max(f(A), f(B))$.

2.5.4 Loop Instructions

The execution time of a loop is at most the number of iterations multiplied by the execution time of the elementary instructions in the loop body.

$$\text{if } B \text{ is independent of } i \text{ then}$$
$$T(f(\text{for}(\text{int } i = 1; i <= n; i++)\{B\})) = \mathcal{O}(nf(B))$$
$$\text{if } B \text{ is dependent on } i \text{ then}$$
$$T(f(\text{for}(\text{int } i = 1; i <= n; i++)\{B(i)\})) = \mathcal{O}(\sum_{i=1}^{n} B(i))$$

Example 33. Consider the algorithm for computing power:

Algorithm 3 Loop Instructions

```
1: $p \leftarrow 1$;                 $c_1$
2: for $i = 1$ to $n$ do
3:     $p \leftarrow p \times x$;    $c_2$
4: end for
```

Let's calculate the complexity of Algorithm 3:

- $f(p \leftarrow 1) = \mathcal{O}(1)$
- For the loop, we perform n iterations, each of which executes in $\mathcal{O}(1)$ time, so $f(\text{loop}) = n \times \mathcal{O}(1) = \mathcal{O}(n)$

Thus, $T(n) = O(1) + O(n) = c_1 + c_2 \times n = O(n)$.

Finally, we have $T(n) = O(n)$, indicating that Algorithm$_2$ executes in linear time in n. The computation may seem lengthy, but with time it should go much faster.

Algorithm 4 Example

Example 34.

```
1: sum$\leftarrow$ 0;
2: for $i = 0$ to $n - 1$ do
3:     sum$\leftarrow$ sum$+i \times i$;
4: end for
```

Solution

$T(n) = c_1 + c_1 + n(c_2 + c_3 + c_4 + c_5 + c_1)$
Where c_1: Assignment, c_2: Increment, c_3: Test, c_4: Addition, c_5: Multiplication
$T(n) = 2c_1 + n\sum_{i=1}^{5} c_i$ The algorithm has complexity $\mathcal{O}(n)$ or $\mathcal{O}(cn)$.

2.5.5 Recursion

Algorithm 5 factorial(n)

Example 35.
```
1: **if** $n = 0$ **then**
2:     return 1;
3: **else**
4:     return $n \times$factorial($n - 1$);
5: **end if**
```

Solution

Let c_1: cost of the test, and c_2: cost of the multiplication.

$$T(n) = \begin{cases} c_1 \text{ if } n = 0 \\ c_1 + c_2 + T(n-1) \text{ otherwise} \end{cases}$$

$$T(n-1) = \begin{cases} c_1 \text{ if } n-1 = 0 \\ c_1 + c_2 + T(n-2) \text{ otherwise} \end{cases}$$

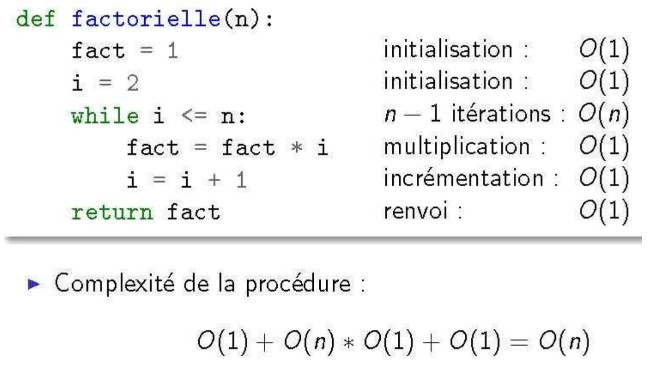
$$T(n-2) = \begin{cases} c_1 \text{ if } n-2 = 0 \\ c_1 + c_2 + T(n-3) \text{ otherwise} \end{cases}$$

$$\vdots$$

$$T(n) = n \times c_2 + (n+1) \times c_1$$

So the complexity is $\mathcal{O}(n)$.

Example Application



```
def factorielle(n):
    fact = 1                  initialisation :    O(1)
    i = 2                     initialisation :    O(1)
    while i <= n:             n − 1 itérations :  O(n)
        fact = fact * i       multiplication :    O(1)
        i = i + 1             incrémentation :    O(1)
    return fact               renvoi :            O(1)
```

► Complexité de la procédure :

$$O(1) + O(n) * O(1) + O(1) = O(n)$$

General Rules

- The execution time of a program depends on:
  1. the number of data,
  2. the size of the code,
  3. the type of computer used (processor, memory),
  4. the time complexity of the underlying abstract algorithm.
- Let n be the size of the data of the problem and $T(n)$ the execution time of the algorithm. We distinguish:
  1. The worst-case time $T_{max}(n)$ which corresponds to the maximum time taken by the algorithm for a problem of size n.
  2. the average time T_{avg} average execution time on data of size n ($\Rightarrow$ assumptions about data distribution).
- The execution time (e.t.) of an assignment or a test is considered constant,
- The time of a sequence of instructions is the sum of the e.t. of the instructions that compose it,
- The time of a conditional branch is equal to the e.t. of the test plus the maximum of the two e.t. corresponding to the two alternatives (in the case of worst-case time).
- The time of a loop is equal to the sum of the cost of the test + the body of the loop + loop exit test.

2.5.6 Estimated Cost vs. Real Cost

The measures presented are only asymptotic estimates of algorithms. In practice, it may sometimes be necessary to reach very large values of n for an $O(n \log n)$ algorithm to outperform a quadratic algorithm.

Complexity analyses can be used to compare algorithms, but the cost model is relatively simple (for example, operations such as disk access or generated network traffic are not taken into account, although these parameters can have a significant influence on a program). It is always necessary to carry out experimental analyses before choosing the best algorithm.

2.5.7 Qualities of an Algorithm

- Qualities:
  1. Maintainable (easy to understand, code, debug),
  2. Fast
- Tips:
  1. Prioritize point 2 over point 1 only if gaining in complexity.
  2. What the algorithm does should be readable in a quick scan: One idea per line.
  3. Also pay attention to accuracy, stability, and security.
- The speed of an algorithm is one element of a whole defining its qualities.

2.5.8 The quality and characteristics of an algorithm

The quality and characteristics of an algorithm are crucial in determining its efficiency, robustness, and suitability for solving a given problem. Here are the main qualities and characteristics of a good algorithm:

Efficiency

- Execution Time Complexity: Efficiency in terms of time complexity is critical, as it determines how quickly an algorithm can solve a problem as the size of the input increases. Common notations like $O(1)$ (constant time), O(n) (linear time),O(n2) (quadratic time), and O(logn) (logarithmic time) help describe an algorithm's time complexity. Algorithms that have lower time complexity are typically preferred for larger datasets, as they can process inputs faster without a significant slowdown.

- Space Complexity: Besides time, space complexity measures how much memory an algorithm uses relative to input size. Space-efficient algorithms use minimal memory, which is important when memory is limited or when the algorithm needs to handle large datasets. Reducing space complexity involves optimizing memory usage, often by reusing memory or using in-place algorithms that modify data directly rather than creating copies.

Accuracy

- Reliability of Results: The algorithm should guarantee correct output every time, assuming valid input. For example, in financial algorithms, small inaccuracies could lead to incorrect predictions or financial loss. Testing for accuracy includes validating the algorithm against a wide range of test cases, including edge cases.

- Numerical Stability: For algorithms involving calculations (e.g., floating-point arithmetic), numerical stability is important. Numerical errors can accumulate with each operation, potentially leading to incorrect results. Stable algorithms manage this by using techniques like scaling, rearranging operations, or choosing stable mathematical methods.

Simplicity

- Readability and Maintainability: A simpler algorithm with clear logic is easier for programmers to understand, maintain, and extend. This often means avoiding unnecessary complexity, which can lead to errors and complicate debugging. Simplicity is also critical when multiple developers work on the same code base since complex algorithms are harder to review and improve.

- Implementation Ease: A simple algorithm should be implementable with minimal lines of code or modularized functions. This also aids in making the code more readable and reduces the risk of errors during coding.

Robustness

- Error Handling: Robust algorithms include error handling to manage unexpected inputs or conditions without crashing. For example, a sorting algorithm that encounters a null input should handle it gracefully instead of generating an error.

- Fault Tolerance: Robust algorithms are designed to be fault-tolerant, meaning they can handle a range of inputs, including extremes, unusual cases, or minor data corruption, without failing. This involves anticipating and coding for special cases, like empty datasets, large integers, or negative values.

Flexibility

- Adaptability to New Requirements: A flexible algorithm can be adapted or expanded to meet new requirements without a complete rewrite. For example, a search algorithm designed to handle basic text might later need to handle multiple languages or complex queries. A flexible design allows easy adjustment to such changes.
- Extensibility for Future Use: Flexibility includes writing an algorithm in a way that supports future extensions. This might mean designing it to work with a variety of input types or data structures, allowing it to be reused in different applications with minimal changes.

Modularity

- Decomposition into Sub-Functions: A modular algorithm divides tasks into smaller functions or modules, each responsible for a specific subtask. For example, a sorting algorithm could separate input validation, comparison operations, and sorting logic into distinct functions. This breakdown improves readability and allows each function to be tested independently.
- Reusability of Modules: Each module in a modular algorithm can often be reused across different parts of a codebase or even in other projects. Reusability reduces the need to rewrite code, which saves time and reduces the likelihood of errors.

Scalability

- Performance with Growing Data: Scalability measures how an algorithm performs as the data size increases. A scalable algorithm maintains reasonable performance even as the data size scales up, which is essential in fields like data science and machine learning. Scalability may involve optimizing algorithms to work with parallel processing, distributed computing, or large-scale storage solutions.
- Efficiency in Distributed Environments: In applications like cloud computing, scalability means that an algorithm can efficiently handle data that is spread across multiple systems. Algorithms that are scalable often incorporate designs that support parallel processing or distributed execution.

Convergence (Especially for Optimization Algorithms)

- Speed of Convergence: For optimization problems, convergence refers to how quickly the algorithm reaches an optimal or satisfactory solution. For example, gradient descent in machine learning iteratively improves towards a minimum error; faster convergence implies fewer iterations and less computation.

- Global vs. Local Convergence: Some algorithms, especially those in optimization, might get "stuck" in local minima rather than finding the global minimum (optimal solution). A good algorithm either ensures global convergence or has mechanisms like random restarts or adjustments to escape local minima.

Summary of Characteristics with Examples

- Efficiency: A quicksort algorithm, with an average time complexity of O(nlogn)O(nlogn), is efficient for sorting large lists.

- Accuracy: A financial forecasting algorithm must provide precise calculations; otherwise, minor errors could lead to significant financial discrepancies.

- Simplicity: Bubble sort is conceptually simple and easy to understand, making it a useful teaching tool, though it is not the most efficient sorting algorithm.

- Robustness: A database search algorithm should handle unexpected inputs (e.g., null or malformed data) without failure.

- Flexibility: A search algorithm designed for text search should be flexible enough to handle advanced search criteria without significant modification.

- Modularity: A large machine learning pipeline often consists of modules for data pre-processing, feature selection, model training, and evaluation, each of which can be reused or tested independently.

- Scalability: MapReduce is an example of a scalable algorithmic framework that allows data to be processed in parallel across large clusters.

- Convergence: Genetic algorithms, used in optimization problems, may converge faster toward optimal solutions with mechanisms to prevent premature convergence to suboptimal solutions.

A good algorithm integrates multiple characteristics based on the problem requirements, ensuring it's both effective and adaptable in real-world applications.

2.6 Recurrence Equations

Reminder

- Asymptotic notation approximates the complexity of algorithms.
- The challenge lies in studying methods to solve recurrence equations.
- The complexity of recursive algorithms is often computable from recurrence equations.

2.6.1 Introduction

Consider the following geometric sequence $(1, 2, 2^2, \ldots, 2^n, \ldots)$. One way to express this sequence of numbers is by defining each nth term in relation to the previous terms, with the first term defined as:

$$\begin{aligned}&t(n) = 2t(n-1), n \geq 1\\&t(0) = 1.\end{aligned}$$

Given a sequence of numbers $(t(1), t(2), \ldots, t(n), \ldots)$, an equation relating the nth term to its predecessors is called a recurrence equation or difference equation. Solving a recurrence equation involves finding an expression for the nth term in terms of the parameter n.

Recurrence equations are divided into two categories:

1. linear
2. non-linear

2.6.2 Linear Equations with Constant Coefficients

Definition 6. A sequence of numbers $(t(1), \ldots, t(n), \ldots)$ satisfies a linear recurrence relation of order k if and only if there exist constants $c_0, \ldots, c_k$ such that:

$$\begin{aligned}&c_k t(n+k) + c_{k-1} t(n+k-1) + \ldots + c_0 t(n) = g(n)\\&t(n_0) = d_0, \ldots, t(n_0+k-1) = d_{k-1}\end{aligned} \tag{2.1}$$

- where
  1. the function $g(n)$ is an arbitrary function of n.
  2. the parameters d_i are constants defining the initial conditions required to start recursion from index n_0.

3. $t(n+k)$ denotes the term determining the order of equation 2.1.

Example 36. The following equation is a linear recurrence equation of order 3.

$$\begin{array}{l} 4t(n+3)+2t(n+1)+t(n)=4n\log n+n+1, \forall n\geq 3 \\ t(1)=1; t(2)=2. \end{array}$$

Note:

If $g(n)=0$ then we say that relation 2.1 is homogeneous. Otherwise, it is called non-homogeneous.

Solving Homogeneous Equations

A linear equation with constant coefficients and homogeneous is of the following form:

$$\begin{array}{l} c_k t(n+k)+c_{k-1}t(n+k-1)+\ldots+c_0 t(n)=0 \\ t(n_0)=d_0,\ldots,t(n_0+k-1)=d_{k-1} \end{array} \tag{2.2}$$

Definition 7. The characteristic equation of relation 2.2 corresponds to the following polynomial equation:

$$c_k r^k + c_{k-1} r^{k-1} + \ldots + c_1 r + c_0 = 0 \tag{2.3}$$

For example, the characteristic equation of the recurrence equation:

$$4t(n+3)+7t(n+1)-t(n)=0$$

is as follows:

$$4r^3+7r-1=0$$

theorem 2. The general solution of equation 2.2 is of the following form:

$$t(n)=\sum_{i=1}^{l}\left\{r_i^n \sum_{j=0}^{m_i-1} a_{ij} n^j\right\} \tag{2.4}$$

- where
  - the parameter l ≤k denotes the number of distinct roots of characteristic equation 2.3.
  - the parameter r_i denotes a root of characteristic equation 2.3.
  - the parameter m_i denotes the multiplicity of root r_i.

– the coefficients a_{ij} are constants determined from initial conditions. *Note that the notation a_{ij} used is for the sake of the formula. In actual calculations, single-index constants can be used, as illustrated in the following example.*

For example, if the roots of the characteristic equation of a recurrence equation $t(n)$ have 3 distinct roots r_1 (triple root), r_2 (double root), and r_3 (single root), then the general solution is given by the following expression:

$$t(n) = r_1^n(a_1 + a_2n + a_3n^2) + r_2^n(a_4 + a_5n) + a_6r_3^n$$

Example 37. To solve the following relation:

$$\begin{array}{l} t(n) = t(n-1) + t(n-2); \forall n \geq 2 \\ t(0) = 0; t(1) = 1. \end{array}$$

The characteristic equation of this relation is:

$$r^2 - r - 1 = 0$$

The simple roots of this equation are:

$$r_1 = \frac{1+\sqrt{5}}{2} \text{ and } r_2 = \frac{1-\sqrt{5}}{2}$$

Therefore, the general solution is:

$$t(n) = a_1r_1^n + a_2r_2^n$$

In other words:

$$t(n) = a_1\left(\frac{1+\sqrt{5}}{2}\right)^n + a_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

The constants a_1 and a_2 are determined by the initial conditions as follows:

$$\begin{array}{l} t(0) = a_1 + a_2 = 0 \\ t(1) = a_1\left(\frac{1+\sqrt{5}}{2}\right) + a_2\left(\frac{1-\sqrt{5}}{2}\right) = 1 \end{array}$$

By solving this system of two equations and two unknowns, we obtain:

$$a_1 = \frac{1}{\sqrt{5}} \text{ and } a_2 = -\frac{1}{\sqrt{5}}$$

The final solution is:

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

Example 38. To solve the following equation:

$$\begin{array}{l} t(n+3) - 7t(n+2) + 16t(n+1) - 12t(n); \forall n \geq 3 \\ t(0) = 0; t(1) = 1, t(2) = 2. \end{array}$$

The characteristic equation of this relation is:

$$r^3 - 7r^2 + 16r - 12 = 0$$

The roots of this equation are:

1. $r_1 = 3$: simple root.
2. $r_2 = 2$: double root.

