University of Ain Temouchent – Belhadj Bouchaib

Faculty of Science and Technology

Department of Mathematics and Computer Science

Teaching handout

Dr Mansour MEDEDJEL

Associate Professor - Rank A

Algorithms and Data Structures

Lectures and Exercises

Course for 2nd year Bachelor's students in Computer Science

2025 - 2026

Preface

As the field of computer science continues to shape and transform the modern world, the foundational principles of algorithms and data structures remain as essential and relevant as ever. This course handout represents the culmination of years of teaching experience and practical application in this essential field. The objective of this resource is to provide students with a robust foundation that integrates theoretical knowledge with practical application.

This handout is intended for second-year computer science students. It is designed to support both those who are new to the subject and those with prior experience, by providing a clear and structured introduction to fundamental concepts. The content is organized progressively, in alignment with the official program, beginning with essential principles and gradually advancing to more complex topics, in order to facilitate a smooth and effective learning experience.

I would like to express my gratitude to colleagues and students whose questions, feedback, and insights have helped shape and refine this handout over time. Their valuable input has been instrumental in creating a comprehensive and accessible educational resource that addresses key challenges in understanding these concepts.

Remember that mastering algorithms and data structures is not simply a matter of memorizing implementations, but about developing a problem-solving mindset that will serve you throughout your career in the field of computer science. Students are encouraged to engage deeply with these concepts, practice consistently, and revisit earlier material whenever necessary to reinforce understanding and build lasting proficiency.

I hope this material provides you with a rewarding and intellectually enriching learning experience.

Table of content

List of figures

List of tables

Introduction

The study of algorithms and data structures represents a fundamental pillar of computer science. This handout is intended to equip students with the essential skills required to understand, design, and analyze efficient algorithms, as well as the data structures that support them. Understanding these topics is critical for solving complex computational problems effectively and for enhancing software performance.

Course objectives:

This course handout is designed to provide students with:

- A solid understanding of fundamental algorithmic concepts.
- Analytical tools to assess the efficiency of algorithms.
- Mastery of key data structures and their applications.
- Practical skills through targeted exercises.

Structure of the course:

This course is organized into two complementary main parts:

PART 1 : Lectures

This part develops fundamental concepts through five progressive chapters:

I. **Recursion reminder**: We begin by revisiting the powerful concept of recursion, essential in many advanced algorithms. This chapter explores the recursive process and how the call stack works.

II. **Algorithm analysis and complexity**: This chapter provides a comprehensive introduction to algorithm analysis, with a focus on evaluating efficiency in terms of time complexity. Students will explore asymptotic notations such as Big O, and learn to classify algorithms according to their computational complexity. The chapter also develops analytical skills necessary to assess both iterative and recursive algorithms, equipping students with essential tools for comparing and optimizing algorithmic performance.

III. **Sorting Algorithms**: This chapter explores a range of techniques for organizing data, starting with basic methods such as selection sort, insertion sort, and bubble

sort, and progressing to more advanced algorithms including merge sort, quicksort, and binary tree sort. It also includes a comparative analysis of their performance, highlighting differences in efficiency based on time complexity.

IV. **Tree Structures**: The fourth chapter introduces the hierarchical organization of data using tree structures. It begins with an overview of N-ary trees, covering their fundamental properties and implementations, and then focuses on specialized forms such as binary search trees and heaps. These structures are examined for their efficiency and suitability in various computational scenarios.

V. **Graphs**: This final chapter focuses on graph structures, which are essential for modeling complex relationships in computer science and related fields. It introduces fundamental graph properties and common representation methods, including adjacency matrices and adjacency lists. Special attention is given to core traversal algorithms—depth-first search (DFS) and breadth-first search (BFS)—which underpin numerous real-world computational solutions.

PART 2 : Exercises

The second part is devoted to a series of carefully designed exercises intended to reinforce conceptual understanding and enhance problem-solving skills. By applying theoretical knowledge to practical situations, these exercises support the consolidation of learning and foster the development of analytical thinking. Some exercises are intentionally left without solutions to encourage independent reasoning and deeper engagement with the material.

Finally, this course handout aims to provide readers with the theoretical foundations essential for understanding and solving problems related to algorithms, data structures, and graph-based models in computer science. Designed to serve both as a classroom companion and a long-term reference, it supports the development of algorithmic thinking applicable across a broad range of domains—including software development, data science, and other areas—regardless of the programming languages or technologies used.

PART 1 : Lectures

Chapter I : Recursion reminder

1. Introduction

In computer science, and more specifically in algorithmics, **recursion** appears at two main levels:

- **Algorithm/Program**: An algorithm is said to be recursive if it calls itself. In practice, this means that the recursive component is implemented as a function (or subroutine) that invokes itself to solve sub-problems.
- **Data structures**: Recursion is also closely linked to recursive data structures such as **linked lists**, **trees**, and others, which are frequently used in algorithms that naturally require recursive processing.

Example: Factorial

The factorial of a non-negative integer n is defined as:

$$n ! = 1 \times 2 \times 3 \times \ldots \times n$$

$$= n \times (n-1) !$$

With a base case:

$$0! = 1$$

a) Iterative solution:

```
Function Fact (n: integer): integer
Var i, f: integer;
Begin
    If (n = 0 or n = 1) Then
      f ← 1;
    Else
      f ← 1;
      For i from 2 to n do
            f ← f * i;
      EndFor
    EndIf
    Return f;
End
```

The function **Fact** above takes an integer n as input and returns the factorial of n. If n is 0 or 1, the result is 1. Otherwise, it initializes a variable f to 1 and uses a loop to multiply f by every integer from 2 up to n.

b) Recursive solution:

```
Function Fact_rec(n: integer): integer
Begin
  If (n = 0 or n = 1) Then
    Return 1;
  Else
    Return n * Fact(n - 1);        // Recursive call
  EndIf
End
```

The function **Fact_rec** calculates the factorial of a non-negative integer n. If n is 0 or 1 (the base case), it returns 1. Otherwise, it returns n multiplied by the factorial of n - 1 (recursive case). The recursion continues until it reaches the base case.

2. Process of recursion

Recursion works by allowing a function to call itself in order to solve a problem step by step, breaking it down into simpler sub-problems with each call. The process continues until it reaches a **base case**, which provides a direct solution without any further recursion [1].

Each recursive call is placed on the call stack, with its own parameters and local variables. Once the base case is reached, the results are returned step by step, unwinding the stack until the final result is computed.

2.1. The stopping condition (Base case)

Since a recursive function calls itself, it is essential to define a stopping condition (also known as the base case) for the recursive calls. Without this condition, the function would never stop, resulting in infinite recursion and ultimately a **stack overflow error**.

The base case must always be checked first. If the base condition is not met, the function proceeds with the recursive call to solve a smaller or simpler version of the problem.

General syntax:

A recursive function typically follows this structure:

```
Function Name (parameters)
  If (base condition is met) Then
        Return base result;
  Else
        Return recursive call with updated parameters;
```

Where:

- **Function Name**: The name of the recursive function.
- **Parameters**: Inputs that are usually modified in each recursive call.
- **Base case**: The stopping condition that ends the recursion.
- **Recursive case**: The part where the function calls itself with updated arguments.

2.2. The call stack

When there is no recursion, it is generally possible to reserve the necessary memory areas for each function call at compile time. However, in the case of recursive functions, it is impossible to predict in advance how many calls will be made. As a result, recursion requires dynamic memory allocation, which is managed through the call stack.

The call stack of a recursive algorithm is the memory structure that stores all local variables, parameter values, and return addresses for the functions currently being executed. Each new recursive call adds a new frame to the stack, and once the function returns, that frame is popped off the stack.

Note:

*The call stack has a **limited size**.*

*Poor use of recursion — such as making **too many recursive calls** without a proper base case or with very deep recursion — can lead to a **stack overflow**, which causes the program to crash.*

Example: 4!

- **The call phase: Stacking (Pushing onto the Stack)**

Each recursive call adds a new frame to the call stack — this is known as **pushing onto the stack**. In the case of Fact(4), the calls proceed as follows:

Fact(4)
→ 4 * Fact(3)
→ 3 * Fact(2)
→ 2 * Fact(1)
→ 1 **(base case).**

As the calls are made, each one is pushed onto the call stack. Once the base case is reached, the stack unwinds step by step.

- **The return phase: Unstacking**

After reaching the base case, the recursive function begins to return, and each function call is **popped off the stack** in reverse order of the calls:

Fact(1) → returns 1
Fact(2) → returns 2 * 1 = 2
Fact(3) → returns 3 * 2 = 6
Fact(4) → returns 4 * 6 = 24

Each return passes its result to the previous call until the original call receives the final result.

- **Illustration**

Stacking:

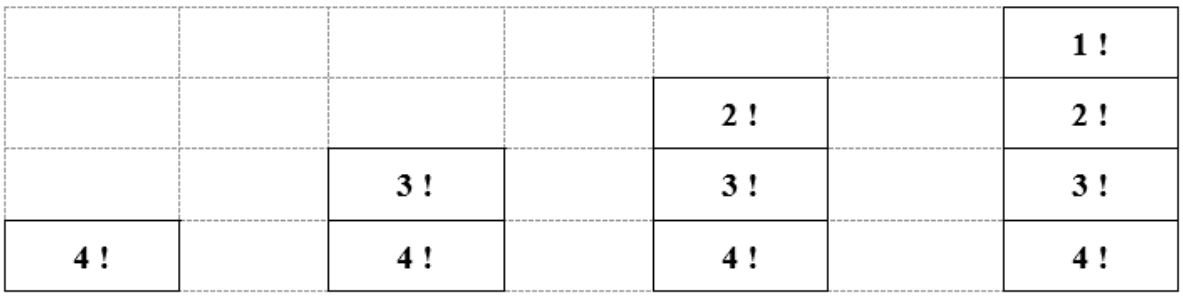


Figure I.1: Execution stack – Stacking.

Unstacking:

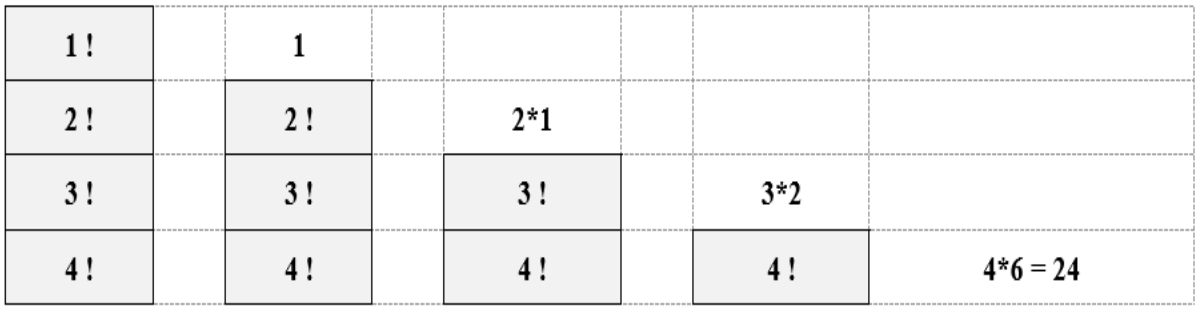


Figure I.2: Execution stack - Unstacking

2.3. Order of recursive calls

Let's consider a procedure D() that displays integers in decreasing order, from n down to one :

```
Procedure D(n: integer)
Begin
  If (n > 0) Then
        write (n);
        D(n - 1);
  EndIf
End
```

Each call to D(n) will first print the current value of n, then recursively call D(n-1), which means the numbers are printed before the recursive step.

For example, calling D(4) will produce the following output:

```
4
3
2
1
```

Let's now invert the order of the print statement and the recursive call in the procedure D():

```
Procedure D(n: integer)
Begin
  If (n > 0) Then
        D(n - 1);
        write (n);
  EndIf
End
```

This time, the function calls itself recursively first, and prints the value only after returning from the deeper call. This results in a bottom-up behavior.
For example, calling D(4) will produce the following output:

1
2
3
4

Thus, the order in which the instructions are placed relative to the recursive call significantly affects the behavior of a recursive function.
Understanding this difference is crucial when designing recursive functions, especially when the order of operations is important (e.g. traversing structures, generating sequences, or building solutions step by step).