Therefore, the general solution is:

$$t(n) = a_1 r_1^n + r_2^n (a_2 + a_3 n)$$

In other words:

$$t(n) = a_1 3^n + (a_2 + a_3 n) 2^n$$

The constants a_1, a_2, and a_3 are determined by the initial conditions as follows:

$$\begin{array}{l} t(0) = a_1 + a_2 = 1 \\ t(1) = 3a_1 + 2a_2 + 2a_3 = 1 \\ t(2) = 9a_1 + 4a_2 + 8a_3 = 2. \end{array}$$

By solving this system of three equations with three unknowns, we find:

$$a_1 = -2; \quad a_2 = 2; \quad a_3 = \frac{3}{2}.$$

Therefore, the final solution is:

$$t(n) = -2 \times 3^n + 2^{n+1} + 3 \times n \times 2^{n-1}.$$

Solving Non-Homogeneous Equations

Recall that a non-homogeneous linear recurrence equation with constant coefficients is such that the function $g(n)$ is non-zero. In other words, it takes the following form:

$$\begin{array}{l} c_k t(n+k) + c_{k-1} t(n+k-1) + \ldots + c_0 t(n) = g(n) \\ t(n_0) = d_0, \ldots, t(n_0 + k - 1) = d_{k-1}. \end{array} \tag{2.5}$$

The principle of resolution, adopted in this section, consists first of eliminating the function $g(n)$ and then solving the resulting homogeneous equation. Let's illustrate this process with the following examples.

Example 39. Consider the equation:

$$t(n+2) - t(n+1) - t(n) = 4 \tag{2.6}$$

This relation also holds for $n+1$, meaning:

$$t(n+3) - t(n+2) - t(n+1) = 4 \tag{2.7}$$

Subtracting Equation 2.6 from Equation 2.7, we obtain the new equation:

$$t(n+3) - 2(t(n+2) + t(n)) = 0$$

This new equation is homogeneous. By applying the resolution method discussed earlier, we get the following general solution:

$$t(n) = a_1 + a_2(1+\sqrt{5})^n + a_3(1-\sqrt{5})^n$$

Since initial conditions are not provided, coefficients a_1, a_2, and a_3 cannot be determined.

Example 40. Consider the equation:

$$\begin{aligned} &t(n) = t(n-1) + n \\ &t(0) = 1 \end{aligned} \tag{2.8}$$

This equation also holds for $n+1$, i.e.:

$$t(n+1) = t(n) + n + 1 \tag{2.9}$$

Subtracting Equation 2.8 from Equation 2.9, we obtain:

$$t(n+1) - 2t(n) + t(n-1) = 1 \tag{2.10}$$

For $n+1$, Equation 2.10 can be written as:

$$t(n+2) - 2t(n+1) + t(n) = 1 \tag{2.11}$$

Again, subtracting Equation 2.11 from Equation 2.10, we get:

$$t(n+2) - 3t(n+1) + 3t(n) - t(n-1) = 0 \tag{2.12}$$

with the following initial conditions:

$$t(0) = 0; \quad t(1) = 1; \quad t(2) = 3.$$

Using the method of solving homogeneous linear equations, we obtain the final solution:

$$t(n) = \frac{n(n+1)}{2}$$

Solving Non-Homogeneous Equations Using the Advancement Operator E

Definition 8. Given a sequence of integers $f(n)$, the advancement operator E is defined as follows:

$$\begin{array}{l} f(n) = c \text{ (a constant)} \Rightarrow E(f(n)) = c \\ f(n) \neq \text{constant} \Rightarrow E(f(n)) = f(n+1). \end{array}$$

Example 41.

- For $f(n) = 2$, $E(f(n)) = 2$
- For $f(n) = 2^n$, $E(f(n)) = 2^{n+1}$

Other operators can also be created by combining the operator E with itself or with constants. For a constant c, the operator of the same name c is defined as:

$$c(f(n)) = c \times f(n)$$

Multiplication and addition of operators are defined as:

$$\begin{array}{l} (E1 \times E2)f(n) = E1(E2(f(n))) \\ (E1 + E2)f(n) = E1(f(n)) + E2(f(n)) \end{array}$$

Example 42. Let's illustrate the application of these operators on the following functions:

- $(E - 2)2^n = E(2^n) - 2 \cdot 2^n = 0$
- $E(n + 1) = n + 2$

Thus defined, it's easy to verify:

- Addition and multiplication of operators are commutative:

$$\begin{array}{l} (E1 + E2)f(n) = (E2 + E1)f(n) \\ (E1 \times E2)f(n) = (E2 \times E1)f(n) \end{array}$$

- Addition and multiplication of operators are associative:

$$\begin{array}{l} ((E1 + E2) + E3)f(n) = (E1 + (E2 + E3))f(n) \\ (E1(E2 \times E3))f(n) = ((E1 \times E2)E3)f(n) \end{array}$$

The interest of the operator E lies in its ability to transform a non-homogeneous equation into an equivalent but homogeneous equation after a number of transformations. Let's see this on the following examples.

Example 43. Let's solve the following equation:

$$\begin{aligned} &t(n+2) - 4t(n+1) + 4t(n) = n^2; \forall n \geq 2 \\ &t(0) = 0; t(1) = 1. \end{aligned} \tag{2.13}$$

Apply the operator E to the term n^2 as follows:

$$\begin{aligned} &E(n^2) = (n+1)^2 = n^2 + 2n + 1 \\ &(E-1)(n^2) = E(n^2) - n^2 = 2n + 1 \\ &(E-1)(2n+1) = E(2n+1) - 2n - 1 = 2 \\ &(E-1)(2) = E(2) - 2 = 0 \end{aligned}$$

Therefore, applying $(E-1)^3$ to both sides of Equation 2.13, we obtain:

$$(E-1)^3(t(n+2) - 4t(n+1) + 4t(n) = n^2) \tag{2.14}$$

Expanding this relation gives:

$$t(n+5) - 7t(n+4) + 16t(n+3) - 16t(n+2) + 7t(n+1) - 4t(n) = 0 \tag{2.15}$$

The characteristic equation of this equation is:

$$r^5 - 7r^4 + 16r^3 - 16r^2 + 7r - 4 = 0$$

which can also be written as:

$$(r-1)^3(r-2)^2 = 0$$

The final solution is therefore:

$$t(n) = (a_0 + a_1 n + a_2 n^2) \times 1^n + (a_3 + a_4 n) \times 2^n$$

The constants a_0, a_1, a_2, a_3, and a_4 are determined by the following initial conditions:

$$\begin{aligned} &t(0) = a_0 + a_1 = 0 \\ &t(1) = a_0 + a_1 + a_2 + 2a_3 + 2a_4 = 1 \end{aligned}$$

It turns out three initial values are missing to determine the values of the four constants. To address this issue, calculate $t(2)$ and $t(3)$ from Equation 2.13, i.e.:

$$\begin{aligned} &t(2) = 4t(1) = 4 \\ &t(3) = 9 \\ &t(4) = 37. \end{aligned}$$

Therefore, the missing equations are:

$$\begin{aligned}
t(2) &= a_0 + 2a_1 + 4a_2 + 4a_3 + 8a_4 = 4 \\
t(3) &= a_0 + 3a_1 + 9a_2 + 8a_3 + 24a_4 = 9 \\
t(4) &= a_0 + 4a_1 + 16a_2 + 16a_3 + 64a_4 = 37.
\end{aligned}$$

By solving this system of equations, we obtain the following values:

$$a_0 = 31; \quad a_1 = \frac{31}{2}; \quad a_2 = \frac{5}{2}; \quad a_3 = -31; \quad a_4 = \frac{11}{2}$$

Therefore, the final solution is:

$$t(n) = 11n2^{n-1} - 312n - 1 + \frac{5}{2}n^2 - \frac{15}{2}n + 31$$

Example 44. Let's solve the following equation:

$$\begin{aligned}
&t(n) = t(n-1) + 2^n \\
&t(0) = 1.
\end{aligned}$$

Applying the operator E, we get:

$$(E-2)2^n = 2^{n+1} - 2$$

Therefore:

$$(E-2)(t(n) - t(n-1)) = 0$$

Expanding this equation, we find that the roots of its characteristic equation are:

$$r_1 = 1 \text{ and } r_2 = 2$$

Therefore, the general solution is:

$$t(n) = a_0 + a_1 2^n$$

By proceeding similarly as before, we obtain:

$$a_0 = -1 \text{ and } a_1 = 2$$

The final solution is therefore:

$$t(n) = 2^{n+1} - 1$$

Note:

There is an elegant method to solve this equation. Indeed, let's write down this equation for different values of n as follows:

$$\begin{aligned}
&t(n) = t(n-1) + 2^n\\
&t(n-1) = t(n-2) + 2^{n-1}\\
&t(n-2) = t(n-3) + 2^{n-2}\\
&\dots\\
&t(2) = t(1) + 2^2\\
&t(1) = t(0) + 2
\end{aligned}$$

By summing the terms on the left and the terms on the right, we arrive at:

$$t(n) = t(0) + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

Example 45. Solve the following equation:

$$\begin{aligned}
&t(n) = t(n-1) + n2^n\\
&t(0) = 0
\end{aligned}$$

Applying the operator E, we obtain:

$$\begin{aligned}
&(E-2)n2^n = (n+1)2^{n+1} - n2^{n+1} - n2^{n+1} = 2^{n+1}\\
&(E-2)2^{n+1} = 2^{n+2} - 2 \cdot 2^{n+1} = 0
\end{aligned}$$

Therefore:

$$(E-2)^2(t(n) - t(n-1)) = 0$$

The characteristic equation of this relation is:

$$(r-2)^2(r-1) = 0$$

The general solution is:

$$t(n) = (a_0 + a_1 n)2^n + a_2$$

Knowing that $t(1) = 2$ and $t(2) = 10$, the values of a_0 and a_1 are:

$$a_0 = -2; a_1 = 2, a_2 = 2$$

Therefore, the final solution is:

$$t(n) = (n-1)2^{n+1} + 2$$

Note

The table below summarizes the expressions to use for eliminating some functions $g(n)$ in non-homogeneous equations. In the table, $P_k(n)$ represents a polynomial in n of degree k and α is an integer value:

Function $g(n)$	Corresponding Annihilator
$g(n) = \text{constant}$	$(E-1)$
$g(n) = P_k(n)$	$(E-1)^{k+1}$
$g(n) = \alpha^n$	$(E-\alpha)$
$g(n) = \alpha^n P_k(n)$	$(E-\alpha)^{k+1}$

Important Remarks:

The following two observations can be used to simplify the resolution of recurrent equations:

- If $E1$ is the annihilator of $g(n)$, then the roots of the characteristic equation of
$$E1(c_k t(n+k) + \ldots + c_0 t(n)) = 0 \tag{2.16}$$
are the values that nullify $E1$ and $c_k r^k + c_{k-1} r^{k-1} + \ldots + c_0$. This allows us to avoid expanding equation 2.16, as done previously, and thus avoids cumbersome calculations.

  Example 46. Let's reconsider the equation:
$$t(n+2) - 4t(n+1) + 4t(n) = n$$
  We obtained the equivalent equation:
$$(E-1)^3(t(n+2) - 4t(n+1) + 4t(n)) = 0$$
  Instead of expanding this equation, we simply state that its roots are:

  $r_1 = 1$ (triple root of $(E-1)^3 = 0$)
  $r_2 = 2$ (double root of the characteristic equation: $r^2 - 4r + 4 = 0$)

- If $E1$ is the annihilator of $f(n)$ and $E2$ is the annihilator of $g(n)$, then $(E1 \times E2)$ is the annihilator of the function $f(n) + g(n)$.

  Example 47. Let $k(n) = n3^n + n^2$. From the table above, we deduce:
$$(E-3)^2(n3^n) = 0$$
$$(E-1)^3(n^2) = 0$$
  Therefore, the annihilator of $k(n) = n3^n + n^2$ is $(E-3)^2(E-1)^3$.

2.6.3 Non-linear Equations

Except for homogeneous linear equations with constant coefficients, there are no systematic methods to solve other types of recurrence equations. Ideally, one transforms the equation into an equivalent form for which a solution is known, using various techniques.

In this section, we will review some approaches for solving certain types of nonlinear recurrence equations, particularly those common in algorithm analysis.

Let's begin with an important class of equations, namely divide-and-conquer equations. The general form of this class is as follows:

$$\begin{aligned} &t(n) = at(n/k) + g(n) \\ &t(n_0) = c. \end{aligned} \tag{2.17}$$

where k is a constant and n is a power of k (i.e., $n = k^m$).

Transformation Method

The idea of this approach is to convert the equation into a linear one by defining:

$$Y(m) = t(n) = t(k^m)$$

In this case, we have:

$$t(n/k) = t(k^{m-1}) = Y(m-1)$$

and

$$g(n) = g(k^m) = f(m)$$

Therefore, the original equation transforms into the following linear equation:

$$Y(m) = aY(m-1) + f(m)$$

If we can find an annihilator for $f(m)$, then this equation is solvable.

Example 48. Let's solve the following equation:

$$\begin{aligned} &t(n) = 4t(n/2) + n \\ &t(1) = 1. \end{aligned}$$

After transformations as above, we arrive at the following equation:

$$Y(m) = 4Y(m-1) + 2^m$$

Solving this linear equation (using the theorem from the previous section), we find:

$$Y(m) = a \cdot 2^m + b \cdot 4^m$$

Since $n = 2^m$, we obtain:

$$t(n) = an + bn^2$$

The constants a and b are determined by initial conditions, yielding:

$$a = -1; b = 2$$

Thus, the final solution is:

$$t(n) = 2n^2 - n$$

Substitution Method

The method described above is valid only if we know an annihilator for the function $f(n)$. Unfortunately, this is not always the case. In such situations, another method involves directly developing the recurrence, which may lead to results.

Example 49. Consider solving the following relation:

$$\begin{aligned}&t(n) = t(n/2) + \log\log n\\&t(1) = 1.\end{aligned}$$

Expanding this relation, we get:

$$\begin{aligned}&t(n) = t(n/2) + \log\log n\\&t(n) = t(n/2^2) + \log\log n/2 + \log\log n\\&t(n) = t(n/2^3) + \log\log n/2^2 + \log\log n/2 + \log\log n\\&\ldots\\&t(n) = t(n/2^k) + \textstyle\sum_{i=1}^{k-1} \log\log n/2^i\end{aligned}$$

This expansion stops when $n/2^k = 1$, as the recurrence starts from index 1. That is, when $n = 2^k$. Thus, we obtain:

$$\begin{aligned}\textstyle\sum_{i=1}^{k-1} \log\log n/2^i &= \textstyle\sum_{i=1}^{k-1} \log(k - i)\\&= k\log k - 2(k - 1) + 2\end{aligned}$$

Replacing k with $\log n$, we get:

$$t(n) = 3 + \log n \log\log n - 2(\log n - 1)$$

Variable Transformation Method

This method involves making a judicious change of variables to simplify the given equation.

Example 50. Let's solve the following equation:

$$\begin{aligned} &t(n) = \tfrac{n}{2}t^2(\tfrac{n}{2}) \\ &t(1) = 1. \end{aligned}$$

Assuming $n = 2^k$, we get:

$$t(2^k) = 2^{k-1}t^2(2^{k-1})$$

Setting $Y(k) = t(2^k)$, we obtain:

$$Y(k) = 2^{k-1}Y^2(k-1)$$

The initial condition is $Y(0) = 1$. Now, let's make another change of variables:

$$L(k) = \log Y(k)$$

This leads to the equation:

$$\begin{aligned} &L(k) = 2L(k-1) + (k-1) \\ &L(0) = 0. \end{aligned}$$

The solution to this equation is:

$$L(k) = 2^k - k - 1$$

Thus, we deduce:

$$Y(k) = \frac{2^{2^k-1}}{2^k}$$

Substituting 2^k back with n, we get the final solution:

$$t(n) = \frac{2^{n-1}}{n}$$

Method of Secondary Equations

In the transformation method, we saw that solving the divide-and-conquer equation can be established by transforming it into an equivalent linear equation. The following type of equations can also be solved using a similar approach.

$$\begin{aligned} &t(n) = a(n)t(f(n)) + g(n) \\ &t(n_0) = b. \end{aligned}$$

Example 51. Let's solve the following equation:

$$\begin{aligned} t(n) &= 3t(n/2+1)+n \\ t(3) &= 1. \end{aligned} \tag{2.18}$$

We aim to choose an index k such that the equation above can be written as:

$$Y(k) = 3Y(k-1) + \text{ some function in } k \tag{2.19}$$

Let $n(k)$ be the value of n corresponding to this index k. For equation 2.19 to be equivalent to equation 2.18, we need the following relation to hold:

$$\begin{aligned} n(k-1) &= n(k)/2+1 \\ n(0) &= 3. \end{aligned} \tag{2.20}$$

This secondary equation associated with equation 2.18 can also be written as:

$$\begin{aligned} n(k) &= 2n(k-1)-2 \\ n(0) &= 3. \end{aligned}$$

Solving this equation, we obtain:

$$n(k) = 2^k + 2$$

Thus, equation 2.18 can be written as:

$$\begin{aligned} t(2^k+2) &= 3t(2^{k-1}+2) + 2^k + 2 \\ t(2^0+2) &= 1. \end{aligned}$$

Let $Y(k) = t(n)$, then we have:

$$\begin{aligned} Y(k) &= 3Y(k-1) + 2^k + 2) \\ t(3) &= 1. \end{aligned}$$

The solution to this equation is:

$$Y(k) = 4 \times 3^k - 2^{k+1} - 1$$

As $n = 2^k + 2$, we obtain the final solution:

$$t(n) = 4(n-2)^{\log 3} - 2n + 3.$$

2.6.4 Applications of Recurrence Equations

The study of recurrence equations, in our context, is mainly restricted to the analysis of recursive algorithms and determining their average complexities. Nevertheless, it's useful to overview the various applications of recurrence equations in other domains.

Example 52. Consider the following sequence:

$$1, 5, 13, 25, 41, 61, 85, 113, 145, \ldots$$

From these numbers, we want to find if there's any relationship between them and potentially determine the nth term a_n in terms of n.

One way to approach this is by examining how the differences vary:

$$\begin{array}{ll} a_1 - a_0 = 4; & a_2 - a_1 = 8 \\ a_3 - a_2 = 12; & a_4 - a_3 = 16 \\ a_5 - a_4 = 20; & a_6 - a_5 = 24 \\ a_7 - a_6 = 28; & a_8 - a_7 = 32. \end{array}$$

These calculations suggest the following relation:

$$a_n - a_{n-1} = 4n$$

This relation is simply a linear equation with constant coefficients. Hence, the nth term can be easily computed.

Example 53. Recurrence equations can also be useful in evaluating certain sums. For instance, consider evaluating the sum:

$$S = 12 + 22 + 32 + \ldots + n^2$$

If we define $S(n) = 12 + 22 + 32 + \ldots + n^2$, then it's straightforward to establish the relation:

$$S(n) = S(n-1) + n^2$$

Again, this relation is a linear equation with constant coefficients, making it easy to compute the nth term $S(n)$.

Example 54. Consider the algorithm computing the GCD of two numbers n and m.
Algorithm PGCD(int n,m)
Input: two integers n and m
Output: GCD

```
**Start**
If m=0;
return(n)
else return( PGCD(m,n mod m));
**End**
```

If $t(n, m)$ represents the complexity of the function $PGCD(n, m)$, then the complexity of executing $PGCD(m, n \mod m)$ corresponds to $t(n, n \mod m)$. When $m = 0$, the function $PGCD$ performs two operations (1 test and 1 assignment). Otherwise, in addition to the complexity of $PGCD(n, n \mod m)$, the function $PGCD$ also performs four operations (1 test, 1 assignment, 1 self-call, and 1 modulo operation). Therefore, we get the following recurrence:

$$t(n, m) = \begin{cases} 2 \quad \text{if } m = 0 \\ t(m, n \mod m) + 4 \quad \text{otherwise} \end{cases}$$

This equation represents a nonlinear recurrence relation.

3
Sorting algorithms

3.1 Presentation

According to the dictionary, "to sort" means "to arrange into several classes according to certain criteria." More narrowly, the term "sorting" in computer science is often associated with the process of arranging a set of items in a specified order. For example, sorting N integers in ascending order or N names in alphabetical order. Any set equipped with a total order can provide a sequence of items to be sorted.

Interestingly, intuitively, when given a set to sort, everyone devises different sorting strategies depending on the number of items in the set, such as a deck of 52 cards or 200 students to be sorted alphabetically. Selection sort, bubble sort, insertion sort, quicksort, merge sort... these various methods each have their own characteristics... and their performance level, which corresponds to the algorithm's complexity. The most widely used method today is arguably quicksort, invented by Sir Charles Antony Richard Hoare in 1960 – some say it is the most widely used algorithm in the world!

In the face of a small-scale example, it may seem somewhat trivial to refine the process enough to find an additional sorting algorithm to apply to a small number of elements, as the difference in complexity may not be very noticeable. However, it is important to keep in mind that we may need to sort hundreds of thousands of elements, and while a difference in strategy is visible with small numbers like 52 or 200, it will be even more significant for larger datasets.

We can illustrate this with the general case using the example of sorting integers. This is demonstrated in the animation above. The different methods are initially shown in a Visual Sort, applied to sorting 16 elements. A second level, called Time Sort, allows testing of various algorithms with a large number of elements. Finally, the Log menu keeps track of successive attempts, facilitating comparison between them.

3.2 To learn more about the different sorting methods

To describe more precisely the different sorting methods, their procedures, their complexity, we assume we have an array `tab` of N integers numbered from 1 to N, and we aim to sort the integers in ascending order, "from left to right" if we want to give a representation of the array. The sorting procedures will be written in a "pseudo-French code" that should be clear enough to not require translation of keywords. However, this "pseudo-code" is also close enough to a programming language for a computer scientist to easily translate it into an existing programming language like Java or C.

3.3 Complexity of algorithms

To evaluate the complexity of the various sorting algorithms presented, we will count the number of comparisons and value exchanges between two elements of the array, excluding assignments and comparisons on loop counting variables.

The methods presented fall into two types:

- Methods that sort elements pairwise, more or less efficiently, but always require comparing each of the N elements with each of the other N-1 elements, resulting in a number of comparisons of the order of N^2 — denoted by the Big-O notation $O(N^2)$. For example, for $N = 1000$, $N^2 = 10^6$; for $N = 10^6$, $N^2 = 10^{12}$. Algorithms of this type include:
  - Elementary sorting method, selection sort
  - Its variant, bubble sort or sinking sort
  - A method similar to sorting cards in a game, insertion sort
- Methods that are faster because they sort subsets of these N elements and then merge the sorted elements, illustrating the "divide and conquer" principle. The number of comparisons is of the order $N \log(N)$. For example, for $N = 1000$, $N \log(N) \approx 10000$; for $N = 10^6$, $N \log(N) \approx 20 \times 10^6$. Algorithms of this type include:
  - The famous quicksort algorithm
  - Finally, merge sort

This list is not exhaustive, as there are methods particularly suited to specific types of data. Radix sort is an example of such a method, tailored for certain types of data.

3.4 Sort by selection

It consists of finding in the array the index of the smallest element, i.e., the integer *min* such that tab[k] $\geq$ tab[min] for all k. Once this index is found, elements tab[1] and tab[min] are exchanged – this exchange requires a temporary variable of type integer – then the same procedure is applied to the remaining elements tab[2], ..., tab[N].

Algorithm 6 Selection Sort

```
INTEGER $i, k$
INTEGER $min$
INTEGER $tmp$
for $i \leftarrow 1$ to $N - 1$ do
    {Finding the index of the minimum element}
    $min \leftarrow i$
    for $k \leftarrow i + 1$ to $N$ do
        if tab[$k$] < tab[$min$] then
            $min \leftarrow k$
        end if
    end for
    {Swapping values between the current index and the minimum}
    $tmp \leftarrow$ tab[$i$]
    tab[$i$] $\leftarrow$ tab[$min$]
    tab[$min$] $\leftarrow tmp$
end for
```

It is easy to count the number of operations. Regardless of the initial order of the array, the number of comparisons remains the same, as does the number of exchanges. At each iteration, we consider the element tab[i] and compare it successively to tab[$i+1$], $\ldots$, tab[N]. Therefore, we perform $N - i$ comparisons.

The total number of comparisons is:

$$\sum_{i=1}^{N-1} (N - i) = \frac{N(N-1)}{2}$$

and there are $(N - 1)$ exchanges.

Regarding its complexity, selection sort is said to be $O(N^2)$, both in the best case, average case, and worst case scenarios. This means its execution time is proportional to the square of the number of elements to be sorted.

3.5 Bubble sort

The "bubble sort" is a variant of selection sort. It involves traversing the array tab and swapping any pair of consecutive elements (tab$[k]$, tab$[k+1]$) that are out of order — which constitutes an exchange and therefore requires an integer-type temporary variable. After the first pass, the largest element will be positioned in the last slot of the array, tab$[N]$, and the same procedure is then applied to the array composed of elements tab$[1], \dots,$ tab$[N-1]$. The name of this sort comes from the movement of the largest "bubbles" towards the right.

Algorithm 7 Bubble Sort

```
Input: Array tab of integers
Output: Sorted array tab in ascending order
INTEGER $i, k$
INTEGER $tmp$
for $i \leftarrow 1$ to $N-1$ do
    for $k \leftarrow 1$ to $N-i$ do
        if tab$[k] >$ tab$[k+1]$ then
            tmp $\leftarrow$ tab$[k]$
            tab$[k] \leftarrow$ tab$[k+1]$
            tab$[k+1] \leftarrow$ tmp
        end if
    end for
end for
```

The number of comparisons in the bubble sort procedure is the same as for selection sort:

$$N\sum_{i=2}^{N}(i-1) = \frac{N(N-1)}{2}.$$

The number of exchanges depends on the initial order of elements in the array:

- In the best case scenario, the array is already sorted, so no exchanges are needed.
- On average, it is shown that the number of exchanges is:

$$\frac{N(N-1)}{4}.$$

- In the worst case scenario, the integers in the array are initially given in descending order. In this case, an exchange is performed for each comparison, resulting in:

$$\frac{N(N-1)}{2}.$$

In any case, the complexity of bubble sort remains $O(N^2)$, meaning it is of the same order as the square of the number of elements.

3.6 Insertion sort

This sorting method is very different from selection sort and resembles the method used to sort cards in a game: we take the first card, tab[1], then the second, tab[2], which we place relative to the first, then the third tab[3], which we insert in its place relative to the first two, and so on. The general principle is therefore to consider that the first (i-1) cards, tab[1], ..., tab[$i-1$], are sorted and to place the i-th card, tab[i], in its correct position among the (i-1) already sorted cards, until $i = N$.

To place tab[i], we use a temporary variable tmp to store its value, which we compare successively with each element tab[$i-1$], tab[$i-2$], ... moving them to the right as long as their value is greater than that of tmp. We then assign to the position in the array left vacant by this shift the value of tmp.

Algorithm 8 Insertion Sort

```
1:  tmp, $i \leftarrow 0$
2:  **for** $i \leftarrow 2$ to $N$ **do**
3:      tmp $\leftarrow$ tab[$i$]
4:      $j \leftarrow i - 1$
5:      **while** $j > 0$ and tab[$j$] $>$ tmp **do**
6:          tab[$j+1$] $\leftarrow$ tab[$j$]
7:          $j \leftarrow j - 1$
8:      **end while**
9:      tab[$j+1$] $\leftarrow$ tmp
10: **end for**
```

Comparison with the two previous algorithms shows that the complexity of insertion sort depends more heavily on the initial order of the array. Here we count the number of comparisons (which is one less than the number of shifts):

- In the best case scenario, the initial array is sorted, and we perform one comparison at each insertion, thus making $N-1$ comparisons.
- On average, it is shown that the number of comparisons is:
$$N-1+\frac{N(N-1)}{4}$$
- In the worst case scenario, where the integers in the array are initially in descending order, an exchange is made with each comparison, resulting in $N(N-1)/2$ exchanges.

Unlike selection sort and bubble sort, which require a constant number of comparisons, insertion sort results in very few comparisons when the initial array is nearly sorted. Thus, insertion sort exhibits better properties in such cases.

3.7 Quick sort

This sorting method, arguably the most widely used today — some even say it's the most used algorithm worldwide — was invented by Sir Charles Antony Richard Hoare in 1960. It exemplifies the "divide and conquer" principle, which involves recursively applying a method designed for a given problem size to smaller, similar subproblems. This general principle often leads to algorithms that achieve significant reductions in complexity.

An element is chosen randomly from the array as the pivot, whose value is assigned to a variable, say *pivot*. The array is then partitioned into two zones: elements less than or equal to *pivot* and elements greater than or equal to *pivot*. If the smaller elements are moved to the front of the array and the larger ones to the back, then the pivot can be placed in its final position between these two zones. This process is recursively applied to each partition until each is reduced to a single element.

The choice of pivot remains the most critical part of the sorting process. In the previous algorithm, it is chosen randomly from the elements of the array, but this choice can prove catastrophic: if the pivot is consistently chosen as the smallest element of the array, quicksort degenerates into selection sort.

It is shown that the complexity of this sorting method is:

- In the best case scenario, $O(N\log N)$;
- On average, $O(NlogN)$;
- In the worst case scenario, $O(n^2)$.

There are numerous techniques to minimize the likelihood of quicksort's worst-case scenario, making it the fastest average-case sorting method among those used.

Algorithm 9 Quicksort

```
pivotIndex ← random integer between start and end
tmp ← tab[pivotIndex]
tab[pivotIndex] ← tab[start]
tab[start] ← tmp
k ← start
for i ← start + 1 to end do
    if tab[i] < tab[start] then
        tmp ← tab[i]
        tab[i] ← tab[k+1]
        tab[k+1] ← tmp
        k ← k + 1
    end if
end for
tmp ← tab[start]
tab[start] ← tab[k]
tab[k] ← tmp

return k
```

3.8 Merge sort

This sorting method is another example of a technique that applies the "divide and conquer" principle. Given two sorted sequences of elements, with respective lengths $L1$ and $L2$, it is straightforward to obtain a third sorted sequence of length $L1+L2$ by merging the two previous sequences, as illustrated in the *fusion* procedure.

For the needs of the *mergeSort* procedure, we will give the following form to the *merge* procedure which merges two sequences of elements placed in an array *tab*, respectively between indices *start* and *mid* and between indices *mid + 1* and *end*:

It can be observed that the *merge* procedure requires an auxiliary array as large as the number of elements to be merged. This is the main drawback of merge sort, as its complexity in all cases is $O(N \log N)$, at the cost of an auxiliary array as large as the initial array, which can be limiting in memory-constrained situations.

The recursive procedure for merge sort is then:

The version used in the demonstration animation is actually the iterative version. It involves sorting subarrays of length $2, 4, \ldots$, powers of 2, up to the length of the array.