3. Divide and conquer recursive algorithm

The **Divide and conquer** paradigm is an algorithmic strategy used to solve complex problems by breaking them down into simpler subproblems.
It typically follows three main steps:

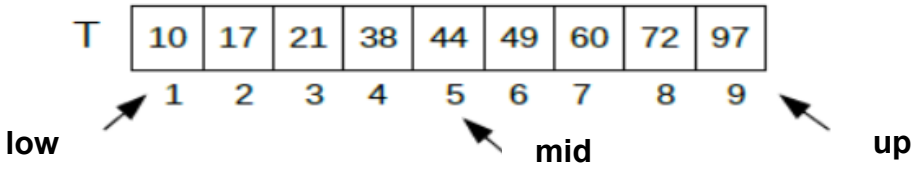
- **Divide**: Split the problem into a number of smaller subproblems, which are simpler instances of the original problem.
- **Conquer**: Solve each subproblem independently. If a subproblem is still too complex, apply divide and conquer recursively.
- **Combine**: Combine the solutions of the subproblems to form the final solution to the original problem.

Thus, designing a recursive algorithm to perform a specific operation T on a dataset D essentially means following these key steps within the divide and conquer paradigm:

a) **Decompose the task T** into smaller sub-tasks of the same nature, operating on smaller portions of the data D.
b) **Identify the base case** (stopping condition) and define the **recursive formula**.
c) **Optionally, test the logic** with a concrete example to ensure correctness.
d) **Write the complete recursive algorithm** based on the above insights.

3.1. Application: Binary search (Dichotomous search)

Let us consider the task of writing a recursive function that performs a **binary search** to find **an element x** in a **sorted array** T of integers.



The element x is either in the first half or the second half of T. To determine which, it is compared to the middle element.

We will assume that the function returns:

- the index of element **(i)**, if it exists,
- the value **(-1)**, otherwise.

a) Decomposition of the process:

Searching for an element in the array will lead, if it is not found in the middle, to restarting the search on one half of the array, then on a quarter, and so on. At each recursive call, it is necessary to know between which indices, i_{low} and i_{up}, the element should be searched.

b) Stopping condition:

The search ends in one of two situations:

- **The element is found** – when the element being searched for matches the middle element of the current interval.
- **There are no more positions to check** – when the starting index i_{low} becomes greater than the ending index i_{up} , which means the search space is empty and the element is not in the array.

c) Test cases:

For example, if x is:

38 → the function returns: 3

97 → the function returns: 8

25 → the function returns: -1

d) Algorithm:

```
Function Dicho(T: array of integers, low, up, x: integer): boolean
Var mid: integer;
Begin
    If (low <= up) Then
        mid ← (low + up) div 2;  // Integer division
        If (x = T[mid]) then Return True;
        Else If (x < T[mid]) then Return Dicho(T, low, mid - 1, x);
            Else Return Dicho(T, mid + 1, up, x);
    Else Return False;
End
```

Note: The initial call is **Dicho(T, 0, n-1, x)** where n is the size of T.

4. Conclusion

In conclusion, recursive algorithms are often appreciated for their simplicity and clarity. They tend to be shorter and more intuitive to write and understand than their iterative counterparts. However, this simplicity comes at a cost: recursive algorithms can be less efficient in terms of execution time and are more demanding in memory usage. Due to the use of the call stack for each recursive call, there is also a risk of stack overflow if the recursion is too deep. Therefore, while recursion offers elegant solutions for many problems, it should be used with consideration of performance and memory constraints. Table I-1 shows the main differences between recursive and iterative algorithms.

Table I-1: Recursive vs. iterative algorithms.

Criteria	Recursive	Iterative
Readability and simplicity	Recursive algorithms can be more elegant for certain problems, like tree search or sorting, as they are simpler and easier to understand.	Iterative algorithms can sometimes be more verbose and less intuitive for certain problems.
Memory usage	It can consume a lot of memory due to the call stack. If the recursion depth is high, it can lead to a stack overflow.	It is generally more memory-efficient because it uses local variables.

Performance	It can be less efficient due to the overhead of function calls, similar to high recursion depth.	It is often faster.
Cases of use	Ideal for problems like tree search, recursive sorting algorithms, and so on.	Ideal for simple loops, iterating over arrays, and so on.

Ultimately, the choice between the recursive and iterative approaches depends on the problem at hand, performance requirements, and preferences for code readability. It is often beneficial to evaluate both approaches and select the one that best aligns with the goals.

Chapter II : Algorithm analysis and Complexity

1. Introduction

An algorithm is considered effective if it satisfies the following essential criteria:

- **Correct**: The algorithm must perform the tasks it was designed to accomplish accurately and reliably. In other words, it should produce the expected results and operate without errors for all valid inputs it receives.
- **Complete**: The algorithm must handle all possible scenarios that might arise during its execution. It should address every conceivable case and provide an appropriate response for each situation, ensuring that no scenarios are left unaddressed.
- **Efficient**: The algorithm must accomplish its task in an optimal manner, minimizing the resources required. This means the algorithm should execute with minimal cost in terms of execution time (speed) and memory usage (space), while still performing its functions correctly.

Example: Polynomial Evaluation

Consider $P(x)$ as a polynomial of degree n:

$P(x) = a_0 + a_1.x + \dots + a_{n-1}.x^{n-1} + a_n.x^n$

Where:

n is a non-negative integer,

$a_0, a_1, a_{n-1}, \dots, a_n$ are the coefficients of the polynomial.

1st variant:

```
Begin
        P←0 ;
        For i from 0 to n
          P ← P+ ai*X^i ;
        End For
End
```

Total operations:
(n+1) additions
(n+1) multiplications
(n+1) exponentiations

2nd variant:

```
Begin
        Inter ← 1 ; P ← 0 ;
        For i from 0 to n
          P ← P + Inter *ai ;
          Inter ← Inter * X ;
        EndFor
End
```

Total operations:
(n+1) additions
2(n+1) multiplications

3rd variant:

In this variant, we apply Horner's scheme [1] [2]:

$$P(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x(a_n)) \ldots))$$

```
Begin
        P ← an ;
        For i from n-1 downto 0
          P ← P*x + ai ;
        EndFor
End
```

Total operations:
n additions
n multiplications

Thus, it is crucial to conduct an analysis of an algorithm prior to its implementation. This analysis involves assessing its efficiency in terms of both time and memory usage. Furthermore, algorithmic analysis enables the comparison of various algorithms, thereby facilitating the selection of the most optimal solution.

2. Algorithm analysis

Algorithms are evaluated based on their resource consumption, particularly in terms of execution time and memory usage for data storage and processing.

Algorithm analysis involves determining an algorithm's time and space complexity relative to the problem size, typically measured by the size of the input data.

- **Time complexity** refers to the execution time.
- **Space complexity** refers to the memory required for execution.

Consequently, an algorithm's complexity is assessed independently of the underlying machine or programming language, focusing solely on the size of the input data it processes.

Note: *The remainder of this course focuses on the time complexity of algorithms.*

[1] ***William George Horner (1786 - 1837) : a British mathematician.***

3. Time complexity

Analyzing an algorithm's time complexity consists of determining the number of elementary operations needed to solve a given problem. An elementary operation is any assignment, logical test (=, <, >, ...,), arithmetic operation (+ , - , * , /), and function call. When a function (or procedure) is called, its cost is the total number of elementary operations generated by the function or procedure call.

Consequently, the computational time required by an algorithm on a given machine is directly proportional to the number of elementary operations performed. The selection of these operations, as well as the input parameter used to measure the problem size, depends on the specific problem being addressed [3].

For example:

- Sorting an array of integers:
  - Basic operations: comparisons and assignments.
  - Parameters: size of the array.
- Matrix addition:
  - Elementary operations: addition and assignment.
  - Parameters: size of the matrix.

3.1. Rules for calculating the number of operations

In this section, we examine the rules for estimating the complexity of an algorithm [4].

- **Rule 1** : Sequences of instructions

  The number of operations in the instruction sequences (separated by ;) is accumulated.

  A ← 0 ; (1 op.)
  C ← 0 ; (1 op.)
  A ← A+1 ; (2 op.)
  B ← A+C; (2 op.)

  There are 6 operations.

- **Rule 2** : Tests (or conditional statements)

  In general, it is difficult to know the exact number of statements of a type "If (condition) I1 Otherwise I2". A markup can therefore be applied as follows:

  # (**If**(condition) I1 **Else** I2 ;) ≤ #(condition) + max(#(I1), #(I2))

  Where # is the number of operations.

- **Rule 3** : Loops (or iterative statements)

  This is the sum of the number of operations during the execution of the i^{th} iteration of the loop, for example: **For** (i **from**...) I(i);

  So, #(**For** (i **from** ...) I(i);) = ∑#(I(i))

- **Rule 4** : Functions calls

  For simple functions, the same calculation rules defined previously are applied. However, in the case of recursive functions, counting requires the expression of a recurrence equation and its resolution.

  For example:

```
Function fact(n : integer) : integer
Begin
        If (n>1) Then
                Return(n*fact(n-1));
        Else Return (1) ;
        EndIf
End
```

  The recurrence equation is expressed as follows:

  C(0/1) = ic (integer comparison: elementary operation)

  C(n) = ic + C(n-1) + im (integer multiplication: elementary operation)

  Thus, *C(n) = n * ic + (n-1) * im*

Note: *It should be noted that:*

i. *The analysis of time complexity is different from the analysis of the exact execution time of an algorithm, which depends on several factors such as the machine, the compiler, and the language used.*

ii. *Spatial complexity is calculated in the same way, but in terms of memory cells (bytes) rather than the number of elementary operations.*

3.2. Types of complexity

Let:

- D_n *:* the dataset *d* of size *n,*
- *C(d) :* the cost of executing the algorithm on *d.*

There are three types of complexity [5]:

- **Worst-case complexity**: This corresponds to the maximum cost of the algorithm for a dataset of size *n*.

  $C_{max}(n) = max \{C(d), d \in D_n\}$

- **Best-case complexity:** This corresponds to the minimum cost of the algorithm for a dataset of size *n*.

  $C_{min}(n) = min \{C(d), d \in D_n\}$

- **Average-case complexity:** This is the average complexity of the algorithm across a dataset of size *n*.

  $C_{avg}(n) = \sum \{C(d) \times p(d), d \in D_n\}$

  where *p(d)* is the probability that *d* is the input to the algorithm.

To illustrate, we will calculate the complexity of a sequential search algorithm in an array of n elements, assuming that the indices vary from 0 to n-1.

The following algorithm will be analyzed:

```
Begin
    // Input: an array 'A' of size 'n'; an element 'x'
    // Output: 'i' if 'x' appears at index 'i' of 'A', otherwise '-1'
    i← 0 ;                          // c1 : cost of the assignment operation
    While (i<n and A[i] ≠ x) do   // c2 : cost of the two test operations
        i← i+1 ;        // c3 : cost of the assignment and addition operations
    EndWhile
    If (A[i]=x) Then     // c4 : cost of the test operation
        Return i ;              // c5 : cost of the return function
    Else Return -1 ;            // c6 : cost of the return function
End
```

Best case:

In the optimal scenario, element x is situated at the initial position within the array, designated as A[0]. Consequently, $C_{min}(n)$ is equal to the sum of c1, c2, c4, and c5.

Worst case:

In the event of an unsuccessful outcome, it can be assumed that element x is either present in the final position of the array (x=A[n-1]) or is absent from the array entirely. Consequently, the loop will be executed n times, and $C_{max}(n)$ can be expressed as follows:

$$C_{max}(n) = c1+ (n+1)\ c2+ n\ c3+ c4+ (c5|c6)$$
$$\approx n\ (c2+c3) + c1+ c4+ (c5|c6)$$

Average case:

In the case of an average, let q be defined as $p(x \in A)$ and 1-q be defined as $p(x \notin A)$, where p is the probability. It should be noted that D^i_n (for $1 \le i \le n$) represents the set of data where x is in the i^{th} place and D^0_n represents the set of data where x is absent.