Algorithm 10 Merge Procedure

```
1: i $\leftarrow$ start
2: j $\leftarrow$ mid + 1
3: **for** k $\leftarrow$ start **to** end **do**
4:     **if** (j > end) **or** (i $\leq$ mid **and** tab[i] < tab[j]) **then**
5:         tmp[k] $\leftarrow$ tab[i]
6:         i $\leftarrow$ i + 1
7:     **else**
8:         tmp[k] $\leftarrow$ tab[j]
9:         j $\leftarrow$ j + 1
10:    **end if**
11: **end for**
12: **for** k $\leftarrow$ start **to** end **do**
13:     tab[k] $\leftarrow$ tmp[k]
14: **end for**
```

Algorithm 11 Recursive Merge Sort Procedure

```
1: **if** start < end **then**
2:     mid $\leftarrow$ (start + end)/2
3:     MergeSortRecursivetab, tmp, start, mid
4:     MergeSortRecursivetab, tmp, mid + 1, end
5:     Mergetab, tmp, start, mid, end
6: **end if**
7: tmp $\leftarrow$ new array of size $N$
8: MergeSortRecursivetab, tmp, 1, $N$
```

Algorithm 12 Iterative Merge Sort Procedure

```
 1: tmp $\leftarrow$ new array of size $N$
 2: i $\leftarrow 1$
 3: start $\leftarrow 1$
 4: end $\leftarrow$ start + i + i $- 1$
 5: while i $< N$ do
 6:    start $\leftarrow 1$
 7:    while start + i $- 1 < N$ do
 8:       end $\leftarrow$ start + i + i $- 1$
 9:       if end $> N$ then
10:          end $\leftarrow N$
11:       end if
12:       Mergetab, tmp, start, start + i $-$ 1, end
13:       start $\leftarrow$ start + i + i
14:    end while
15:    i $\leftarrow$ i + i
16: end while
```

3.9 Strategies for Designing Sequential Algorithms

3.9.1 Divide and Conquer

Divide and conquer (e.g., fast exponentiation, quicksort, Strassen's algorithm for matrix multiplication, etc.)

3.9.2 Overview of the Method

The divide and conquer method is an approach that can sometimes lead to efficient solutions for algorithmic problems. The idea is to break down the initial problem of size n into several sub-problems of smaller sizes and then recombine the partial solutions.

Example 55. Merge Sort Algorithm
To sort an array of size n, we split it into two arrays of size $n/2$, and the merge step allows us to recombine the two solutions in $n-1$ operations. We can describe it as follows:

We estimate the complexity by counting the number $T(n)$ of comparisons

Algorithm 13 Merge Sort

```
Require: $n$: integer; $T[*]$: Array
 1: Procedure MergeSort(T):
 2: if $n \leq 1$ then
 3:     Return (T)
 4: else
 5:     $n = |T|$;
 6:     $T_1$ =MergeSort($T[0 \dots n/2]$);
 7:     $T_2$ =MergeSort($T[n/2+1 \dots n]$);
 8:     Return Merge($T_1$,$T_2$);
 9: end if
10: End Procedure.
```

performed by the algorithm. We find that:

$$\begin{cases} T(0) = 0 \\ T(1) = 0 \\ T(n) \approx 2T(\frac{n}{2}) + n - 1. \end{cases}$$

The symbol $\approx$ is used since there are integer parts to consider for rigor.

3.9.3 General Form

The general form considered is:

- Divide: the problem is split into a sub-problems of sizes $\frac{n}{b}$, with $a \geq 1$ and $b > 1$.
- Conquer: the sub-problems are solved recursively.
- Combine: the solutions to the sub-problems are used to construct the solution to the original problem in $O(n^d)$ time, with $d \geq 0$.

The equation to solve for complexity is:

$$\begin{cases} T(1) = \text{constant} \\ T(n) \approx aT(\frac{n}{b}) + O(n^d). \end{cases}$$

theorem 3. Consider the equation $T(n) = aT(\frac{n}{b}) + O(n^d)$. Let $\lambda = \log_b a$. We have three cases:

1. if $\lambda > d$, then $T(n) = O(n^\lambda)$;

2. if $\lambda = d$, then $T(n) = O(n^d \log n)$;
3. if $\lambda < d$, then $T(n) = O(n^d)$.

Example 56. For merge sort, we have $a = 2$, $b = 2$, $\lambda = d = 1$, yielding a complexity of $O(n \log n)$.

In practice, only cases 1 and 2 may lead to interesting algorithmic solutions. In case 3, the entire cost is concentrated in the combine phase, often indicating more efficient solutions exist.

3.9.4 Examples

Example 57. Binary Search
Given a sorted array T of size n, we are interested in an algorithm that searches if $x \in T$ using binary search. In the recursive version, we specify a start index d and end index f to search for x in T between positions d and f. The initial call uses $d = 0$ and $f = n - 1$.

$$t(n) = \begin{cases} 1 \text{ if } n \leq 1 \\ t(\frac{n}{2}) + 1 \text{ if } n > 1 \end{cases}$$

Binary search complexity is thus $O(\log n)$.

Example 58. Fast Exponentiation
We want to calculate x^n for given x and n, optimizing the complexity relative to n. The naive method (multiplying x by itself n times) gives linear complexity, but we can improve it by using:

$$x^n = \begin{cases} x \text{ if } n = 1 \\ (x^2)^{\frac{n}{2}} \text{ if } n \text{ is even and positive} \\ x(x^2)^{\frac{n-1}{2}} \text{ if } n > 2 \text{ is odd.} \end{cases}$$

The complexity of fast exponentiation is thus $O(\log n)$.

Further examples and dynamic programming can be expanded similarly.

Algorithm 14 Binary Search

```
Require: $x, d, f, m$: integer;
    $T[*]$: Array;
 1: Procedure Search($T, x, d, f$):
 2: if $f < d$ then
 3:     Return false
 4: else
 5:     $m = \frac{d+f}{2}$;
 6:     if $T[m] = x$ then
 7:         Return True
 8:     else
 9:         if $T[m] < x$ then
10:             Return Search($T, x, m+1, f$)
11:         else
12:             Return Search($T, x, d, m-1$)
13:         end if
14:     end if
15: end if
16: End Procedure.
```

4
Trees

4.1 Definitions and Theorem

Trees are special graphs, widely popular in algorithms and computer science.

A forest is a graph without cycles, where each connected component is acyclic and thus connected by definition. The definition of a forest aligns well with the usual sense of a collection of trees, where each connected component is a tree.

4.1.1 Number of edges in a graph:

Let $G = (X; U)$ be a graph, n the number of nodes $n = |X|$, and m the number of edges $m = |U|$.

- If G is connected, $m \geq n - 1$.
- If G is acyclic, $m \leq n - 1$.

4.1.2 Tree

A tree is a connected graph without cycles. Therefore, it has $n-1$ edges, $m = n-1$. Hence, a tree is a graph that connects all nodes with a minimum number of edges.

REMARKS

- Adding even a single edge to a tree creates a cycle.
- A connected graph has a spanning tree as a subgraph.

Example

A related graph:

A tree extracted from the graph:

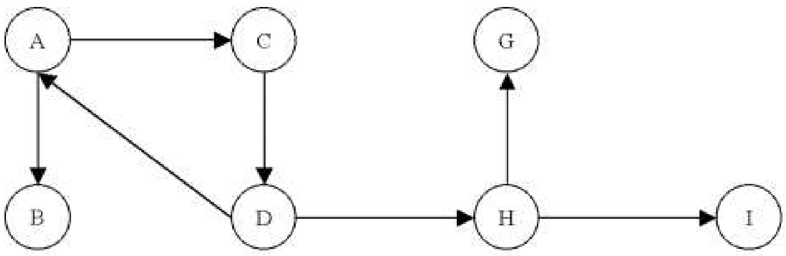


Figure 4.1: Convexe graph

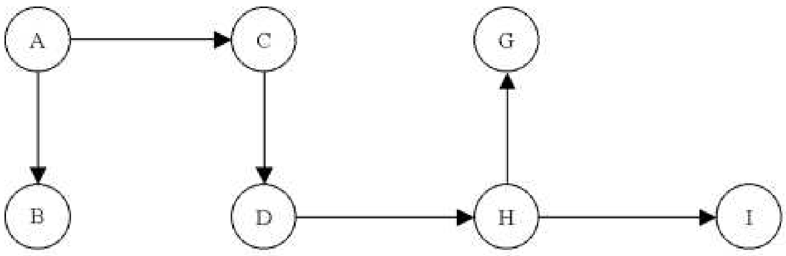


Figure 4.2: Tree

4.2 Root, anti-root

Often, to manipulate a tree, we specify a particular vertex of the graph which we call the root. In the case of undirected graphs, the choice of a root r in the tree is arbitrary. In the case of directed graphs, the root is uniquely defined as the vertex without a predecessor in the tree.

The choice of a root is, in a certain sense, equivalent to orienting the tree, with the root appearing as the common ancestor in the manner of a genealogical tree. The vocabulary of graph theory directly draws inspiration from this: we speak of children, parents, siblings, etc.

- A node a of a graph G is a root of G if there exists a path connecting a to every node of the graph G.
- A node a of a graph G is an anti-root of G if there exists a path connecting every node of the graph G to a.

Example

- A is a root of the graph.

- I is an anti-root of the graph.

4.3 Tree, anti-tree

- A graph G is a rooted tree with root a if G is a tree and a is a root.
- A graph G is an anti-rooted tree with anti-root a if G is a tree and a is an anti-root.

A rooted tree is a tree with a distinguished vertex, called the root. A rooted tree is usually depicted with the root at the top of the drawing and the leaves at the bottom.

In a rooted tree, we can assign a rank to the vertices. The rank is the distance from the vertex to the root.

We say that the height of the rooted tree is the maximum rank (4 in the example opposite).

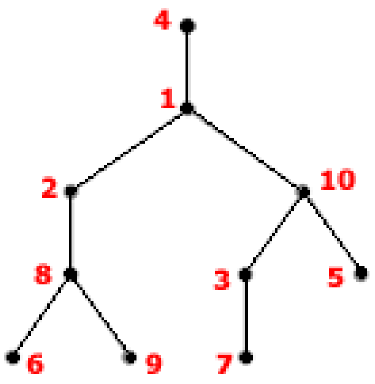


Figure 4.3: Tree, anti-tree

- The root is vertex 4. Vertices 5, 6, 7 and 9 are the leaves.
- Vertex 4 (the root) has rank 0, vertex 1 has rank 1, vertices 2 and 10 have rank 2, vertices 3, 5 and 8 have rank 3 and vertices 6, 7 and 9 have rank 4.
- The height of the tree is 4

4.4 Tree covering

4.4.1 Definition:

Let $G = (X, U)$ be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G: A spanning tree for a graph $G = (X, U)$ is a tree constructed solely from the edges of U and that connects ("spans") all the vertices of X. Therefore, a spanning tree of a graph G is a graph T such that:

- The graph T is a tree.
- The graph T is a subgraph of G.

4.4.2 Algorithm for constructing a spanning tree:

We choose an arbitrary vertex of the graph, then build a simple path from this vertex by adding edges of G as long as possible. If the path thus constructed contains all the vertices of the graph, the path is a spanning tree. Otherwise, we return to the second-to-last vertex of the path and from there, if possible, construct a new simple path as long as possible that does not contain any vertex of the first path constructed. If this is not possible, we go back to the third-to-last vertex and start again. If the graph is connected, this process can be repeated until all vertices are exhausted to obtain a spanning tree.

Example:

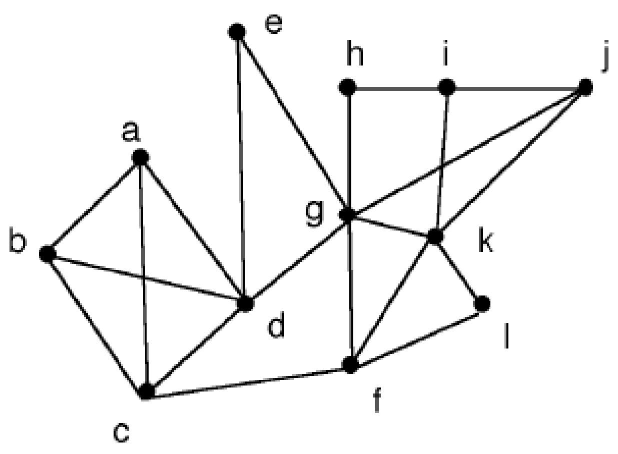


Figure 4.4: Graph G

- Let's start, for example, from vertex a
- 1st step: we construct the simple path $a, b, c, d, g, h, i, j, k, l, f$
- 2nd step: we go back to vertex g to form the path g, e
- A spanning tree is then:

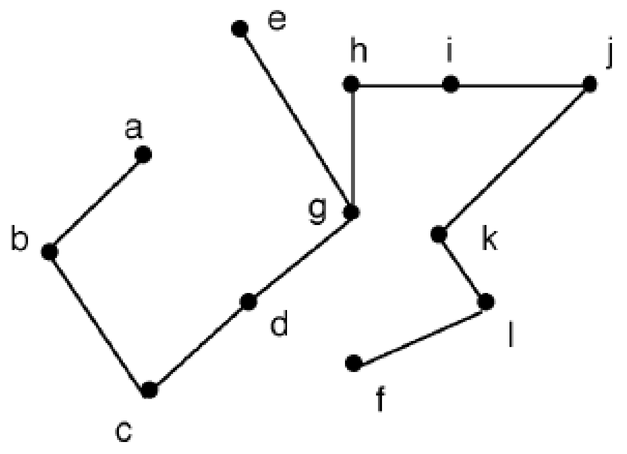


Figure 4.5: spanning tree of graph G

4.5 Minimum weight spanning tree

Let G be a weighted graph. The minimum spanning tree problem consists in finding a spanning tree whose sum of the weights $c(e)$ of the edges is minimum.

Example:

Minimizing the cost of installing power lines between houses, that is, we want to connect all the houses without having unnecessary lines, hence the search for a spanning tree. Then, we associate the cost with the length of the cables, so we want to minimize the total length of the cables used.

There are 2 famous algorithms to solve the minimum spanning tree (MST) problem. Each of these 2 algorithms particularly uses one of the characterizations of trees to find a minimum spanning tree: either by considering trees as connected graphs with the minimum number of edges or by considering trees as acyclic graphs with the maximum number of edges. These 2 algorithms also use 2 very different solving techniques:

- Prim's algorithm: It maintains a connected subgraph that grows step by step during the construction.
- Kruskal's algorithm: It maintains an acyclic partial graph during the construction. While search algorithms are specific to graphs, this one uses a more general solving paradigm: greedy algorithms.

KRUSKAL'S Algorithm:

Principle:

The principle of Kruskal's algorithm to find a minimum spanning tree in a graph G is first to sort the edges in increasing order of their weight. Then, in this order, the edges are added one by one to a graph G' to gradually build the tree. An edge is added only if its addition to G' does not introduce a cycle, in other words, if G' remains a tree. Otherwise, we move to the next edge in the sorted order.

Algorithm 15 Kruskal

```
Require: $G = (X, U)$ a graph.
Ensure: $U'$ a set of edges.
 1: Intermediate variables: $i$ an integer.
 2: Begin
 3: Sort the edges of $G$ in increasing order of weights; {W}e denote them
    $u_1, u_2, \ldots, u_m$.
 4: $U' \leftarrow \emptyset$;
 5: for $i \leftarrow 1$ to $m$ do
 6:     if the graph $(X, U' \cup \{u_i\})$ does not contain a cycle then
 7:         $U' \leftarrow U' \cup \{u_i\}$;
 8:     end if
 9: end for
10: End
```

Example

- $U' \leftarrow \emptyset$;
- $i \leftarrow 1$; $U' \leftarrow \{u_1\}$;
- $i \leftarrow 2$; $U' \leftarrow \{u_1, u_2\}$;
- $i \leftarrow 3$; $U' \leftarrow \{u_1, u_2, u_3\}$;

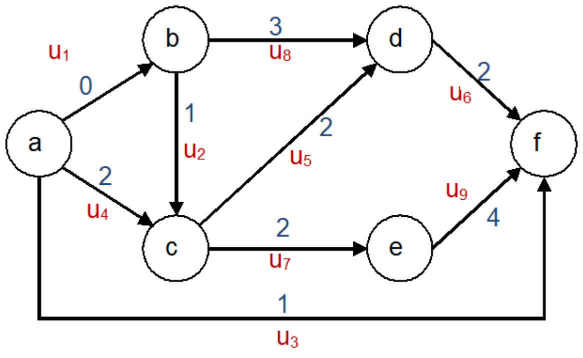


Figure 4.6: Example

- $i \leftarrow 4$; $U' \leftarrow \{u_1, u_2, u_3\}$;
- $i \leftarrow 5$; $U' \leftarrow \{u_1, u_2, u_3, u_5\}$;
- $i \leftarrow 6$; $U' \leftarrow \{u_1, u_2, u_3, u_5\}$;
- $i \leftarrow 7$; $U' \leftarrow \{u_1, u_2, u_3, u_5, u_7\}$;
- $i \leftarrow 8$; $U' \leftarrow \{u_1, u_2, u_3, u_5, u_7\}$;
- $i \leftarrow 9$; $U' \leftarrow \{u_1, u_2, u_3, u_5, u_7\}$;

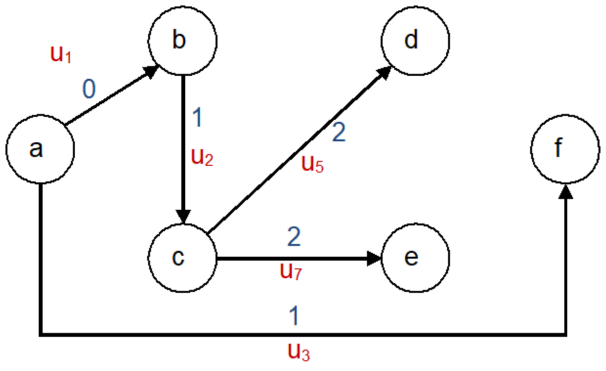


Figure 4.7: Kruslkal

4.6 PRIM algorithm

4.6.1 PRINCIPLE:

Prim's algorithm is based on the CHARACTERIZATION of trees as minimally connected graphs in the sense of INCLUSION: one cannot remove an edge from a tree without disconnecting it. The idea of the algorithm is to MAINTAIN a connected partial subgraph by CONNECTING a new vertex at each STEP. Prim's algorithm will thus GROW a tree until it SPANS all the vertices of the graph. If at some STEP a SET U of vertices are connected to each other, to CHOOSE the next vertex to connect, the algorithm starts from a simple OBSERVATION: in a spanning tree, there must necessarily be an edge that connects one of the vertices of U with a vertex OUTSIDE of U. To CONSTRUCT a minimum spanning tree (MST), it is sufficient to CHOOSE among these OUTGOING edges the one with the LOWEST WEIGHT. To DETECT the OUTGOING edges, we can MARK the vertices already connected as the algorithm PROGRESSES. An OUTGOING edge then NECESSARILY connects a MARKED vertex and an UNMARKED vertex. Prim's algorithm thus APPEARS as an ADAPTATION of the search algorithm for the MST problem. One QUESTION remains: which vertex to START from? Well, the CHOICE of the INITIAL vertex does not MATTER... every vertex must EVENTUALLY be connected to the others in the final tree.

4.6.2 PSUDO-ALGORITHM

Algorithm 16 Prim's Algorithm 1

```
1: INITIALIZE $T \leftarrow \emptyset$
2: CHOOSE an ARBITRARY vertex $v$ from $X$
3: MARK $v$
4: while there are UNMARKED vertices do
5:     FIND the EDGE with the SMALLEST WEIGHT that CONNECTS a
       MARKED vertex to an UNMARKED vertex
6:     ADD this EDGE to $T$
7:     MARK the NEWLY CONNECTED vertex
8: end while
```

Algorithm 17 Prim's Algorithm 2

```
Require: $G = (X, U)$ a connected graph with positive edge weights
Ensure: $T$ a minimum weight spanning tree
1: Initialize $F$ to empty set
2: Mark an arbitrary vertex
3: while there exists an unmarked vertex adjacent to a marked vertex do
4:     Select an unmarked vertex $y$ adjacent to a marked vertex $x$ such that $(x, y)$
       is the outgoing edge of the lowest weight
5:     $F := F \cup \{(x, y)\}$
6:     Mark $y$
7: end while
8: Return $T = (X, F)$
```

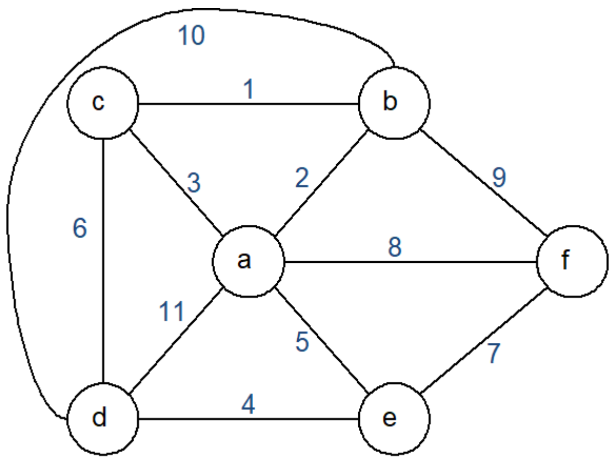


Figure 4.8: Exmaple

Example

Application Example

Use the algorithm to design a minimum cost communication network connecting all the computers represented by the following graph:

Possible solution (the minimum cost is equal to 4700 euros)

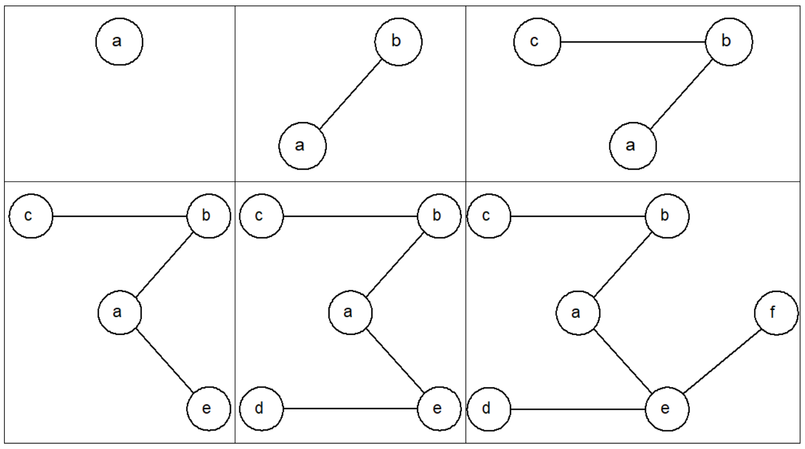


Figure 4.9: Prim's Algorithm

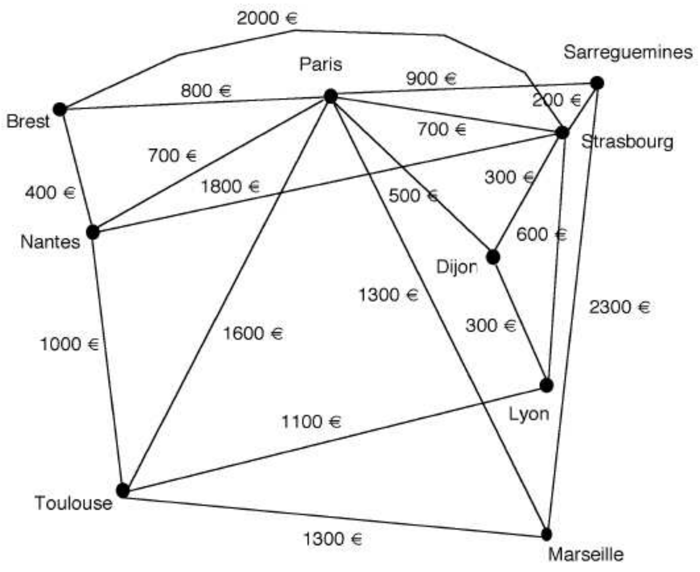


Figure 4.10: Exmaple

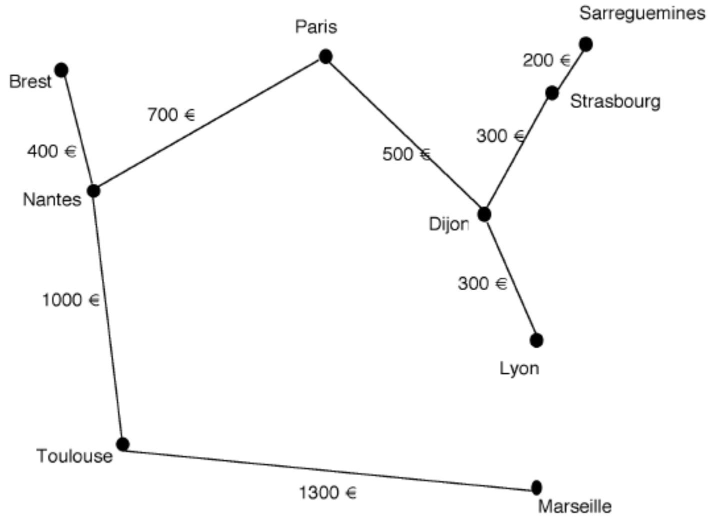


Figure 4.11: Sol

5 Graph

5.1 Introduction to graphs

For more than a century, the rise of graph theory was largely inspired and guided by the Four Color Conjecture. The resolution of this conjecture by K. Appel and W. Haken in 1976, the year of the publication of our first book *Graph Theory with Applications*, marked a turning point in its history. Since then, the subject has experienced exponential growth, largely due to its role as an essential structure in modern applied mathematics. Computer science and combinatorial optimization, in particular, rely on graph theory and contribute to its development. Furthermore, in a world where communications are of paramount importance, the great adaptability of graphs makes them indispensable for the design and analysis of communication networks.

Application Areas of Graph Theory Graphs (and consequently graph theory) are used in numerous fields. Here are a few examples:

- **Communication networks**: road networks represented by a road map, railway networks, telephone networks, television relay networks, electrical networks, information networks within an organization, etc.

- **Production management**: activity-on-vertex graphs, better known as PERT graphs ("Program Evaluation and Research Task" or "Program Evaluation Review Technique").

- **The study of electrical circuits**: Kirchhoff, who studied electrical networks, can be considered one of the precursors of this theory.

- **Chemistry, sociology, and economics**: the notion of a clique is an example of the involvement of graph theory in these disciplines.

5.2 Definitions

5.2.1 Graph

A graph is a diagram consisting of a finite set of points and a set of arrows connecting each pair of these points. The points are called the vertices of the graph, and the arrows are called the edges of the graph.

5.2.2 Mathematical Definition of a Graph

A graph is represented by a pair of two sets $G = (X, U)$.

- Where X: the set of nodes (or vertices)
- U: the set of edges (undirected graph) or arcs (directed graph).

Formally:
A graph $G = (X, U)$ is the pair consisting of:

- A set $X = \{x_1, x_2, x_3, \ldots, x_n\}$
- A set $U = \{u_1, u_2, u_3, \ldots, u_m\}$ of elements of the Cartesian product

$$X \times X = (x, y) \mid x, y \in X$$

5.2.3 Node (Vertex)

A node is a basic element of a graph; it can be an object (a city, a station, a number, etc.), a concept, knowledge, or an idea.

5.2.4 Arc, Edge

An arc connects two nodes, represented by a pair (x, y) where x and y are nodes. An arc can be directed, meaning the order of x and y is important in the pair (x, y), so $(x, y) \neq (y, x)$.

Directed Graph

or **Digraph**
An arc can be undirected (edge), and in this case, the order of x and y in the pair (x, y) does not matter, so $(x, y) = (y, x)$.

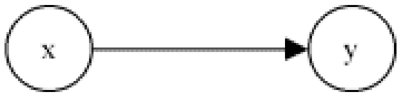


Figure 5.1: Oreinted graph

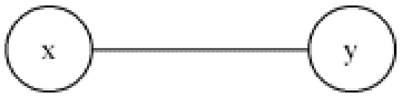


Figure 5.2: Undirected graph

Undirected Graph

Note:

An undirected arc can always be transformed into a situation where only directed arcs are present.

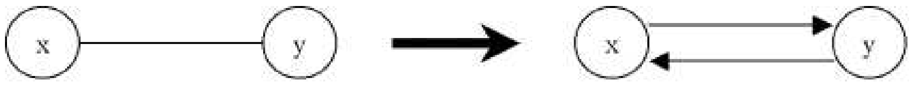


Figure 5.3: correspendance graph

Example

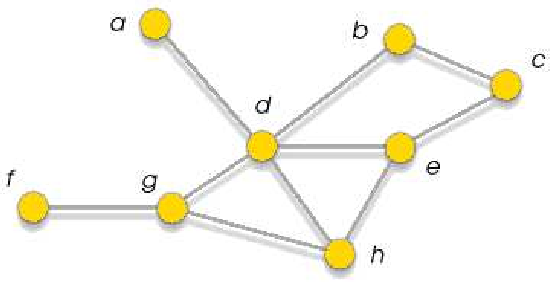


Figure 5.4: Example graph

- $G = (X, U)$
- $X = \{a, b, c, d, e, f, g, h\}$

- $U = \{(a, d), (b, c), (b, d), (d, e), (e, c), (e, h), (h, d), (f, g), (d, g), (g, h)\}$

5.3 Order of a Graph

5.3.1 Order

The order of a graph is the number of its vertices. Order (G) = $|X|$.

5.3.2 Loop

A loop is an arc whose initial endpoint is the same as its final endpoint. For example, $(x; x)$ is a loop.

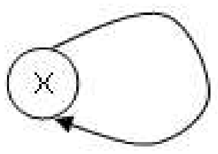


Figure 5.5: Loop

5.3.3 Adjacency

Two vertices x and y are adjacent if there exists an arc (x, y) in U. The vertices x and y are then said to be neighbors. For an arc $u = (x; y)$ we say that:

- x is adjacent to y and y is adjacent to x
- x and y are adjacent to u and u is adjacent to x and y

Example

- Al: Algeria
- Ma: Morocco
- Tu: Tunisia
- Lb: Libya
- Mr: Mauritania

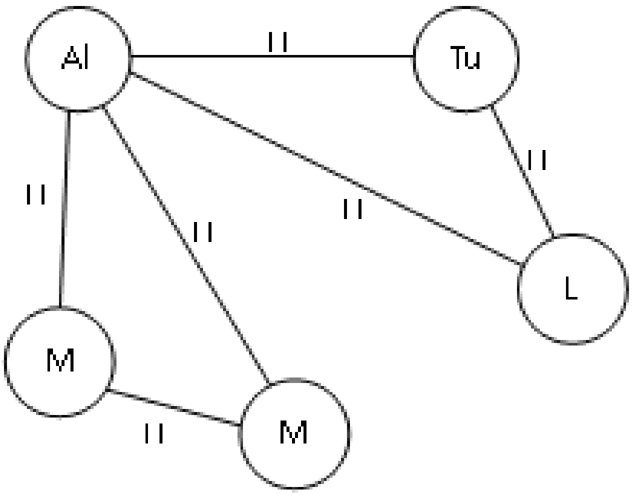


Figure 5.6: Adjacency

5.3.4 Incidence

An edge is incident to a vertex x if x is one of its endpoints.