- $p(D^i_n) = q/n$
- $p(D^0_n) = 1- q$
- $C(D^i_n) = i\ (c2+c3) + c1+ c4+ c5$
- $C(D^0_n) = n\ (c2+c3) + c1+ c4+ c6$

In order to facilitate the calculation, it is proposed that $C(D^i_n) = i$ and $C(D^0_n) = n$.

Thus, $C_{avg}(n) = \sum_{i=0}^{n} p(D_n^i) * C(D_n^i)$

$$= (1-q)*n + \sum_{i=1}^{n} q/n * i$$
$$= (1-q)*n + (q/n)*n*(n+1)/2$$
$$= (1-q)*n + (q/2)*(n+1)$$

Let us assume that x is present in the array A. In this case, we have q=1, which allows us to conclude that $C_{avg}(n) = (n+1)/2$.

Alternatively, if we assume that x has a one-in-two chance of being present in the array, we have q=1/2. In this case, we can conclude that $C_{avg}(n)= ¼(3n+1)$.

Note:

It should be noted that the best-case and worst-case complexities represent the lower and upper bounds, respectively, on the algorithm's execution time.

The average complexity reflects the general behavior of the algorithm, but requires knowledge of the distribution or probability of the data. In practice, the focus is on worst-case complexity, which provides an upper time bound that the algorithm cannot exceed in all situations. Therefore, average complexity and extremal complexities are linked by the following relationship: $C_{min}(n) \leq C_{avg}(n) \leq C_{max}(n)$.

4. Asymptotic complexity

We may consider two algorithms, A and B, which are designed to solve the same problem. Each algorithm has a distinct complexity, with A having a complexity of $C_A = 100n$ and B having a complexity of $C_B = n^2$. This leads to the question of which of these algorithms is more efficient.

The ratio of the complexities of B to A is $C_B / C_A = n/100$.

This ratio allows us to conclude that, for:

- $n < 100$: B is more efficient.
- $n = 100$: A and B are equally efficient.
- $n > 100$: A is more efficient.

It should be noted that as n increases, A becomes more efficient than B.

Consequently, when the data size is relatively small, the majority of algorithms designed to solve the same problem exhibit similar performance characteristics. However, as the data size grows, the behavior of an algorithm's complexity becomes a crucial factor. This behavior is referred to as asymptotic complexity.

It is not always necessary to know the precise cost of an algorithm; a simple approximation of its complexity is sufficient to determine whether one algorithm is more efficient than another.

For two algorithms A_1 and A_2 with respective complexities $C_1(n)$ and $C_2(n)$, if the asymptotic order of growth of A_1 is strictly greater than that of A_2, we can immediately conclude that A_2 is better than A_1 for large values of n.

In mathematics, the comparison of asymptotic orders of growth is typically conducted using **Landau notation**, also known as "*Big O*" notation.

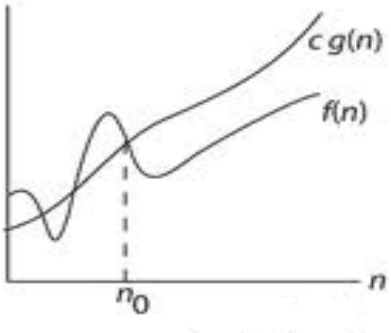
4.1. Landau notation (Big *O*)

Definition: Let *f* and *g* be two functions. We write *f(n) = O(g(n))* if there exists a positive constant *c* and an integer n_0 such that for all $n > n_0$, $f(n) \leq c.g(n)$.

Thus, *f(n) = O(g(n))* means that *f* is asymptotically dominated by *g*.

f(n) = O(g(n)) is read as "f is big O of g" or "f is in the order of g".

f(n) = O(g(n))



Example:

f(n) = 3n+1, g(n) = n

f(n) = O(g(n)) because for $n \geq n_0=1$ and *c=3+1=4,* we have *3n+1 ≤ 4n*

4.2. Rules for calculating asymptotic complexity

Firstly, it is necessary to establish the following properties:

- Reflexivity: $f = O(f)$
- Transitivity: If $f = O(g)$ and $g = O(h)$ Then $f = O(h)$
- Product by a constant: $O(\alpha f) = \alpha O(f) = O(f)$, $\alpha > 0$
- Sum and product of functions: $O(f)+O(g)=O(max\{f, g\})$

  $O(f).O(g)=O(f.g)$

The general rules below are applied for calculating complexity [6].

Rule 1: Elementary instructions

The execution time of an elementary instruction, such as assignment, read, write, and compare, is independent of the size of the input data. The complexity of an elementary instruction is ***O(1)***, indicating that the time required for its execution is constant.

Rule 2: Sequence of two modules

The complexity of a sequence of two modules M_1 with complexity $O(f(n))$ and M_2 with complexity $O(g(n))$ is equal to the greater complexity of the two modules:

O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(max(f(n), g(n)))

Rule 3: Conditional statements

The complexity of a conditional statement is equal to the sum of the complexity of the test and the maximum complexity of the modules involved.

If (<Cond> [$O(h(n))$]) **then** $M1$ [$O(f(n))$];

Else $M2$ [$O(g(n))$];

The complexity of this statement is equal to: ***O(h(n))+Max(O(f(n)),O(g(n)))***

Rule 4: Loops

The complexity of a loop is equivalent to the sum of the complexities of the loop body (including tests) over all iterations.

```
While (<Cond> [O(h(n))]) do
   P ;     [O(f(n))]          /* m times*/
EndWhile
```

The complexity of the loop can be described as follows:

$\sum^{m} Max(O(h(n)), O(f(n)))$ where m is the number of iterations.

Note:

The overall complexity of a code block with nested loops is obtained by multiplying the complexity of the inner module by the product of the sizes of all enclosing loops. For example, the complexity of the following block is expressed as $O(n^2)$.

```
For i from 1 to n do
        For j from 1 to n do
                k← k+1 ;
```

4.3. Complexity of recursive algorithms

The complexity of a recursive algorithm is determined by solving a recurrence equation. The following example illustrates this concept:

```
Function Fact(n : integer) : integer
Begin
        If (n <= 1) Then  Return 1 ;
        Else return  n * Fact(n - 1) ;
End
```

The complexity of this recursive function, *C(n),* is equal to the sum of :

- the complexity of the test ($n \leq 1$);

And the maximum of the complexities of the following statements:

- return 1;
- return n*Fact(n - 1);

Thus, we can write:

$$C(n) = \begin{cases} a & \text{if } n \leq 1 \\ b + C(n-1) & \text{otherwise} \end{cases}$$

In this context, the term "a" encompasses the following aspects:

- The complexity of the test,
- The complexity of the return function.

Similarly, the term "b" encompasses the following aspects:

- The complexity of the test,
- The complexity of the operation of multiplying n by the result of *Fact(n - 1)*,
- and the complexity of the return function.

The computational complexity of Fact(n-1) (denoted as *C(n-1)*) will be calculated recursively using the same decomposition. To solve this equation, the following procedure should be followed:

$$\begin{aligned} C(n) &= b + C(n-1) \\ &= b + [b + C(n-2)] \\ &= 2b + C(n-2) \\ &= 2b + [b + C(n-3)] \\ &= 3b + C(n-3) \\ &= \ldots \\ &= n\,b + C(n-n) \\ &= n\,b + C(0) = n\,b + a \end{aligned}$$

Thus, Fact()is in ***O(n)*** because b is a positive constant.

5. Complexity classes

Complexity classes group computational problems based on the execution time (or space) required to solve them as input size increases. These classes help categorize problems by their inherent time-based difficulty.

- ***f(n) = O(1)*** : Constant complexity
  The execution time is independent of data size.
- ***f(n) = O(log(n))*** : Logarithmic complexity
  Very small increase in execution time with increasing parameters.
  Example: Algorithms that decompose a problem into a set of smaller problems (dichotomy).
- ***f(n) = O(n)*** : Linear complexity
  Linear increase in execution time with increasing parameter (if parameter doubles, time doubles).
- ***f(n) = O(n log(n))*** : Quasi-linear complexity
  Slightly above O(n).
  Example: algorithms that decompose a problem into simpler, independently treated problems, and combine the partial solutions to compute the general solution.
- ***f(n) = O(n^2)*** : Quadratic complexity
  When the parameter doubles, execution time is multiplied by 4.
  Example: algorithms with two nested loops.
- ***f(n) = O(n^i)*** : Polynomial complexity
  If the parameter doubles, the execution time is multiplied by 2^i.
  Example: Algorithm with i nested loops.
- **f*(n) = O(2^n)*** : Exponential complexity
  When the parameter doubles, the execution time is raised to the power of 2.
  Example: Fibonacci recursive algorithm.
- **f*(n) = O(n!)*** : Factorial complexity
  The execution time increases factorially with the input parameter.
  Example: Set permutation algorithm

In algorithm analysis, efficiency is generally compared by worst-case execution time, with an algorithm considered more efficient if its time is an order of growth smaller. A problem's complexity is defined by its best possible algorithm. These core concepts help us evaluate algorithms and understand the inherent difficulty of problems.

Chapter III : Sorting algorithms

1. Introduction

Sorting is one of the classic algorithmic problems. In this chapter, we will try to formalize this problem as follows:

Let $E = \{e / e \in N\}$

Sorting the set E means rearranging its elements according to a well-defined order.

Example: Sort E in ascending order

> ***Input***: *an integer n, a set* $E = \{ e_1, e_2, ..., e_n \}$
>
> ***Processing***: *reorder input set*
>
> ***Output***: *a sorted set* $E = \{ e_1, e_2, ..., e_n \}$ *such that* $e_1 \leq e_2 \leq ... \leq e_n$

The simplest sort logic is:

- Compare each element with its neighbor and swap them, if necessary;
- Perform as many passes until all elements are sorted.

There are sorting algorithms with different methods and complexity. The question is, which algorithm is the most optimal?

Note:

> *In the following, we consider:*
>
> - *A basic data structure, namely an* ***array of integers****;*
> - *The sorting is done in* ***ascending order****.*

2. Common sorting algorithms

In this section, we examine several commonly used sorting algorithms [1][7]. We assume a basic data structure—an array of integers A—that we aim to sort in ascending order.

2.1. Selection sorting

Principle:

- Initially, the unsorted part is the entire array (A).
- The algorithm proceeds in passes (or iterations). In each pass, we find the smallest element in the current unsorted part of the array.
- If necessary (i.e., if the smallest element found is not already the first element of the unsorted part), the smallest element is swapped with the first element of the unsorted part.
- After each pass, the first element of the (previous) unsorted part becomes the last element of the sorted part, and the boundary between sorted and unsorted shifts one position to the right. The sorted part grows, and the unsorted part shrinks.

Algorithm:

```
Procedure Selection_Sort (A : Array of integers, n : integer)
Var   i, j, min_idx : integer;
Begin
        // Outer loop: iterates through the array to mark the boundary of the sorted portion
        For i from 0 to n-2
                min_idx ← i ; //Assume the current element is the min of the unsorted portion

                // Inner loop: find the min element index in the remaining unsorted portion
                For j from i+1 to n-1
                    If (A[j] < A[min_idx]) Then
                                    min_idx ← j;     // Found a new minimum
                    EndIf
                EndFor

                // If the min element found is not the element we started with (at index i)
                // swap it with the first element of the unsorted portion
                If (min_idx <> i) Then
                Swap(A[i], A[min_idx]) ;        // Swap is a separate permutation function
                EndIf
        EndFor
End
```

Complexity:

Let *C(n)* denote the cost of the algorithm in terms of number of operations (*n*: the size of the array).