Example

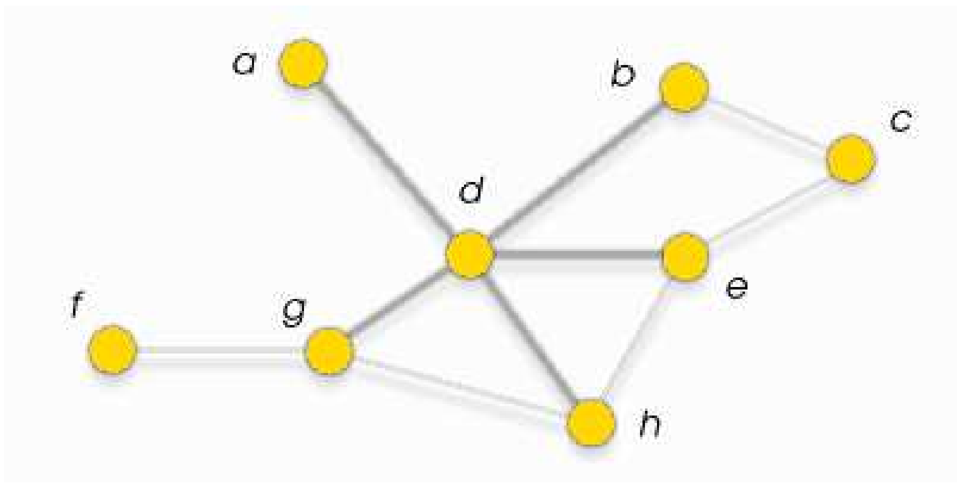


Figure 5.7: Incidence

The edges incident to d are:

- (d, a)
- (d, b)

- (d, e)
- (d, h)
- (d, g)

5.3.5 Degree

Out-degree

It is the number of adjacent arcs that start from it. It is denoted as $d^+(x)$: the set of arcs in G outgoing from node x, $d^+(x) = |\{u \in U \mid u = (x; y) \text{ where } y \in X\}|$: y is a successor of x.

In-degree

It is the number of adjacent arcs that arrive at it. It is denoted as $d^-(x)$: the set of arcs in G incoming to node x, $d^-(x) = |\{u \in U \mid u = (y; x) \text{ where } y \in X\}|$: y is a predecessor of x.

5.3.6 Degree of a Node x

It is the number of adjacent arcs to x. It is denoted as $d(x)$. $d(x) = d^+(x) + d^-(x)$. $d(x) = |\{u \in U \mid u = (x; y) \text{ or } u = (y; x) \text{ where } y \in X\}|$.

Example

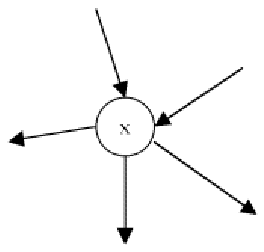


Figure 5.8: Degrees

- Successors: $d^+(x) = 3$
- Predecessors: $d^-(x) = 2$
- Degree: $d(x) = 5$

Properties

1. **Handshake Lemma:** The sum of the degrees of the vertices in a graph is equal to 2 times the number of its edges. An edge $e = (x, y)$ in the graph is counted exactly 2 times in the sum of degrees: once in $d(x)$ and once in $d(y)$.

2. For a simple graph of order n, the degree of a vertex is an integer between 0 and $n - 1$. A vertex of degree 0 is said to be isolated: it is not connected to any other vertex.

5.4 Graph representation

5.4.1 Adjacency matrix

A graph can be represented by an $n \times n$ matrix (where $n = |X|$), called an adjacency matrix, which can only contain the values 0 and 1. Each row and each column of the matrix represents a node. Thus, a cell indicates the relationship between two nodes.

- 0 means that the two nodes are not connected by a directed arc,
- 1 means that the two nodes are connected by a directed arc.

Example

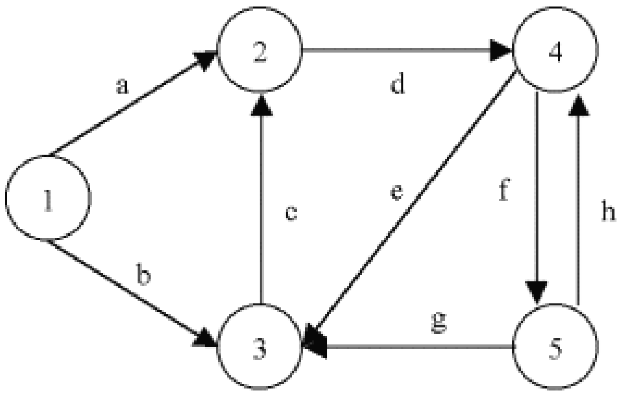


Figure 5.9: Adjacency matrix

For $n = 5 \Rightarrow M = (5 \times 5)$

.	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	1	1	0

Remark

- Only m cells of the matrix are non-zero out of n^2 cells.
- The non-zero elements on the diagonal represent loops.
- Two arcs with the same endpoints (multigraphs) cannot be represented with this matrix.
- The sum of the elements in row i gives $d^+(x_i)$.
- The sum of the elements in column j gives $d^-(x_j)$.
- This representation is efficient in terms of memory usage when the graph is sufficiently dense (i.e., when there are enough arcs).
- It allows for relatively straightforward implementation of algorithms.

5.4.2 Impact matrix

A graph can be represented by an $n \times m$ matrix (where $n = |X|$ and $m = |U|$), called an incidence matrix, which can contain only the values 0, 1, and -1. Each row of the matrix is associated with a node and each column with an arc. Thus, a cell indicates the relationship that exists between a node and an arc.

- 0 means that the node and the arc are not adjacent,
- 1 means that the node is the initial endpoint of the arc,
- -1 means that the node is the terminal endpoint of the arc.

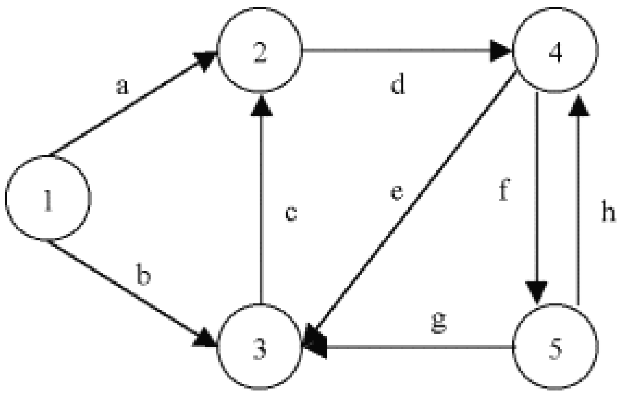


Figure 5.10: Impact matrix

Example

.	a	b	c	d	e	f	g	h
1	1	1	0	0	0	0	0	0
2	-1	0	-1	1	0	0	0	0
3	0	-1	1	0	-1	-1	0	0
4	0	0	0	-1	1	0	-1	1
5	0	0	0	0	0	1	1	-1

Remark

- Each column contains exactly one -1 and one +1 (directed graph).
- Only 2m cells of the matrix are non-zero out of mn cells.
- Loops cannot be represented.
- This representation takes up a lot of memory.
- Moreover, its use rarely yields good results for algorithms. In particular, for graph traversal, its use is difficult.
- However, for some problems such as the minimum cost flow, this matrix has significant direct meaning and can thus be useful.

5.4.3 List matrix

5.5 special graphs

5.5.1 Complete Graph

A graph is said to be complete if all nodes are adjacent to each other.

$$G = (X;U)/(x;y) \notin U \Rightarrow (y;x) \in U$$

Example:

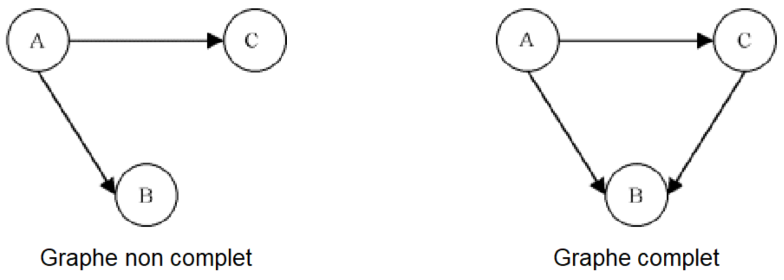


Figure 5.11: Complete Graph

5.5.2 Regular Graph

A graph is considered regular if all nodes have equal degrees. If the common degree is k, then the graph is k-regular.

$$\forall x, y \in X \text{ then } d(x) = d(y)$$

Example:

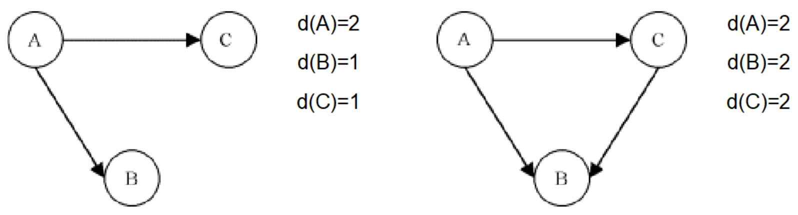


Figure 5.12: Regular and irregular Graph

5.5.3 Subgraph, Partial Graph, Partial Subgraph

- **Subgraph of G:** Consists of considering only a part of the vertices of X and the induced links by U. This is the graph G without some vertices and the adjacent (incident) arcs.
- **Partial Graph of G:** Consists of considering only a part of the edges of U. This is the graph G without some arcs.
- **Partial Subgraph of G:** It is a partial graph of a subgraph of G.

Example:

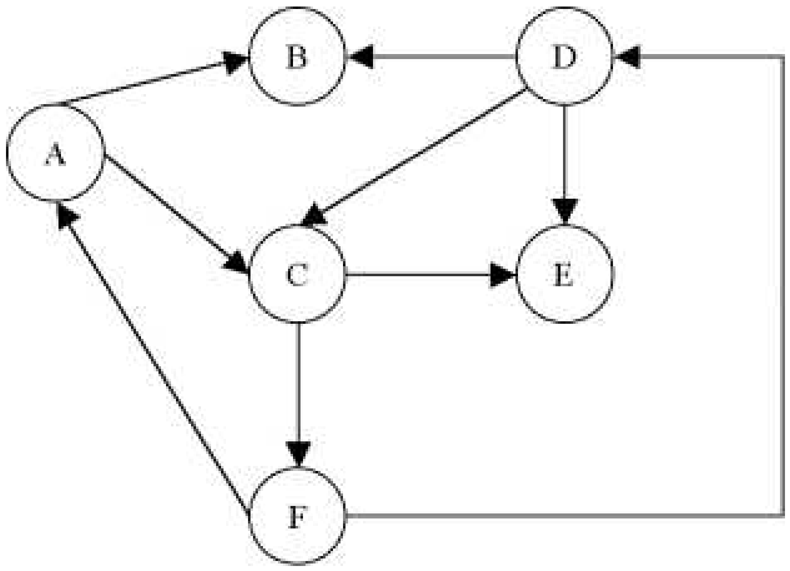


Figure 5.13: Graph G

5.5.4 P-Graph, Multi-Graph

A P-graph ensures that there are never more than P arcs of the form $(i; j)$ between any two nodes. If $P > 1$, the graph is a multi-graph.

Example:

- $\Rightarrow G1$: a 3-graph
- $\Rightarrow G2$: a 1-graph

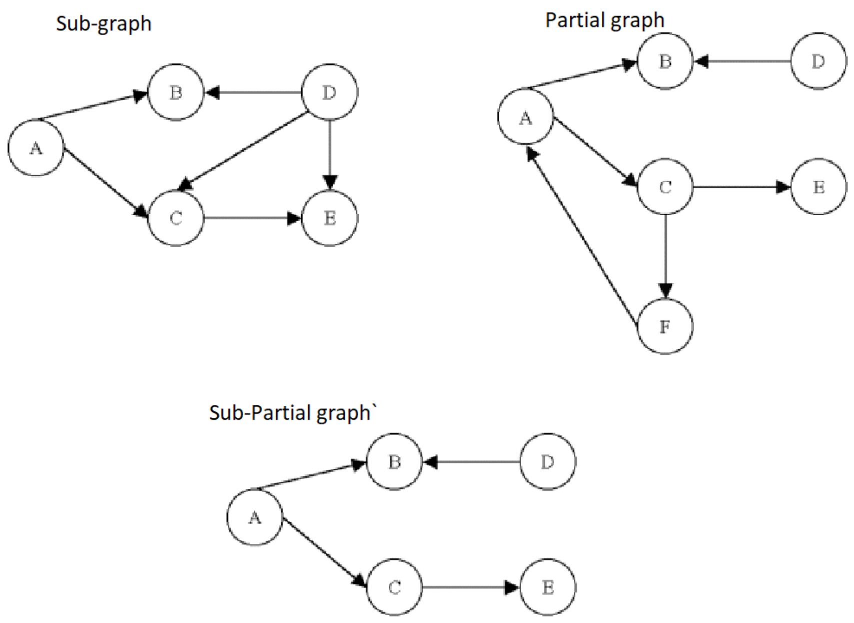


Figure 5.14: sub and Partial Graph of G

5.5.5 Acyclic Graph

A graph is acyclic if it contains no cycles.

5.5.6 Bipartite Graph

A graph is bipartite if its vertices can be divided into two sets X and Y, such that all edges of the graph connect a vertex in X to a vertex in Y. Let Let $X1 \cap X2 = \emptyset$ and for all $u = (x; y) \in U$, $x \in X1$ and $y \in X2$.

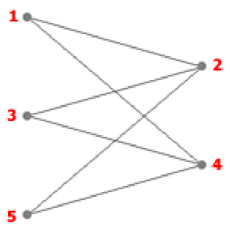


Figure 5.15: Bi-partite Graph of G

6
Path problem in a graph

6.1 Introduction

Implementing graph algorithms requires traversal techniques. Traversing a graph involves selecting a vertex and enumerating its neighboring vertices by following the edges as far as possible; each enumerated vertex may undergo local processing.

Searching for a path in a graph is beneficial when the goal is to determine the shortest path from a source point to a destination point. It aims to minimize the number of edges traversed. In a weighted graph, where edges are valued by distance, travel time, or other metrics, the objective is to find the path with the minimum value. However, in some cases, it is useful to search for the path of maximum length.

6.2 Definitions

Network

A network is a graph $G = (X, U)$ to which is associated a function $d : U \to \mathbb{R}$ that assigns a length to each arc. We denote $R = (X, U, d)$ for such a graph.

Length

The length of a path (chain, circuit, or cycle) is the sum of the lengths of each arc that composes it. By convention, a path (chain, circuit, or cycle) that does not contain any arc has a length of zero.

Absorbing Circuit

A circuit is said to be absorbing if its length is negative.

Example

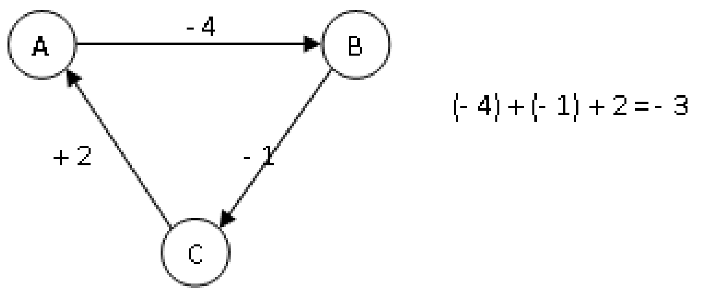


Figure 6.1: example absorbing circuit

6.3 Path in a graph

6.3.1 Search algorithm

The traversal starts from s (distance 0) and sequentially visits its neighbors (distance 1), then the neighbors of its unvisited neighbors (distance 2), and so on...

To traverse the graph from vertex s, we use a search algorithm.

Algorithm: *Traversal*

```
• Inputs: $G = (X, U)$ - Graph, $s$ - Vertex
• Initialize all vertices as unmarked; Mark $s$.
• While there exists an unmarked vertex adjacent to a marked vertex:
      Select an unmarked vertex $y$ adjacent to a marked vertex.
      Mark $y$.
• End While
```

There are two main "opposite" strategies to select the vertex to mark at each step:

- **Depth First Search (DFS):** In this exploration, the algorithm aims to go "deep" into the graph quickly, moving away from the starting vertex s. At each step, it selects a neighbor of the vertex marked in the previous step.
- **Breadth First Search (BFS):** In contrast, the algorithm aims to exhaust the list of vertices close to s before continuing to explore the graph further.

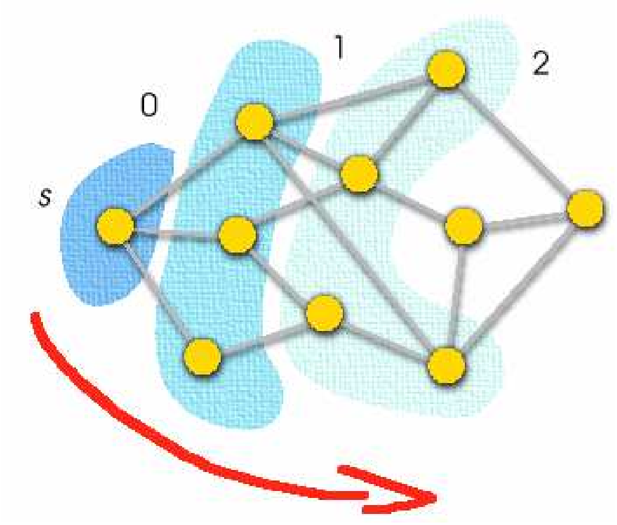


Figure 6.2: Search algorithm

6.3.2 Depth First Search (DFS)

DFS is useful for testing graph connectivity, determining connected components, detecting cycles, etc.

Principle:

DFS involves advancing along a path as far as possible, then traversing adjacent paths. Since a graph can contain cycles, it's essential to mark visited vertices. If DFS reaches a dead end without visiting all nodes, it needs to restart from another unvisited node.

Example

6.3.3 Breadth-First Search (BFS)

This algorithm is fundamental in solving certain problems such as the shortest path problem.

Principle:

First, visit the vertices that are 'i' arcs away from the starting vertex, then visit the vertices that are 'i + 1' arcs away, and so on.

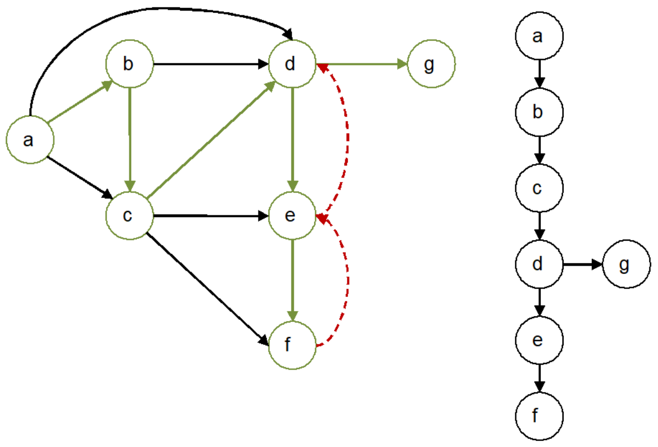


Figure 6.3: DFS Search algorithm

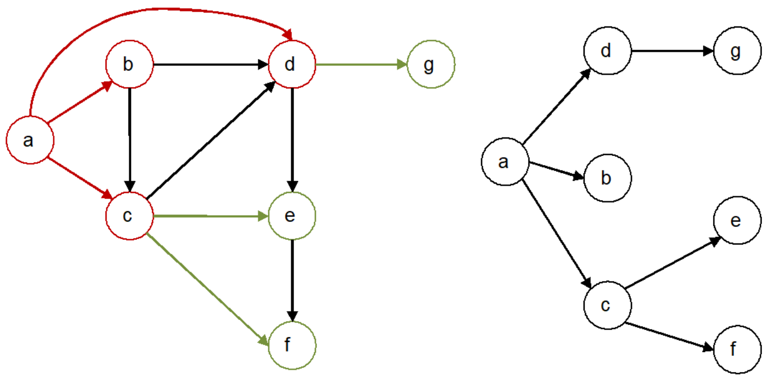


Figure 6.4: BFS Search algorithm

Example:

6.4 Path finding in an unvalued graph

6.4.1 Problem Positioning:

aph $G = (X, U)$ and two nodes a and b in G, the problem is to extract a path in G from a to b.

6.4.2 Generic Algorithm:

Principle:

We traverse the graph following the arcs and mark the visited nodes to avoid visiting them a second time. To perform this traversal, we have a set Z: the set of nodes remaining to be visited. Initially, there is only the node a. Then at each iteration, we take a node from Z, mark it to avoid visiting it again. We remove it from Z and put its successors in Z, only if they are not already marked. We stop if we reach the node b, or when $Z = \emptyset$.

6.5 Problem Positioning:

Given a graph $G = (X, U)$ and two nodes a and b in G, the problem is to extract a path in G from a to b.

6.6 Generic Algorithm:

Principle:

We traverse the graph following the arcs and mark the visited nodes to avoid visiting them a second time. To perform this traversal, we have a set Z: the set of nodes remaining to be visited. Initially, there is only the node a. Then at each iteration, we take a node from Z, mark it to avoid visiting it again. We remove it from Z and put its successors in Z, only if they are not already marked. We stop if we reach the node b, or when $Z = \emptyset$.

Algorithm:

Example

$$Z \leftarrow x_0$$

$$Accessible(x_0) \leftarrow true$$

First iteration:

$$x \leftarrow x_0; Z \leftarrow \emptyset$$

$$u = (x_0, x_1), (x_0, x_2)$$

Algorithm 18 GenericPathSearch

```
Require: $G = (X, U)$ : a graph; $a, b$ : two nodes of $G$
Ensure: Pred() : function indicating through which arc (or node) one arrives at
  a given node during the traversal from $a$
  Accessible() : function indicating if a node is accessible from $a$
  Z : set of nodes remaining to be visited

  for each $x \in X$ do
    Accessible(x) ← false
    Pred(x) ← Nil
  end for
  Z ← $\{a\}$
  Accessible(a) ← true
  while (Z $\neq \emptyset$) and (not Accessible(b)) do
    Choose $x \in Z$
    Z ← Z - {x}
    for each $u = (x, y) \in U$ do
      if not Accessible(y) then
        Z ← Z + {y}
        Accessible(y) ← true
        Pred(y) ← (x, y)
      end if
    end for
  end while
```

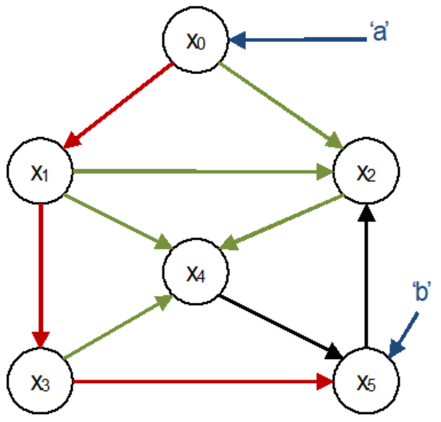


Figure 6.5: Example Search algorithm

- $$y \leftarrow x_1$$

  :
  $$Z \leftarrow x_1; Accessible(x_1) \leftarrow true; Pred(x_1) \leftarrow (x_0, x_1)$$

- $$y \leftarrow x_2 : Z \leftarrow (x_1, x_2); Accessible(x_2) \leftarrow true; Pred(x_2) \leftarrow (x_0, x_2)$$

Second iteration:

$$x \leftarrow x_1; Z \leftarrow x_2$$
$$u = (x_1, x_2), (x_1, x_3), (x_1, x_4)$$

- $$y \leftarrow x_2$$

- $$y \leftarrow x_3 : Z \leftarrow x_2, x_3; Accessible(x_3) \leftarrow true; Pred(x_3) \leftarrow (x_1, x_3);$$

- $$y \leftarrow x_4$$
  $$Z \leftarrow x_2, x_3, x_4; Accessible(x_4) \leftarrow true; Pred(x_4) \leftarrow (x_1, x_4)$$

6.7 Finding the shortest path

6.7.1 Definition:

In a weighted graph, the weight $c(p)$ of a path p is the sum of the weights of the edges along the path. In what follows, we will refer to the weight of a path as its length.

Example:

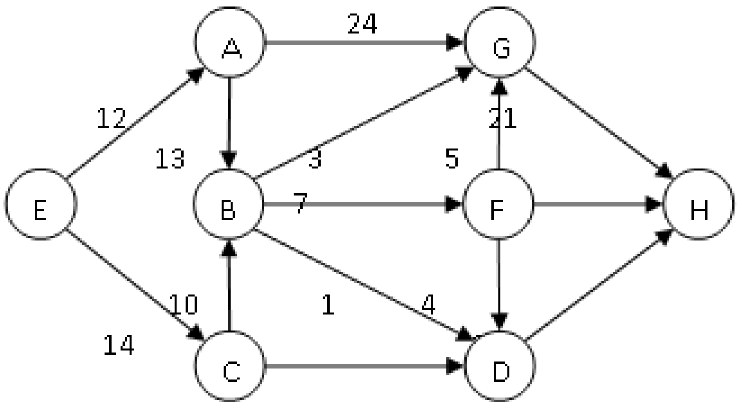


Figure 6.6: Example Search algorithm

The shortest path between two vertices E and H is defined as the path with the least weight connecting E to H.

2. Problem Positioning

Let $R = (X, U, d)$ be a network. The task is to find the path (or paths) of minimum (or maximum) length from one vertex to another given vertex.

The optimal path problem can be solved in many ways depending on the structure of the network:

- When the network is acyclic (but can be absorbing) $\Rightarrow$ Ford or Bellman algorithm;

- When the network has only non-negative lengths (but can have cycles) $\Rightarrow$ Dijkstra's algorithm.

Ford or Bellman Algorithm:

Finding the shortest path, minimum cost:

First step: Number the vertices of the weighted graph in any order, with the initial vertex denoted as x_0 and the final vertex as x_{n-1} (where $n = |X|$);

Second step: Assign values t_i to the vertices x_i such that:

- to $x_0 \Rightarrow t_0 = 0$;
- to $x_i \Rightarrow t_i = \infty \qquad$ for $1 \leq i \leq n-1$.

Third step: For each arc (x_i, x_j),

Fourth step: Repeat the third step until no arc can further decrease the t_i.

EXAMPLE 1:

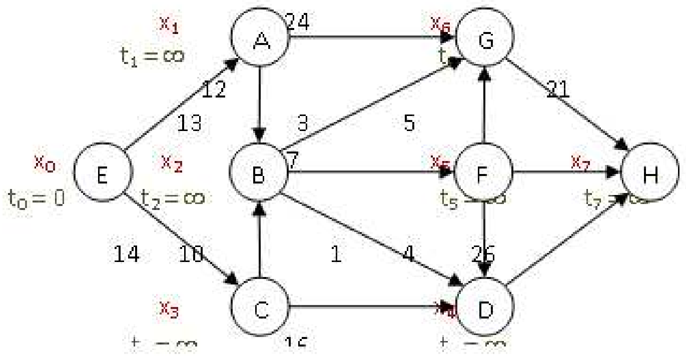


Figure 6.7: Example Ford-Bellman algorithm

1st step:

Number the vertices of the weighted graph in any order, with the initial vertex denoted as 'x0' and the final vertex as 'xn-1' (where n=|X|);

2nd step:

Assign values ti to vertices xi such that:

- at $x_0 \to t_0 = 0$;
- at $x_i \to t_i = 0$ with $1 \leq i \leq n-1$.

3rd step:

For each arc (x_i, x_j),

4th step:

Repeat the 3rd step until no arc can increase t_i anymore. NOTE:

- Ford's algorithm finds the shortest path between a vertex and all other vertices.
- When the graph is acyclic, it is preferable to number the vertices according to their generation levels.
- $T[i]$ is the weight of the shortest path(s) leading to x_i.

6.8 Dijkstra's algorithm

This algorithm determines the shortest paths from a vertex 's' to all other vertices in the network $R = (X; U; d)$. The lengths on the arcs are non-negative.

Principle:

We divide the vertices into two groups:

- Those for which we know the shortest path from 's', the set S;
- Those for which we do not know this distance yet, the set S'.

Initially, $S' = X$ and $S = \emptyset$, and all vertices x have $T[x] = +\infty$ except for s, where $T[s] = 0$.

At each iteration, we select from S' the vertex y that has the smallest distance to vertex x, and this vertex is moved to S. Then, for each successor y of x, we check if its shortest known distance so far can be improved by passing through x. If so, $T[y]$ is updated.

We then repeat the process with another vertex.

Algorithm:

Algorithm 19 GenericPathSearch

```
Require: $R = (X, U, d)$: a network with vertices $X$, arcs $U$, and distances $d$
Require: $s$: the starting vertex for Dijkstra's algorithm
Ensure: $T[]$: shortest distances from $s$ to all vertices; $Pred()$: predecessors in shortest paths
 1: Initialize sets and arrays:
 2: $S \leftarrow \emptyset$ {Set of vertices for which shortest path is known}
 3: $S' \leftarrow X$ {Set of vertices for which shortest path is not known}
 4: for each vertex $x \in X$ do
 5:     $T[x] \leftarrow +\infty$ {Shortest known distance from $s$}
 6:     Pred($x$) $\leftarrow$ Nil {Predecessor arc in shortest path}
 7: end for
 8: $T[s] \leftarrow 0$ {Distance from $s$ to $s$ is zero}
 9: while $S' \neq \emptyset$ do
10:     Choose $x \in S'$ such that $T[x] = \min\{T[y] : y \in S'\}$
11:     $S' \leftarrow S' - \{x\}$ {Move $x$ from $S'$ to $S$}
12:     $S \leftarrow S \cup \{x\}$
13:     for each arc $(x, y) \in U$ do
14:         if $T[y] > T[x] + d(x, y)$ then
15:             $T[y] \leftarrow T[x] + d(x, y)$
16:             Pred($y$) $\leftarrow (x, y)$
17:         end if
18:     end for
19: end while
```

EXEMPLE

Remark:

- Dijkstra's algorithm uses a greedy strategy, selecting the least costly vertex at each step; at each iteration, it fixes a vertex.
- In Dijkstra's algorithm, arcs are relaxed only once. In contrast, in Ford's algorithm, arcs can be relaxed multiple times.
- If there is an inaccessible vertex, the path terminates using the variable Sortie.

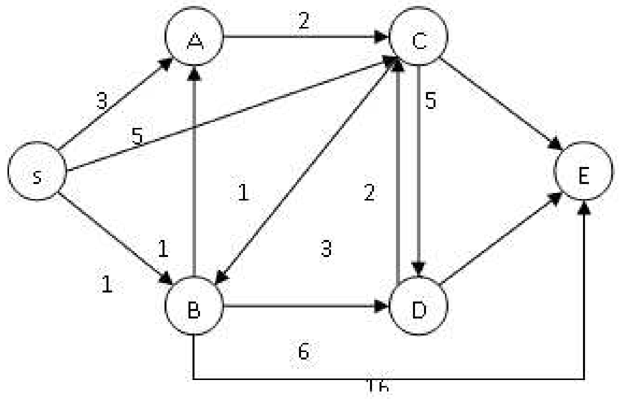


Figure 6.8: Example Dijkstra algorithm

7 Scheduling problem

7.1 Scheduling methods

Scheduling problems initially emerged in the planning of large projects with the aim of reducing their completion time. Such projects consist of numerous stages, also referred to as tasks. There exist temporal relationships between these stages, for example:

- A stage must start on a specific date;
- A certain number of tasks must be completed before another can begin;
- Two tasks cannot be performed simultaneously (e.g., they use the same machine);
- Each task requires a certain amount of manpower. Therefore, it is essential to avoid exceeding the total available manpower capacity at any given time.

All these constraints are not straightforward to consider in problem resolution. Here, we will focus only on the first two types of constraints. We aim to determine a schedule, an order of stages that minimizes the total project completion time. Based on this schedule, we will see that the timing of certain stages can potentially be adjusted without delaying the project, whereas others, known as "critical tasks," will delay the entire project with any local delay.

To address these types of problems, various methods exist such as the Gantt chart, the PERT (Program Evaluation and Review Technique) method, and the MPM (Méthode Potentiels Métra) method. These methods aim to:

- Resolve constraint compatibility issues;
- Design scheduling;
- Identify critical tasks.