$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

$= \sum_{i=0}^{n-2} (n - 1 - i)$

= (n-1) + (n-2) + … +1

= n(n-1)/2

➔ ***O(n²)***

2.2. Insertion sorting

Principle:

- Initially, the sorted part of the array consists only of the first element (A[0]). The unsorted part is the rest of the array (from A[1] onwards).
- At each iteration, we select the first element from the unsorted part. This element is called the "key".
- We compare the key with the elements in the already sorted part, moving backward from the right end of the sorted part.
- While comparing, elements in the sorted part that are greater than the key are shifted one position to the right to make space.
- Once the correct position for the key is found (where the element to its left is less than or equal to the key), the key is inserted into that position.

Algorithm:

```
Procedure Insertion_Sort(A: Array of integers, n : integer)
Var   i, j, key : integer;
Begin
        // The outer loop iterates from the second element to the last
        For i from 1 to n-1
            key ← A[i] ;       // Store the element to be inserted
            j ← i – 1 ;        // Initialize j to the last element of the sorted portion

            // While j is within the sorted portion and the key is smaller than A[j]
            While ((j >= 0) and (key < A[j])) do
                A[j + 1] ← A[j] ;       // Shift element A[j] one position to the right
                j ← j – 1 ;             // Move to the previous element in the sorted portion
            EndWhile

            A[j + 1] ← key ;    // Insert the key in its correct position in the sorted portion
        EndFor
End
```

Complexity:

- **Best case**: the array is sorted in the correct order (ascending in this case):

  $C_{min}(n) = \sum_{i=1}^{n-1} 1 = (n-1)$

  ➔ ***O(n)***

- **Worst case**: the array is sorted in reverse order (descending in this case):

  $C_{max}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$

  ➔ ***O(n^2)***

2.3. Bubbles sorting

Principle:

- Initially, the entire array is considered unsorted. A sorted portion will grow from the end of the array.
- The algorithm consists of repeatedly stepping through the unsorted portion of the array. This is often called a "pass". During each pass, it compares each pair of adjacent elements in the unsorted portion (e.g., the first with the second, then the second with the third, and so on).
- If a pair of adjacent elements is found to be out of order (e.g., the element on the left is greater than the element on the right in an ascending sort), those two elements are swapped.
- After a complete pass through the unsorted portion, the largest (or smallest) element among the unsorted ones will have "bubbled up" to its correct final position at the end of the currently unsorted section of the array.
- The size of the unsorted portion decreases by one element after each pass. The process is repeated until the entire array is sorted (which can be detected when a pass completes without any swaps).

Algorithm:

```
Procedure Bubble_Sort(A : Array of reals, n : integer)
Var    i, j : integer;
Begin
    // Outer loop: Controls the number of passes.
    For i from n - 1 down to 1
```

```
    // Inner loop: Compares adjacent elements and swaps them if they are out of order
            For j from 0 to i-1
                    If (A[j] < A[j + 1]) Then
                            Swap(A[j], A[j + 1]); // Assumes a separate Swap procedure exists
                    EndIf
            EndFor
    EndFor
End
```

Complexity:

$C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$ ➔ $\boldsymbol{O(n^2)}$

2.4. Merge sorting

Principle:

This is a divide-and-conquer algorithm. It works as follows:

- **Divide:** The array is recursively divided into two (nearly) equal sub-arrays until each sub-array contains only one element (which is considered sorted).
- **Conquer:** The algorithm makes two recursive calls to sort the two sub-arrays.
- **Combine:** The two sorted sub-arrays are then merged back together into a single sorted array using a specific merging procedure.

Algorithm:

```
// Procedure to divide an array A from index st to en
Procedure Merge_Sort (A: Array of integers, st, en: integer)
Var   mid: integer;
Begin
    If (st < en) Then
            mid ← (st + en) div 2;          // The middle point to divide the array
            // Sort left and right halves recursively
            Merge_Sort (A, st, mid);
            Merge_Sort (A, mid + 1, en);
            // Merge the two sorted halves
            Merge (A, st, mid, en);
    EndIf
End
```

```
// Procedure to merge two sorted sub-arrays of A
Procedure Merge (A: Array of integers, st, mid, en: integer)
Var  i, j, k : Integer;
     A_temp : array of integers;       // A temporary array to hold the merged result
Begin
     // Initialize indices for the left sub-array, right sub-array, and temporary array
     i ← st ;
     j ←  mid + 1 ;
     k ←  st ;

     // Merge elements from the two sub-arrays into A_temp
     While (i <= mid) And (j <= en) Do
          If (A[i] <= A[j]) Then
               A_temp[k] ←  A[i]; i++;
     Else      A_temp[k] ← A[j]; j++;
     EndIf
          k++;
     EndWhile
     // Copy any remaining elements from the left sub-array (if any)
     While (i <= mid) Do
          A_temp[k] ← A[i]; i++; k++;
     EndWhile
     // Copy any remaining elements from the right sub-array (if any)
     While (j <= en) Do
          A_temp[k] ← A[j]; j++; k++;
     EndWhile
     // Copy the merged elements from A_temp back into the original array A
     // The elements in A_temp from index st to en are now sorted.
     For k from st to en
          A[k] ← A_temp[k];
     EndFor
End
```

Complexity:

$$C(n) = \begin{cases} 1, & if\ n \leq 1 \\ 2\ C\left(\frac{n}{2}\right) + n, & otherwise \end{cases}$$

$C(n) = 2\ C\left(\frac{n}{2}\right) + n$ (n is the merging cost)

$= 2\ (2\ C\left(\frac{n}{4}\right) + \frac{n}{2}) + n$

$= 4\ C\left(\frac{n}{4}\right) + 2\ n$

$= \ldots$

$= 2^k\, C\left(\frac{n}{2^k}\right) + k\, n$

$= n\, C(1) + \log_2 n \,.\, n \quad (\text{for } n = 2^k)$

➔ $C(n) = O(n \log_2(n))$

2.5. Quick sorting

Principle:

This is also a divide-and-conquer algorithm that works as follows:

i. Select any element from the array, referred to as the **pivot**.

ii. Partition the array into two parts such that:

- Elements in the first part are less than or equal to the pivot.
- Elements in the second part are greater than or equal to the pivot.

iii. Make two recursive calls to sort the two parts of the array in the same way.

Algorithm:

```
Procedure Quick_Sort (A: Array of integers, st, en: integer)
Var    p_index: integer; // Index of the pivot
Begin
        If (st < en) Then
                // Partition the sub-array and get the pivot's final index
                p_index ← Partition(A, st, en);

                // Recursively sort subarrays left and right of pivot
                Quick_Sort (A, st, p_index - 1);
                Quick_Sort (A, p_index + 1, en);
        EndIf
End

// Function to partition a sub-array A[st..en] around a pivot (A[en])
Function Partition (A: Array of integers, st, en: integer): integer
Var    i, p_ind, pivot_value: integer;
Begin
        pivot_value ← A[en];    // Select the last element as the pivot value
        p_ind ← st;        //This indice will track the boundary of the part <= pivot
```

```
        // Iterate through the array from the start up to the element before the pivot
        For i from st to en - 1
            If (A[i] <= pivot_value) Then
                Swap (A[i], A[p_ind]);
                p_ind ++;        // Increment p_ind to extend the left section
            EndIf
        EndFor

        // Now, p_ind is the correct position for the pivot
        // Swap the original pivot element (which is still A[en]) with the element at A[p_ind]
        Swap (A[en], A[p_ind]);
        Return (p_ind); // Return the final index of the pivot
End
```

Complexity:

Best-case (balanced partition) :

$C(n) = n + 2\ C(\frac{n}{2})$ ➔ ***O(n.log (n))***

Worst case (e.g. already sorted array) :

$C(n) = n + C(0) + C(n-1)$ ➔ $\boldsymbol{O(n^2)}$

2.6. Binary tree sorting

Principle:

- Given an array of integers, we construct a **binary search tree** (**BST**)[2] by inserting all values from the input array into the tree, one by one.
- The following property must be satisfied during construction: For any given node x:
  - All values in its left subtree must be less than the value of x.
  - All values in its right subtree must be greater than the value of x.
  - (Handling of duplicate values varies; they are typically placed in the right subtree or handled with a counter in the node).
- Perform an in-order (infix) traversal of the constructed BST and copy the elements back into the original array (or a new sorted array).

[2] Please refer to Chapter IV : Tree structures for more details.

Generic algorithm:

```
Procedure BST_Sort (Tree: Pointer to Node; st, en : integer; A: Array of integers)
Var i : integer, p : Pointer to Node;
Begin
        For i from st to en
                Allocate(p);      // Create new node
                Affect_value(p, A[i]);    // Assign value of A[i] to the new node
                Insert_BST(Tree, p);  // Construct the tree
        EndFor
        // Procedure for In-order traversal that copies elements back to the array A
        Infix_copy (Tree);
End
```

Complexity:

Best-case (balanced tree): $C(n) = \mathbf{O(n.log\ (n))}$

Worst case: $C(n) = \mathbf{O(n^2)}$

These complexities reflect the cost of inserting n elements into a BST. In the best-case scenario, the tree remains balanced, and each insertion takes O(log n), resulting in a total of O(n log n). In the worst case, if the elements are inserted in sorted order, the tree becomes a linear chain and each insertion takes O(n). This results in an overall time complexity of $O(n^2)$.

3. Conclusion

An overview of the costs for different sorting algorithms is presented in the table below.

Table III-1: Complexity of sorting algorithms.

Algorithm	**Best-case complexity** ($C_{min}(n)$)	**Worst-case Complexity** ($C_{max}(n)$)
By selection	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$
By insertion	$\mathbf{O(n)}$	$\mathbf{O(n^2)}$
Bubbles	$\mathbf{O(n^2)}$	$\mathbf{O(n^2)}$
By merging	$\mathbf{O(n \log_2 (n))}$	$\mathbf{O(n \log_2 (n))}$
Quick	$\mathbf{O(n \log_2 (n))}$	$\mathbf{O(n^2)}$
By BST ((Tree Sort)	$\mathbf{O(n \log_2 (n))}$	$\mathbf{O(n^2)}$

In essence, although Insertion Sort is efficient for particular inputs (like nearly sorted data) and Quick Sort/BST Sort are often fast in practice, algorithms guaranteeing O(n log n) performance even in the worst case (like Merge Sort) are more reliable and scalable for large, unsorted datasets than algorithms with $O(n^2)$ complexity. Selecting the right algorithm depends on factors such as data volume, the potential for the worst case, and the need for stable sorting.

Chapter IV : Tree structures

1. Introduction

In linear structures, such as arrays and linked lists, the following features are observed [6]:

1. Direct access to array elements (by index) is possible, resulting in faster retrieval. However, insertion and deletion require an offset.

2. In linked lists, insertion and deletion are only feasible by modifying the chaining. Nevertheless, access to elements (links) is sequential, resulting in slower retrieval.

Trees represent a compromise between the two previous structures, offering a balance between the two aforementioned characteristics:

1. Relatively fast access to an element (node) from its key.

2. The addition and deletion operations are not costly.

This chapter introduces the concept of tree-like structures, followed by an exploration of different types of trees, with a particular emphasis on Binary Search Trees (BSTs) and Heaps.

2. N-ary trees

2.1. Definition

In algorithmics, a tree is defined as an abstract data structure represented by a cycle-free graph, wherein each node is capable of having between zero and several ascendants and/or descendants. This concept is exemplified in Figure IV.1 below.