7.2 Gantt chart

The Gantt chart, invented in the 1890s by the Polish engineer Karol Adamiecki, was later popularized by the American Henry Gantt, whose version of the chart bears his name. This chart, presented in tabular form, graphically represents the various tasks and their respective durations within a project.

In the previous figure, we observe that tasks (and sometimes the roles responsible for their execution) are listed on the vertical axis. Time units (days, weeks, months) chosen to sequence the project are plotted on the horizontal axis.

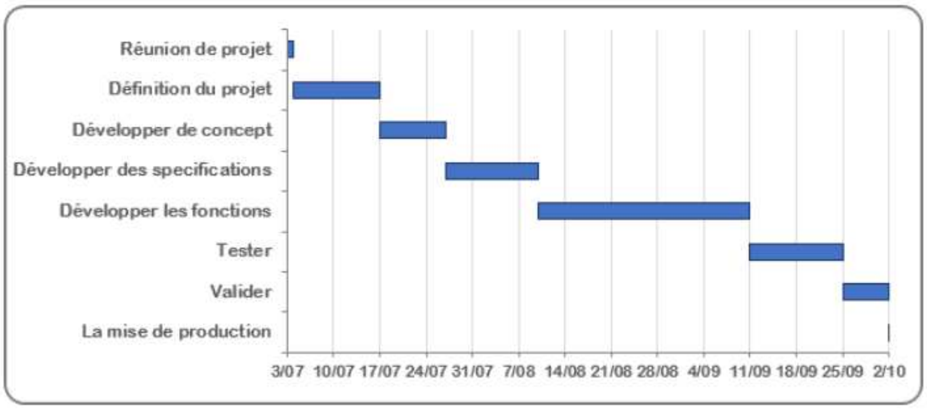


Figure 7.1: example of Grantt method

This means that each task is assigned a time unit, represented by a horizontal bar.

We then fully grasp the power of the Gantt chart, which, through a highly graphical representation, allows us to quickly understand:

- The different planned tasks,
- The start, end, and estimated duration of each task,
- Any overlaps between tasks,
- The start and end of the entire project.

It is important to note that the Gantt chart is often used in conjunction with the Program Evaluation and Review Technique (PERT) diagram, especially for complex projects involving task interdependencies.

By constructing a network, the PERT diagram prepares the Gantt chart: using a system that determines earliest and latest possible dates for each stage, project time management is facilitated.

7.3 PERT method

To present these scheduling problems, we can use the PERT (Program Evaluation and Review Technique) method, which involves ordering multiple tasks in the form of a graph based on their dependencies and chronology, all contributing to the completion of a project. This tool was created in 1957 for the US Navy (specifically for the development of the Polaris missile program) and enables the calculation of the best project completion time and the establishment of the corresponding schedule.

Example

To prepare vegetable soup, the tasks and their durations are:

- Buy vegetables (task A, duration: 30 minutes)
- Wash and peel vegetables (task B, duration: 5 minutes)
- Slice vegetables (task C, duration: 5 minutes)
- Boil salted water (task D, duration: 5 minutes)
- Cook vegetables (task E, duration: 1 hour or 60 minutes)
- Blend vegetables (task F, duration: 5 minutes)

Translated into PERT, the sequence is as follows:

Task	**Description**	**Dependencies**
A	Buy vegetables	None
B	Wash and peel vegetables	A
C	Slice vegetables	B
D	Boil salted water	None
E	Cook vegetables	D
F	Blend vegetables	E, C

To develop and utilize a PERT network, we can distinguish 5 main steps:

1. Establish the list of tasks

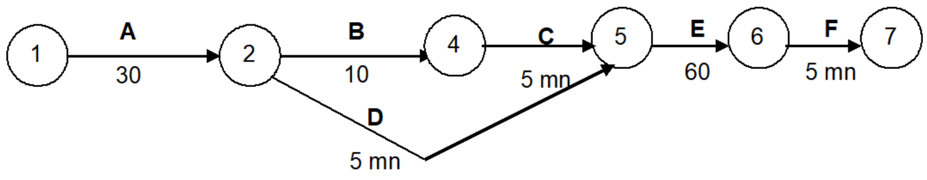


Figure 7.2: example of Scheduling

- Provide an exhaustive list of tasks to be executed.
- Evaluate the duration of tasks and determine the necessary resources to accomplish them.
- Encode tasks to facilitate the construction of the network (A, B, C, D,...)

Example: To determine the maximum duration of work required for the construction of a warehouse, consider Table 1:

Tasks	**Duration**
A. Study, design, and plan approval	4 days
B. Site preparation	2 days
C. Order materials (wood, bricks, cement, roof sheet)	1 day
D. Excavate foundations	1 day
E. Order doors, windows	2 days
F. Material delivery	2 days
G. Foundation pouring	2 days
H. Door, window delivery	10 days
I. Wall and roof construction	4 days
J. Installation of doors and windows	1 day

2. Determine the precedence relationships

   - Answer the following questions:
     (a) Which task(s) must be completed immediately before another can start?
     (b) Which task should follow a specific task?
   - This information is summarized in Table 2:

Table 2:

Preceding	To Complete This Task	Following
-	A	C, D, E
-	B	D
A	C	F
A, B	D	G
A	E	H
C	F	G
D, F	G	I
E	H	J
G	I	J
H, I	J	-

3. Draw the PERT network

   - A network consists of stages and tasks (A, B, C, D).
   - The presentation code is as follows:
     (a) Symbolize each stage with a circle (start or end of a task).
     (b) Use an arrow to signify a task (above the arrow, indicate the task code; below, specify its duration).

   Rules for Representing a PERT Network:

   (a) Each task is represented by exactly one arrow.

   (b) Two tasks cannot be represented by two arrows with the same origin and endpoint. If tasks are simultaneous, represent them with separate arrows starting from the same origin.

This structured approach in LaTeX will effectively present the steps involved in constructing and utilizing a PERT network for project management and scheduling. Adjust the formatting and details as per your specific documentation needs.

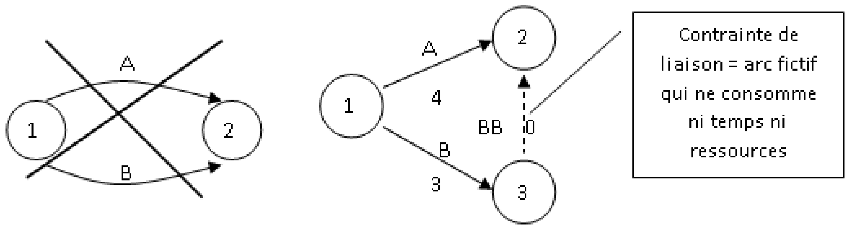

A
1
2
B
A
2
1
4
BB
0
B
3
3
Contrainte de liaison = arc fictif qui ne consomme ni temps ni ressources

Figure 7.3: example

REMARK:

To determine the first task(s) = the only one(s) not listed in the left column of the precedence table.

- Draw the PERT network.

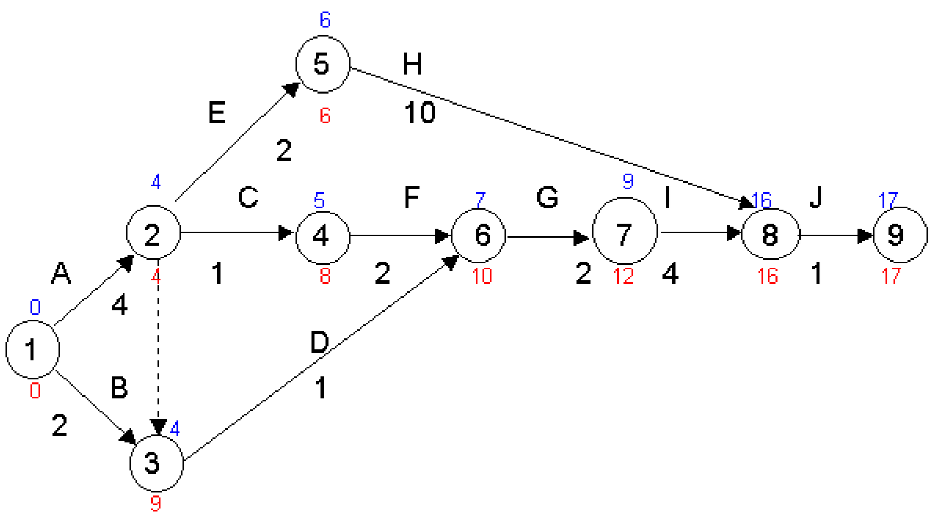


Figure 7.4: example

7.3.1 Calculate task dates and determine the critical path

Once the durations of all tasks in the network have been estimated, we can calculate the start and finish dates for each of them. This is done in 2 steps:

- Calculate earliest start dates:

  Forward pass = earliest dates: We determine the earliest possible start dates for each task in the project. The technique is as follows:

  - Start with 0 (step 1 = 0), represented by a rectangle above the step.
  - For subsequent steps:
    - ∗ If there is only one task (one path) between two steps:

$$\text{Earliest date } j = \text{Earliest date } i + \text{Task duration } T_{i,j}$$

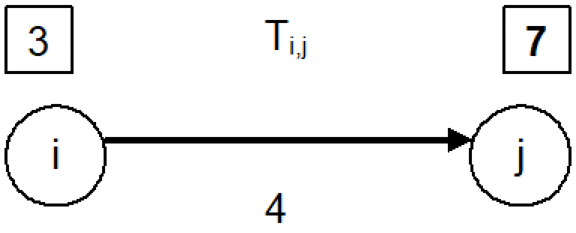


Figure 7.5: example

- $*$ If there are multiple paths to reach step j:

  Earliest date $j = \max((\text{Earliest date } i + \text{Task duration } T_{i,j}), (\text{Earliest date } k + \text{Task d}$

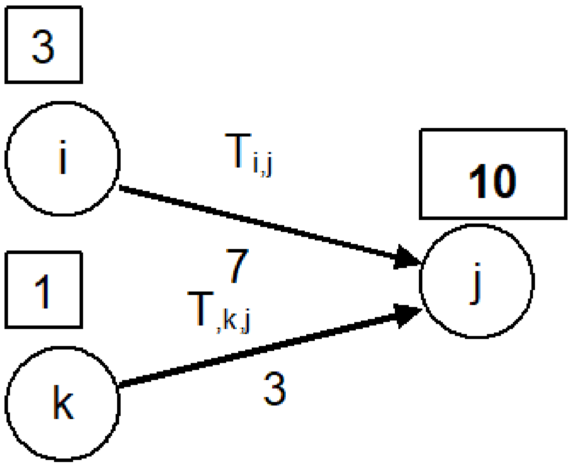


Figure 7.6: example

- Determine the critical path:

  Highlight the path on the network that, formed by the succession of tasks, gives the longest time. It is called critical because any delay in one of the tasks on this path will delay the project completion. Start from the terminal point and identify all steps that satisfy the following equality:

$$\text{Earliest date } j - \text{Earliest date } i - \text{Task duration } T_{i,j} = 0$$

7.3.2 Calculate latest dates

Backward pass = latest dates: Determine the latest possible dates by which tasks must be completed without affecting the optimal project finish time. The technique is as follows:

- Start with the terminal step using its earliest date, represented by a red circle.
- For subsequent steps:
  - If only one arrow leaves step i:

$$\text{Latest date } j = \text{Latest date } i - \text{Task duration } T_{i,j}$$

  - If there are multiple arrows leaving step i:

$$\text{Latest date } i = \min((\text{Latest date } j - \text{Task duration } T_{i,j}), (\text{Latest date } k - \text{Task duration }$$

7.3.3 Calculate total margins for each task

Maximum time range in which the task can be performed without changing the project completion date:

$$\text{Total margin } = \text{Latest finish date } j - \text{Earliest start date } i - \text{Task duration } T_{i,j}$$

Task	Total Margin
A	4 - 0 - 4 = 0
B	9 - 0 - 2 = 7
C	8 - 4 - 1 = 3
D	10 - 4 - 1 = 5
E	0
F	10 - 5 - 2 = 3
G	3
H	0
I	3
J	0

7.4 MPM Method

The execution of a project often involves the completion of various tasks. Some of these tasks can be performed simultaneously, while others need to be executed in a specific order.

7.4.1 Definition

Scheduling a project involves organizing the project while respecting task precedence constraints, with the goal of minimizing the total duration of completion.

Note:

The BTS program offers a choice between two scheduling methods: PERT method and MPM method. The method presented here is the MPM method (Méthode des Potentiels Métra).

7.4.2 Method

To construct a scheduling graph, the following steps are performed:

1. Start by determining the level of each task in the graph.
2. Represent the project with a weighted graph, where:
   - each task is represented by a vertex,
   - vertices are vertically aligned by level,
   - arcs represent precedence constraints (an arc goes from i to j if task j depends on task i),
   - the weight of each arc is the duration of the task that initiates the arc,
   - two vertices (not corresponding to tasks) are placed at the ends of the graph: Start and End.

Example

Task Breakdown for Warehouse Construction:

- **A - Plan acceptance**
  Immediate Predecessors: None
  Duration: 4 days
- **B - Site preparation**
  Immediate Predecessors: None
  Duration: 2 days
- **C - Material ordering**
  Immediate Predecessors: A
  Duration: 1 day

- **D - Foundation digging**
  Immediate Predecessors: A, B
  Duration: 1 day
- **E - Door and window ordering**
  Immediate Predecessors: A
  Duration: 2 days
- **F - Material delivery**
  Immediate Predecessors: C
  Duration: 2 days
- **G - Foundation pouring**
  Immediate Predecessors: D, F
  Duration: 2 days
- **H - Door and window delivery**
  Immediate Predecessors: E
  Duration: 8 days
- **I - Walls, frame, roof**
  Immediate Predecessors: G
  Duration: 4 days
- **J - Door and window installation**
  Immediate Predecessors: H, I
  Duration: 1 day

Bibliography

[1] VARDY, Alexander. Algorithmic complexity in coding theory and the minimum distance problem. In : Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. 1997. p. 92-109.

[2] WELCH, William J. Algorithmic complexity: three NP-hard problems in computational statistics. Journal of Statistical Computation and Simulation, 1982, vol. 15, no 1, p. 17-25.

[3] RUFFINI, Giulio. Models, networks and algorithmic complexity. arXiv preprint arXiv:1612.05627, 2016.

[4] ZENIL, Hector, KIANI, Narsis A., et TEGNÉR, Jesper. Methods of information theory and algorithmic complexity for network biology. In : Seminars in cell and developmental biology. Academic Press, 2016. p. 32-43.

[5] THOMAS, H., et al. Introduction to algorithms. 2009.

[6] SEDGEWICK, Robert et WAYNE, Kevin. Algorithms. Addison-wesley professional, 2011.

[7] ALI, Irfan, NAWAZ, Haque, KHAN, Imran, et al. Performance comparison between merge and quick sort algorithms in data structure. International Journal of Advanced Computer Science and Applications, 2018, vol. 9, no 11.

[8] Christian Prins : Algorithmes de graphes (avec programmes en Pascal) Eyrolles, Paris, 1994

[9] Bernard Roy : Algebre moderne et théorie des graphes TomeII, Dunod, 1989 Le livre de M.Gondrou et M.Minoux existe en version francaise Graphes et Algorithmes, Eyrolles, Paris 1984

[10] N. Belharat, Recherche operationnelle : Theorie des graphes, Pages bleus

[11] BONDY, John Adrian, MURTY, Uppaluri Siva Ramachandra, et al.Graph theorywith applications. London : Macmillan, 1976.

[12] WEST, Douglas Brent, et al.Introduction to graph theory. UpperSaddle River : Prentice hall, 2001.

[13] BOLLOBAS, Bela. Graph theory: an introductory course. Springer Science and Business Media, 2012.

[14] CHARTRAND, Gary et OELLERMANN, Ortrud R. Applied and algorithmic graph theory. New York : McGraw-Hill, 1993.

[15] BOLLOBAS, Bela. Modern graph theory. Springer Science and Business Media, 2013.