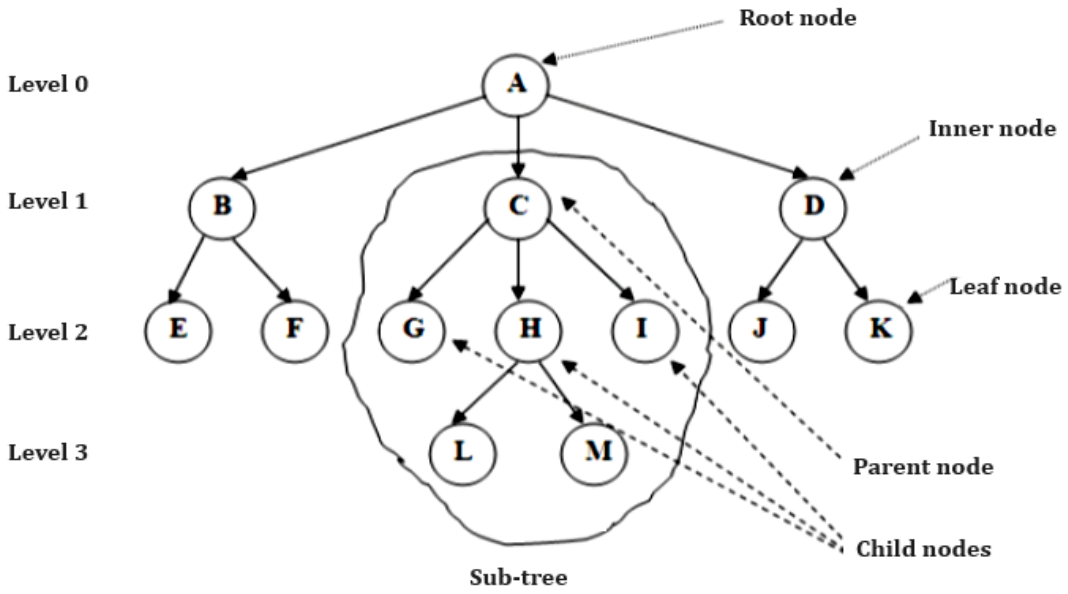


Figure IV.1 : N-ary tree [6].

From the example shown in the figure above, we can define the following terms:

- A node: a tree element with a label.
- A label: the value or information associated with a node (e.g., "A", ...).
- Parent node: a direct ascendant of a node (e.g., Parent(C) = A, Parent(L) = H, Parent (A) = {}).
- Child nodes: direct descendants of a node (eg. Child(A) = { B,C,D }, Child(H) = { L,M }, Child(M) = {}).
- The descendants of a node (e.g. Desc(C)={G,H,I,L,M}, Desc(B)={E,F},...).
- The ascendants of a node (e.g. Asc(L)= {H,C,A}, Asc(E)= {B,A},...).
- The root: a single node that has no parent (e.g. A).
- The leaf: a node that has no children (e.g. E, F, G, L, M, I, ...).

Note :

The name of a tree is usually associated with the name (label) of its root. For example, Tree A in Figure IV.1.

2.2. Tree characteristics

Each tree is associated with certain characteristics [1][6]:

- **Size of a tree**: The size of a tree is defined as the total number of nodes it contains. For example, the size of tree A is 13, while the size of an empty tree is 0.
- **Level of a node**: The level of a node is defined as follows: the level of the root is equal to 0, while the level of any node (other than the root) is equal to the level of its parent plus 1. For example, the level of E, F, G, H, I, J, or K is equal to 2.
- **Depth (height) of a tree**: This is the maximum level within the tree structure. For illustrative purposes, the depth of tree A is equal to 3.
- **Degree of a node**: This is defined as the number of its child. For example, the degree of the nodes A, B, C, E, and H is 3, 2, 3, 0, and 2, respectively.
- **Degree of a tree**: is defined as the maximum of the degrees of its nodes. To illustrate, the degree of tree A is equal to 3.

2.3. Use of trees

Trees are commonly used to represent or manipulate various types of data in a wide range of applications, such as file systems, databases (for indexing), domain name servers (DNS), and more. Another case is the representation of arithmetic expressions.

Example:

The expression **(2 * 5) + (6/7)** is represented by the following tree:

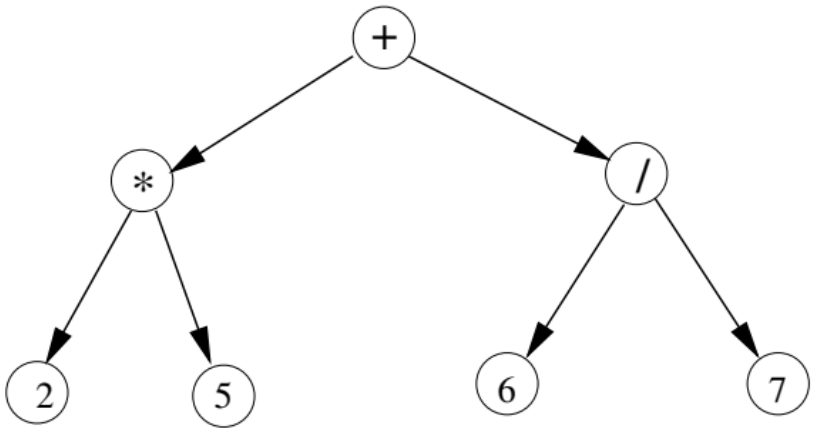


Figure IV.2: Tree representation of an arithmetic expression [8].

2.4. Trees implementation

Trees can be implemented either statically or dynamically. In what follows, we focus on the dynamic representation, in which a Node structure can be defined as follows:

```
Type Node = Structure
        Info : any ;        // Node content (can be of any type)
        NbChild: integer;          // Number of childs
        Child : Array[1..NbChild] of Pointer to Node ;
EndStructure
```

The tree is declared as follows:

```
Var Root : Pointer to Node ;
```

Example:

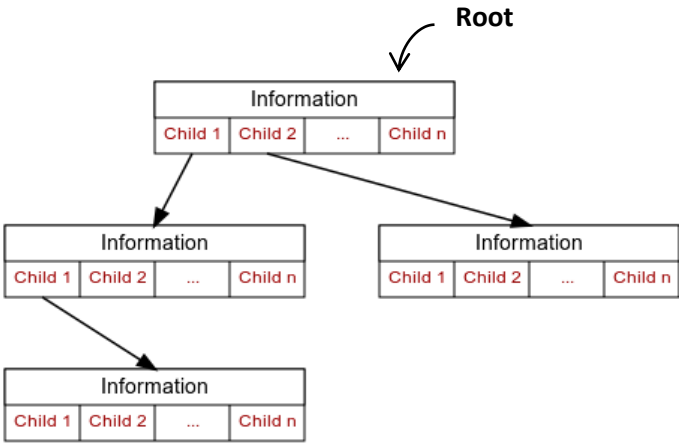


Figure IV.3: Dynamic representation of an n-ary tree [6].

2.5. Operations on trees

To manipulate trees, the following primitives are defined:

- **Allocate (N) :** create a structure of type Node and return its address in N.
- **Free(N) :** free the area pointed to by N.
- **Affect_Val(N, inf) :** assign the value of "inf" in the Info field of the node pointed to by N.
- **Affect_Child (i,N1,N2) :** make N2 child number i of N1.
- **Child(i,N) :** returns child number i of N.
- **Value(N) :** returns the contents of the Info field of the node pointed to by N.
- **Affect_NbChild(N,nb) :** assign the value of “nb” in the NbChild field of the node pointed to by N.
- **Nbr_Child(N) :** returns the number of children in N.

2.6. Tree traversal

Tree traversal is an algorithmic process used to visit all the nodes (or subtrees) of a tree structure. Two main types of traversal are generally distinguished:

- Depth-first traversal
- Breadth-first traversal

2.6.1. Depth-first traversal

A traversal is said to be **depth-first** when the traversal of one subtree is completed **before** the traversal of another begins (see Figure IV.4).

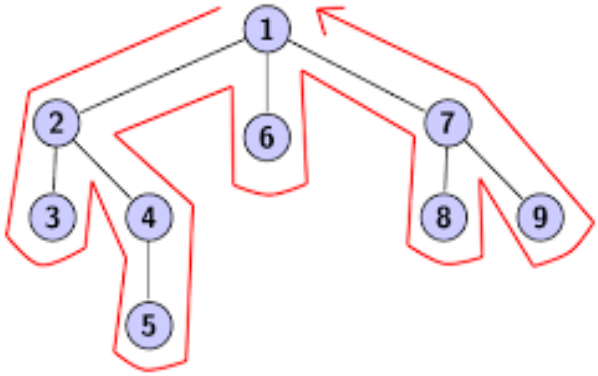


Figure IV.4: Depth-first traversal of a tree.

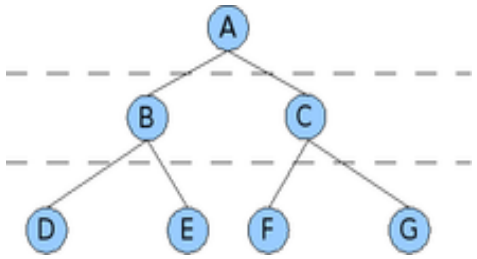
The corresponding algorithm is as follows:

```
Procedure DFT( Node : Pointer to Node)
Begin
        If (Node ≠ Null) Then
                For i from 0 to Nbr_Child(Node)-1
                        DFT (Child(i,Node));
                EndFor
        EndIf
End
```

A depth-first traversal of *n*-ary trees can be categorized as follows:

- **Prefix (pre-order)**: where the current node is processed **before** its descendants.
- **Postfix (post-order)**: where the current node is processed **after** its descendants.

For example, in the tree shown here:



The prefix depth-first traversal yields: **A, B, D, E, C, F,** then **G.**

The postfix depth-first traversal yields: **D, E, B, F, G, C,** then **A**.

2.6.2. Breadth-first traversal

In this traversal, all the nodes at level **i** must be processed **before** the first node at level **i+1** is processed (see Figure IV.5).

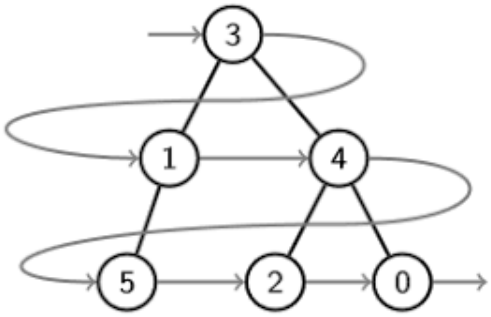


Figure IV.5: Breadth-first traversal of a tree.

Such a traversal requires the storage of all branches (i.e. child nodes) still to be visited. A **queue** is then used to achieve this.

The corresponding algorithm is as follows:

```
Procedure BFT( Node : Pointer to Node)
Var N : Pointer to Node, F : Pointer to ListNode;  // F is a queue
Begin
        If (Node ≠ Null) Then
                InitFile() ;        // Initialize the queue F
                Enqueue(F,Node) ;       // Enqueue the node into F
                While (F≠ Null) do
                        Dequeue(F,N) ;  // Dequeue the head of F into N
                        Write (Value(N)) ;
                        For i from 0 to Nbr_Child(N)-1
                                If (Child (i,N) ≠ Null) Then        Enqueue(F,Child(i,N)) ;
                                EndIF
                        EndFor
                EndWhile
        EndIf
End
```

2.7. Some types of trees

In this section, we present some commonly used types of tree structure. Each type has its own properties and applications, particularly for data organization, searching and efficient storage [1].

- ✓ **N-ary Tree**: As described earlier, an *n*-ary tree is a tree in which the maximum degree of a node is equal to *n*. This means that a node can have up to *n* children, providing a flexible structure for various applications.
- ✓ **B-Tree**: A B-tree of order *m* is a self-balancing tree commonly used in database management systems and file systems. Its main characteristics are:
    - The root has at least two children.
    - Every node, except the root, has between *m/2* and *m* children.
    - All leaf nodes are at the same level, ensuring efficient data retrieval.
- ✓ **Binary Tree**: A binary tree is a structure in which each node has at most two children, called the "left child" and "right child". This property makes it a fundamental structure in search algorithms, indexing, and optimization.
- ✓ **Complete Binary Tree**: A binary tree is complete when all its levels, except possibly the last, are fully filled. The last level must be aligned to the left, ensuring a dense distribution of nodes.
- ✓ **Perfect Binary Tree**: A binary tree is perfect when every internal node has exactly two children, and all the leaves are at the same level. This allows for optimized searches and ensures uniformity in the tree structure.
- ✓ **Balanced Binary Tree**: A binary tree is balanced when the height of the left and right subtrees of every node differs by no more than one. This property ensures optimal performance for search and insertion operations, minimizing the risk of performance degradation due to excessively deep trees.

2.8. Converting an N-ary tree to a Binary tree

The conversion of an *n*-ary tree into a binary tree follows these rules:

a. The root of the binary tree is the same as the root of the *n*-ary tree.

b. The left child of a node in the *n*-ary tree becomes the left child of that node in the binary tree.

c. The right sibling of any node in the *n*-ary tree becomes the right child of that node in the binary tree.

Example:

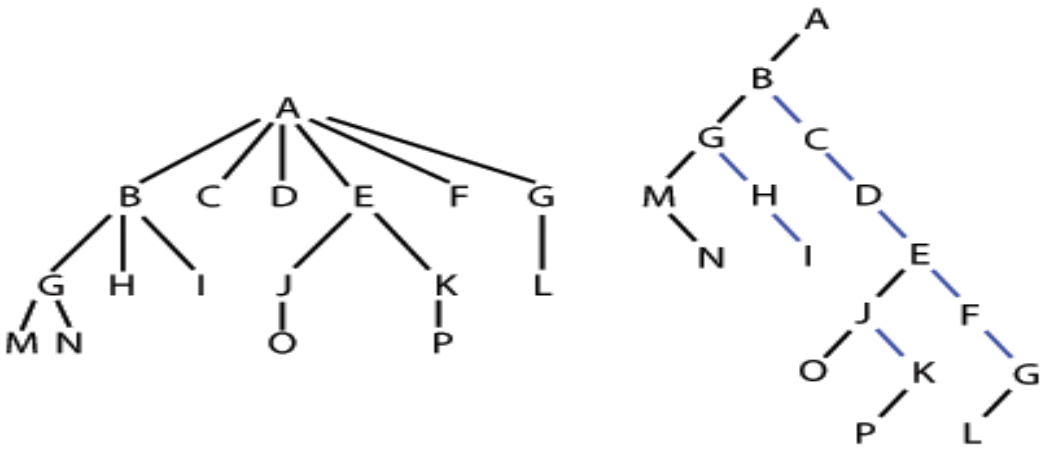


Figure IV.6: Conversion of an n-ary tree (left) to tis binary tree representation (right).

3. Binary trees

In this section, we introduce two important variants of binary trees: **Binary Search Trees (BST)**, which facilitate fast lookup, insertion, and deletion operations, and **Heaps**, which are used primarily in priority queue implementations and efficient sorting algorithms.

3.1. Binary Search Trees (BST)

3.1.1. Definition

A BST is a binary tree that satisfies the following property [9]:

Let x and y be two nodes in the BST:

- If y is a node in the left subtree of x, then key(y) < key(x)
- If y is a node in the right subtree of x, then key(y) > key(x)

Each node has at most one left child and one right child.

Example:

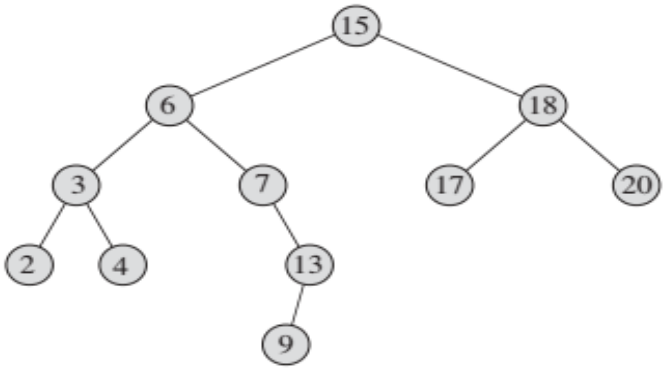


Figure IV.7: Example of a binary search tree [9].

3.1.2. Implementation

BSTs are implemented similarly to *n*-ary trees, using either a static or dynamic representation. In the last, each node is typically represented as a structure or object comprising at least three fields:

```
Type Node = Structure
    key : integer;              // Node key
    Value : any ;               // Node content or value can be of any type (optional)
    rChild, lChild :  Pointer to Node ;        // pointers to the left and right childs
EndStructure

    Var Root : Pointer to Node ;
```

This approach allows for flexible and memory-efficient tree construction, as nodes are created and linked dynamically during program execution.

3.1.3. Operations on BST

To manipulate a BST, the following primitives are defined:

- **Allocate(N)**: create a structure of type Node and return its address in N.
- **Free(N)**: free the memory allocated to the structure pointed to by N.
- **Value(N)**: returns the content of the value field of the node pointed to by N.
- **Key(N)**: returns the content of the key field of the node pointed to by N.
- **LC(N)**: returns the left child of the node N.
- **RC(N)**: returns the right child of the node N.
- **Fath(N)**: Returns the parent of the node N.
- **Affect_Val(N, v)**: assign the value v to the value field of the node N.
- **Affect_Key(N, k)**: assign the value k to the key field of the node N.
- **Affect_LC(N1, N2)**: make N2 the left child of N1.
- **Affect_RC(N1, N2)**: make N2 the right child of N1.

3.1.4. Algorithms on BST

3.1.4.1. Traversal

The traversal (or search) in a BST can be done either in depth or in breadth first.

- **Depth-first Search**

Algorithm:

```
Procedure DFS (Node : Pointer to Node)
Begin
        If (Node ≠ Null) Then
                DFS(LC(Node)) ;
                DFS(RC(Node)) ;
        EndIf
End
```

This algorithm can be [10]:

- **Prefix (preorder)**: Parent, Left Child, Right Child
- **Postfix (postorder)**: Left Child, Right Child, Parent
- **Infix (inorder)**: Left Child, Parent, Right Child

- **Breadth-first Search**

Algorithm:

```
Procedure BFS (Node : Pointer to Node)
Var N : Pointer to Node, F : Pointer to ListNode;  // F is a queue
Begin
    If (Node ≠ Null) Then
        InitFile(F) ;
        Enqueue(F, Node) ;
        While (F ≠ Null) do
            Dequeue(F,N) ;
            … ;            // processing
            If (LC(N) ≠ Null) Then
                Enqueue(F,LC(N)) ;
            EndIf
            If (RC(N) ≠ Null) Then
                Enqueue(F,RC(N)) ;
            EndIf
        EndWhile
    EndIf
End
```

3.1.4.2. Recursive Search

Recursive search in BST is a method where the search process is repeated on subtrees, breaking the problem into smaller instances. This technique exploits the recursive nature of the tree structure, allowing efficient searching by comparing nodes and moving to either the left or right subtree.

Algorithm:

```
Function Research (Node : Pointer to Node, k : Integer) : Pointer to Node
Begin
        If (Node = Null OR Key(Node) = k) Then Return Node ;
        Else If (k < Key(Node)) Then Return Research (LC(Node), k) ;
            Else Return Research (RC(Node), k) ;
            EndIf
        EndIf
End
```

The recursive search algorithm above is used to find a particular node in a BST. The algorithm recursively traverses the tree to find a node with a key that matches the value we are looking for and returns that node.

3.1.4.3. Insertion

Typically, insertion is done at the leaves of the BST.

The insertion algorithm is executed after the search for the key of the element to be inserted:

- If the element already exists, no action is taken.
- Otherwise, the search stops at an empty subtree, which is then replaced by the element to be inserted.

Algorithm:

```
Procedure Insert (Node : Pointer to Node , N: Pointer to Node)
// where: Node is the root and N is the node to insert
Begin
        If (Node = Null)  Then
                Node ← N ;
        Else If (Key(N) < Key(Node)) Then
                If (LC(Node) ≠ Null) Then
                        Insert(LC(Node),N) ;
                Else Affect_LC(Node,N) ;
                EndIf
            Else If (Key(N) > Key(Node)) Then
```

```
                    If (RC(Node) ≠ Null) Then
                        Insert(RC(Node),N) ;
                    Else Affect_RC(Node,N);
                    EndIf
                EndIf
            EndIf
        EndIf
    End
```

3.1.4.4. Deletion

Deleting a node in a BST requires considering several cases, depending on the structure and position of the node to be deleted.

These cases are:

- Node with no children (leaf): Simply remove it directly.
- Node with one child: Replace the node with its child.
- Node with two children: Replace the node with its successor (or predecessor).

Below are the different cases to consider when deleting a node N.

Table IV-1: Deletion cases.

Case				Action
	Node N to delete	**LC(N)**	**RC(N)**	
1	Leaf node	Null	Null	**Replace N with Null**
2	Internal node or root	≠ Null	Null	**Replace N with LC(N)**
3		Null	≠ Null	**Replace N with RC(N)**
4		≠ Null	≠ Null	**Replace N with S (or P), where:** **S: successor** **P: predecessor**

Before performing the deletion, it is important to define the notions of a node's **predecessor** and **successor**.

- **Predecessor of a node**

The predecessor of a node is the node with the greatest key that is smaller than the key of the given node. In a BST, it corresponds to the **maximum value** node in its **left subtree**.

For example, in Figure IV.7, the predecessors of nodes 3, 6, and 17 are respectively: 2, 4, and 15.

- **Successor of a node**

The successor of a node is the node that has the smallest key greater than the key of the given node. In a BST, it corresponds to the **minimum value** node in its **right subtree**.

For example, in Figure IV.7, the successors of nodes 3, 7, and 13 are respectively: 4, 9, and 15.

Note:

> *In what follows, we will assume that the node to be deleted is replaced by its* ***successor****.*

Therefore, the fourth case in Table IV-1 requires the following steps: Find the successor S of N, replace N with S, and delete S.

So, how can we locate the successor of a node?

The successor of a node *x* in a BST is defined as follows:

- If x has a right subtree, then its successor is the node with the **smallest key in that right subtree** (i.e., the **minimum of the right subtree**).
- Otherwise, the successor is the **closest ancestor of x** for which x lies in the left subtree; in other words, it is the **first ancestor whose left child is also an ancestor of x or x itself**.

Based on this definition, the general algorithm for deleting a node N with key k can be described as follows:

1. **Searching for the node to delete**:
    - Start at the root of the BST by comparing the key k with the root's key.
    - If the key k is different, move left or right depending on the comparison result.
    - Repeat this comparison until the target node is found or until a Null reference is reached.
2. **Deleting the node (if found)** according to the cases listed in the previous table (Table IV-1).

3.2. Heaps (Priority Queues)

3.2.1. Introduction

To implement a priority queue, several approaches can be considered:

- A regular queue (FIFO) allows fast insertion (at the end) in *O(1),* however, removing the highest-priority element requires a full search, resulting in *O(n)* time complexity.
- A sorted list allows for fast removal (of the first element) in *O(1)*, but insertion becomes costly, requiring *O(n)*.

In such cases, heaps offer a more efficient and balanced solution.

3.2.2. Definition

A heap is a tree that satisfies two fundamental properties:

1. It is a complete binary tree: all levels are filled, except possibly the last one, where elements are arranged as far left as possible.
2. It can be :
    - **Max heap**: the key of each node is greater than the keys of its descendants.
    - **Min heap**: the key of each node is smaller than the keys of its descendants.

Example:

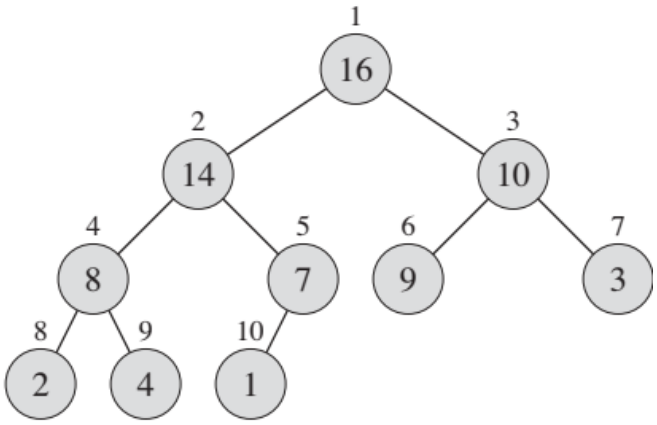


Figure IV.8: Example of a Max heap [2].

Note:

- *In a max heap, the element with the maximum key (the highest priority) is always at the root.*
- *The rest of this course will focus on this type (Max heap).*

3.2.3. Heap implementation

Heaps can be implemented in the same way as BST (Binary Search Trees), either statically or dynamically.

Dynamic representation:

```
Type Node = Structure
        key : integer;
        Value : any ;
        LC, RC, Fath:  Pointer to Node ;
EndStructure

Var Root : Pointer to Node ;
```

3.2.4. Operations on Heaps

3.2.4.1. Insertion

To insert a new element into the heap, we must:

1. Create a node containing the key of this element,
2. Attach this node to the last level in the first available empty spot as far left as possible (create a new level if necessary). This results in a complete binary tree, but not necessarily a heap.
3. Compare the new node's key with its parent's key and swap them if necessary; repeat the process until no more swaps are necessary.

Example:

Suppose we want to insert the element with key 10 into the heap shown in Figure IV.9. The steps of the insertion process are illustrated below in the same figure.

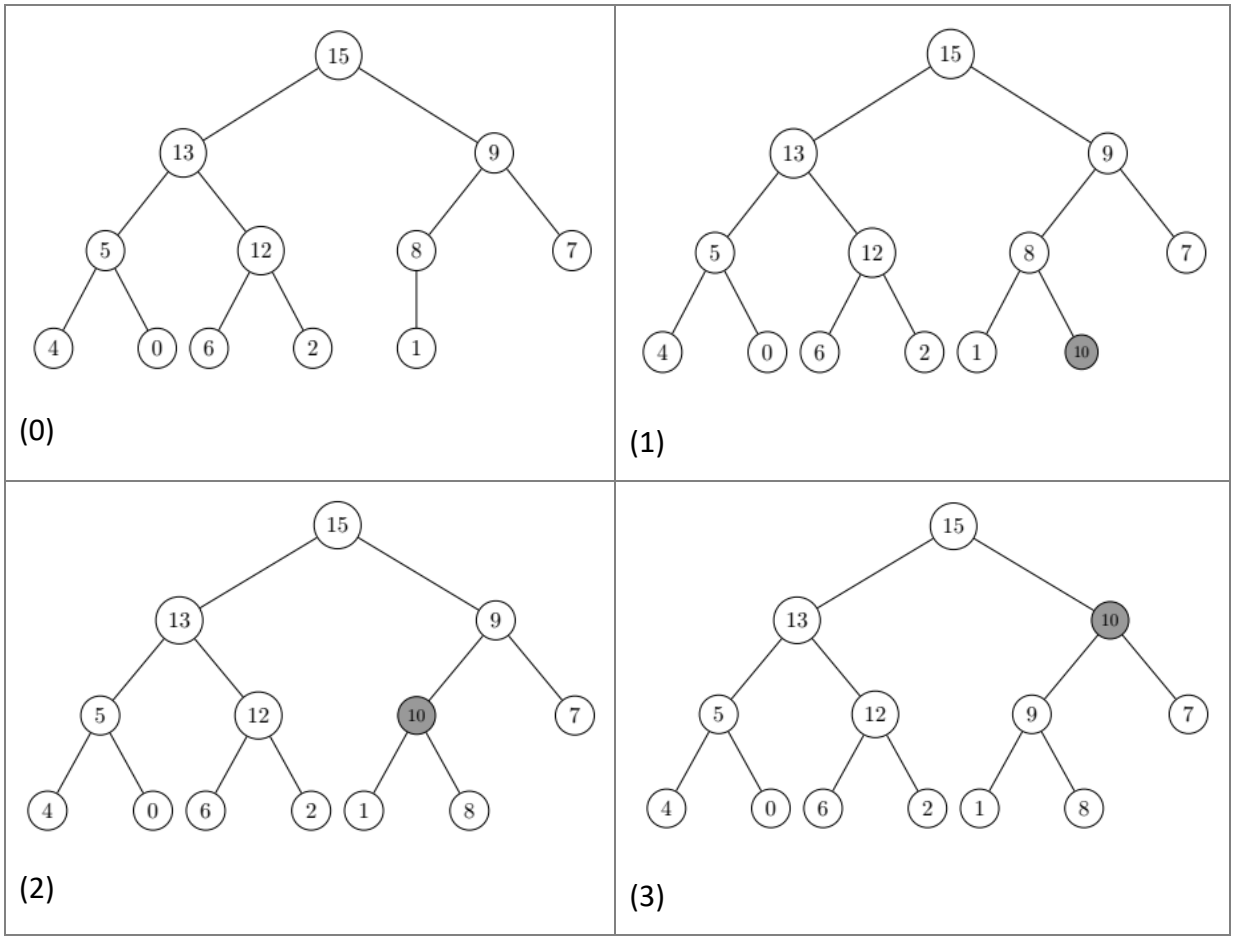


Figure IV.9: Example of element insertion in a Max heap [2].

Algorithm:

```
Function Insert(Root, N : Pointer to Node) : Pointer to Node
Var  P : Pointer to Node;
Begin
        If (Root = Null) Then  Return N ;
        Else
                P←PInser(Root) ; // PInser is the function returning the insertion point
                If (LC(P) = Null) Then
                        Affect_LC(P,N) ;
                Else  Affect_RC(P,N) ;
                EndIf
                While (Key(N) > Key(Fath(N) And Fath(N) ≠ Null) do
                        Swap(N, Fath(N)) ;  // Swap N with its father
                        N ← Fath(N) ;
                EndWhile
                Return Root ;
        EndIf
End
```

In what follows, we present the algorithm of the *PInsert()* function, which is a breadth-first traversal function that returns the insertion point.

```
Function PInser(Node: Pointer to Node) : Pointer to Node
Var N : Pointer to Node; F: Pointer to ListNode; // F: a queue
Begin
        If (Node ≠ Null) Then
                InitFile(F) ;
                Enqueue(F, Node) ;
                While (F ≠ Null) do
                        Dequeue(F, N);
                        If (LC(N) = Null Or RC(N) = Null) Then Return N ;
                        Else
                                Enqueue(F, LC(N) ) ;
                                Enqueue(F, RC(N) ) ;
                        EndIf
                EndWhile
        EndIf
End
```

3.2.4.2. Deletion (Removal)

In the max heap, the highest priority element is always at the root, so deleting consists of removing the root. To do this, we have to:

1) **Replace the root** with the last node (the rightmost node on the last level).
2) **Delete the last node** from the heap. As a result, we obtain a binary tree that is not necessarily a heap.
3) **Compare** the key of the new root with those of its children, and swap it with the larger child if necessary. **Repeat** this process until no further swaps are required.

Example: Suppose we need to perform a deletion on the max heap from Figure IV.8; the steps are illustrated in the same figure.

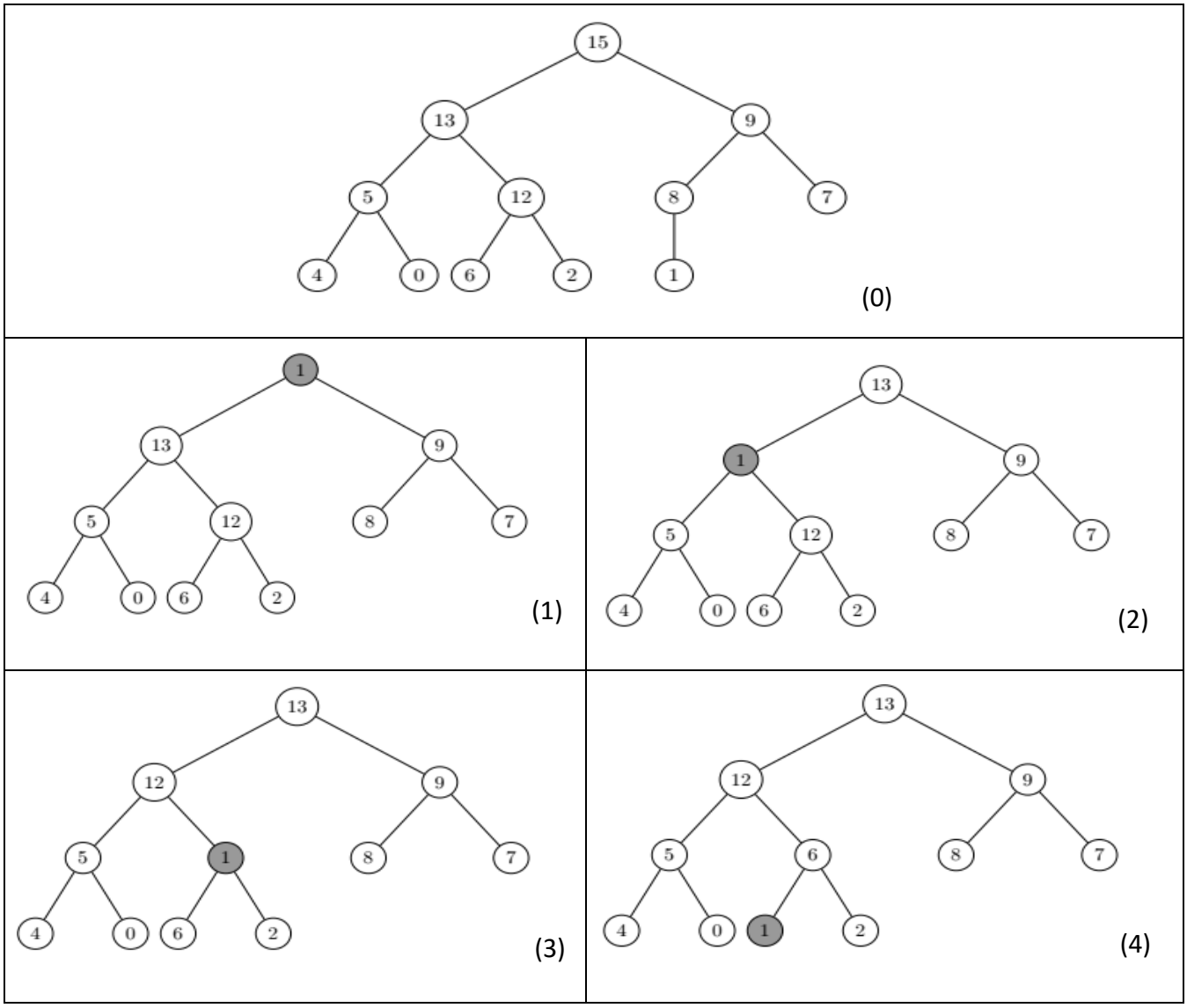


Figure IV.10: Example of deletion in a Max heap [2].

Algorithm:

```
Function Remove(Root: Pointer to Node) : Pointer to Node
Var P,T,M : Pointer to Node;
Begin
        P ← LastNod(Root) ;      //LastNod : is the function returning the last node
        Affect_Key (Root, Key(P)) ;
        If (P = LC(Fath(P))) Then
                Affect _LC(Fath(P),Null) ;
        Else   Affect_RC(Fath(P),Null) ;
        EndIf
        Free (P) ;          //remove the last node
        T ← Root  ;
        While (LC(T) ≠ Null Or RC(T) ≠ Null) do
                M ← Max(LC(T)), RC(T) ) ;          //Max returns the node with the maximum key
```

```
                If (Key(T) < Key(M)) Then
                        Swap (T,M) ;
                        T ← M ;
                Else Return Root  ;
                EndIf
        Endwhile
        Return Root  ;
End
```

The following function returns the last node inserted into the heap.

```
Function LastNod(Node: Pointer to Node) : Pointer to Node
Var N : Pointer to Node; F: Pointer to ListNode; // F: a queue
Begin
        If (Node ≠ Null) Then
                InitFile(F) ;
                Enqueue(F, Node) ;
                While (F ≠ Null) do
                        Dequeue(F, N);
                        If (LC(N) ≠ Null) Then Enqueue(F, LC(N) ) ; EndIf
                        If (RC(N) ≠ Null) Then Enqueue(F, RC(N) ) ; EndIf
                EndWhile
                Return N ;
        EndIf
End
```

Note:

Heaps can also be implemented statically. This approach is highly efficient and commonly used in practice.

3.2.5. Static implementation of heaps

This involves storing the heap elements in an array following a breadth-first order.

For explanation purposes, let's take the example of the heap in the following figure.

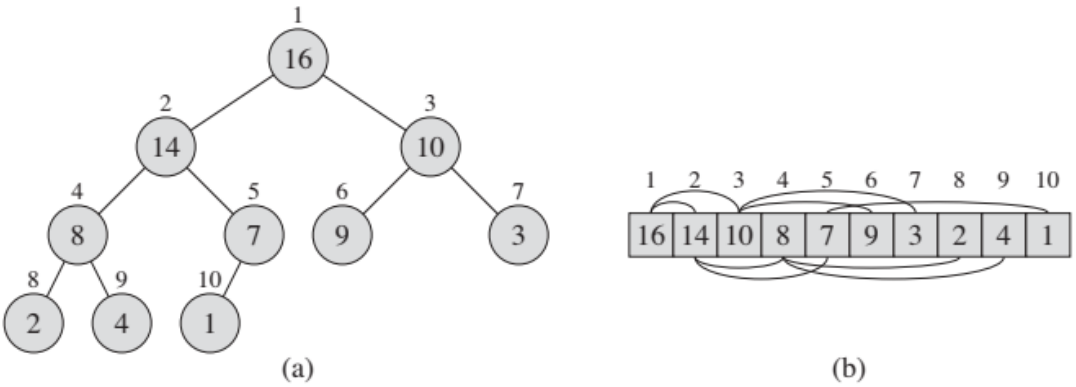


Figure IV.11: A Max heap represented as a binary tree (a) and an array (b) [2].

The numbers above the nodes correspond to the indexes in the array. In this case, the index of the first element is 1. The links above and below the array indicate parent-child relationships, with parents always to the left of their children.

In the resulting array, it can be seen that:

- The root is at position 1.
- The left child of an element with index i, if it exists, is always at position (2i).
- The right child of an element with index i, if it exists, is always at position (2i+1).
- The parent of an element with index i (where i > 1) is at position (i div 2).

This approach is commonly used in Heapsort, where basic operations such as finding the left child (2i), right child (2i + 1), or parent (i / 2) of a node can be performed efficiently. On most computers, multiplication and division by two can be replaced with fast bit-shifting operations. To leverage this efficiency, these operations are usually implemented as macros or inline functions, which help Heapsort run faster with less overhead [2].

Chapter V : Graphs

1. Introduction

Graphs are essential tools for modeling relationships between objects. In this chapter, we introduce the basic definitions of graphs, explore how they can be represented in memory, and study key traversal algorithms such as Depth-First Search (DFS) and Breadth-First Search (BFS). These concepts form the foundation for many applications in computer science.

2. Definition

What is a graph? A graph **G** is defined as a pair **(V, E)**, where:

- **V** is a finite, non-empty set of **vertices** (also called **nodes**), and
- **E** is a set of **pairs of vertices** from **V**, called **edges**.

Thus, a graph is represented as: **G = (V, E)**

3. Types of graphs

Depending on how the edges are defined, graphs can be categorized as **undirected** or **directed**.

➢ **Undirected (Symmetric) Graph**

In an undirected graph, the pair of vertices that defines an edge is unordered. In other words, **(v1, v2)** and **(v2, v1)** represent the same edge.

➢ **Directed Graph**

In a directed graph, each edge is represented by an ordered pair *(v1, v2)*, where **v1** is the origin and **v2** is the destination.

In this context, edges are called **arcs**, and **(v1, v2)** and **(v2, v1)** represent two different arcs.

Example:

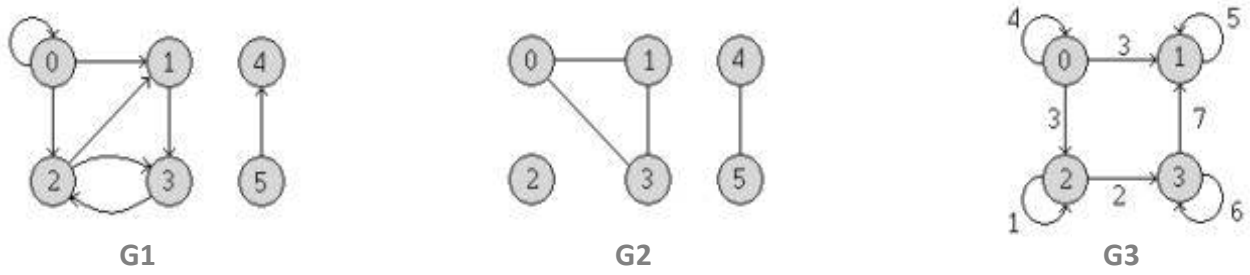


Figure V.1: Examples of graphs.

Where:

G1 = (V, E) = ({0, 1, 2, 3, 4, 5}, {(0, 0), (0, 1), (0, 2), (1, 3), (2, 1), (2, 3), (3, 2), (5, 4)})

G2 = (V, E) = ({0, 1, 2, 3, 4, 5}, {(0, 1), (0, 3), (1, 3), (4, 5)})

G3 = (V, E) = ({0, 1, 2, 3}, {(0, 0), (0, 1), (0, 2), (1, 1), (2, 2), (2, 3), (3, 3), (3, 1)})

4. Basic characteristics of a graph

4.1. Degree of a vertex

The degree of a vertex, denoted **d(v)**, is the number of edges or arcs connected to vertex **v** in a graph.

- In an **undirected graph**, it simply counts the total number of edges incident to the vertex.
- In a **directed graph**, we distinguish between:
  - **In-degree** *(incoming degree)*: the number of arcs directed **toward** the vertex. Notation: $\mathbf{d^-(v)}$
  - **Out-degree** *(outgoing degree)*: the number of arcs directed **away from** the vertex. Notation: $\mathbf{d^+(v)}$

For example, in the graph G1 (Figure V.1), the degrees of the different vertices are as follows:

V	**0**	**1**	**2**	**3**	**4**	**5**
d(v)	4	3	4	3	1	1
$\mathbf{d^-(v)}$	1	2	2	2	1	0
$\mathbf{d^+(v)}$	3	1	2	1	0	1

4.2. The order and size of a graph

The order of a graph is the number of its vertices. ⇒ **Card(V)**

The size of a graph is the number of its edges/arcs. ⇒ **Card(E)**

For example, in G1:

Card(V) = 6

Card(E) = 8

5. Graph representations

Graph representations are methods used to represent the structure of a graph in a way that allows us to perform operations and analyses. In this context, we will focus on two common representations: **Adjacency matrix** and **Adjacency list.**

5.1. Adjacency matrix

An adjacency matrix is a square matrix used to represent a graph, where the size of the matrix is n×n, with n=|V| (the number of vertices in the graph).

- **Unweighted Graph:** If the graph is not weighted, each entry [i][j] in the matrix will contain a boolean value (0 or 1), indicating the absence or presence of an edge between vertex i and vertex j.
- **Weighted Graph:** If the graph is weighted, each entry [i][j] in the matrix, will contain the cost (weight) of the edge connecting vertex i and vertex j. In this case, the value ∞ (infinity) will be used to represent the absence of an edge between the two vertices.

Example:

For a weighted graph with vertices A, B, C, where:

- An edge exists between A and B with weight 3.
- An edge exists between B and C with weight 2.
- No edge exists between A and C.

The adjacency matrix would look like this:

$$\begin{bmatrix} 0 & 3 & \infty \\ 3 & 0 & 2 \\ \infty & 2 & 0 \end{bmatrix}$$

Where:

- The 3 at position [0][1]and [1][0] represents the weight of the edge between A and B.
- The 2 at position [1][2] and [2][1] represents the weight of the edge between B and C.
- The ∞ represents the absence of an edge between A and C.

➢ **Advantages:**

- **Easy to implement:** The adjacency matrix is straightforward to implement since it's just a 2D array.
- **Fast access to graph information:** Accessing the presence or weight of an edge between two vertices requires just one memory access, making it quick.

➢ **Disadvantages:**

- **Redundancy in undirected graphs:** For undirected graphs, the adjacency matrix is symmetric, meaning information is repeated. For example, the edge from A to B is the same as from B to A, which leads to unnecessary redundancy in memory usage.
- **Memory allocation limitations:** An adjacency matrix requires contiguous memory for its 2D array, which can be limiting for very large graphs, especially when the number of vertices is high.
- **Fixed number of vertices:** The adjacency matrix is designed to hold a fixed number of vertices. If the graph evolves (i.e., vertices are added or removed), the matrix size would need to be reallocated, which can be cumbersome.
- **Inefficient for sparse graphs:** If the number of edges is much smaller than the square of the number of vertices ($Card(E) \ll n^2$), the adjacency matrix becomes inefficient in terms of memory usage, as many of its entries would be unnecessary (often set to 0 or ∞).

These advantages and disadvantages highlight that while an adjacency matrix is efficient for dense graphs, it may not be the best choice for sparse graphs or graphs that need frequent modifications.

5.2. Adjacency list

An adjacency list is a more efficient way to represent a graph, especially for sparse graphs. Instead of using a matrix, each vertex has a list (or a similar data structure, such as a linked list) that stores all the vertices that are adjacent to it [9].

5.2.1. Array of lists

Let G=(V,E) be a graph of order n.

This representation consists of an **array of n linked lists**, where each list corresponds to a vertex and contains its **successors** (i.e., the vertices that are adjacent to it). An example of this representation is given in Figure 13 where:

- Each index in the array represents a vertex in the graph.
- The linked list at each index stores all the vertices directly connected to that vertex.
- In a **directed graph**, the list contains successors (outgoing edges).
- In an **undirected graph**, each connection appears in the lists of both connected vertices.

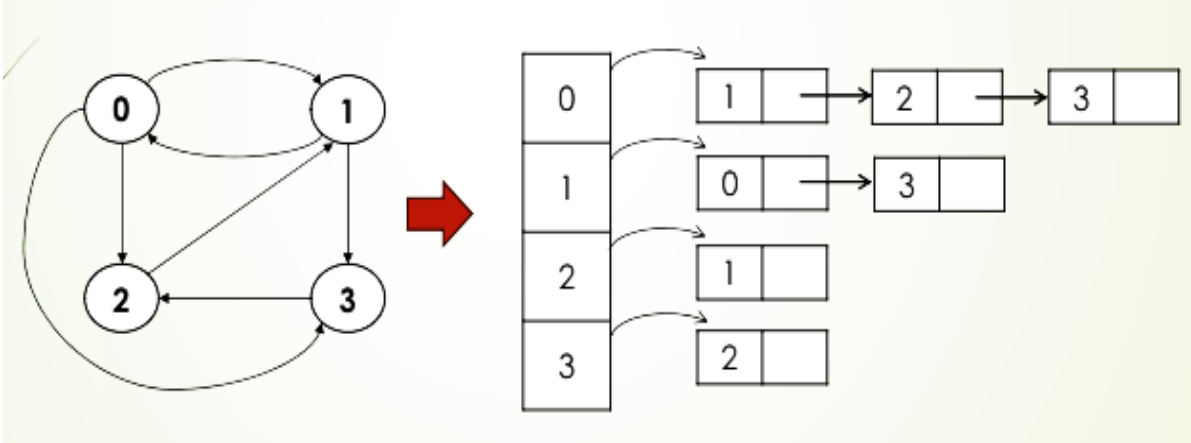


Figure V.2: Array of lists representation.

5.2.2. List of lists

Let G=(V,E) be a graph of order n.

This representation consists of a **list** of **n linked lists**:

- The first-level list contains all the **vertices** of the graph.
- Each vertex is associated with its own linked list, which contains the **edges/arcs** to its **successor vertices**.
- This structure can represent both **directed** and **undirected** graphs.

This representation is **fully dynamic**, meaning:

- Vertices and edges can be added or removed easily.
- It adapts well to graphs whose structure changes during execution.

An example of this representation is given in Figure V.3 below:

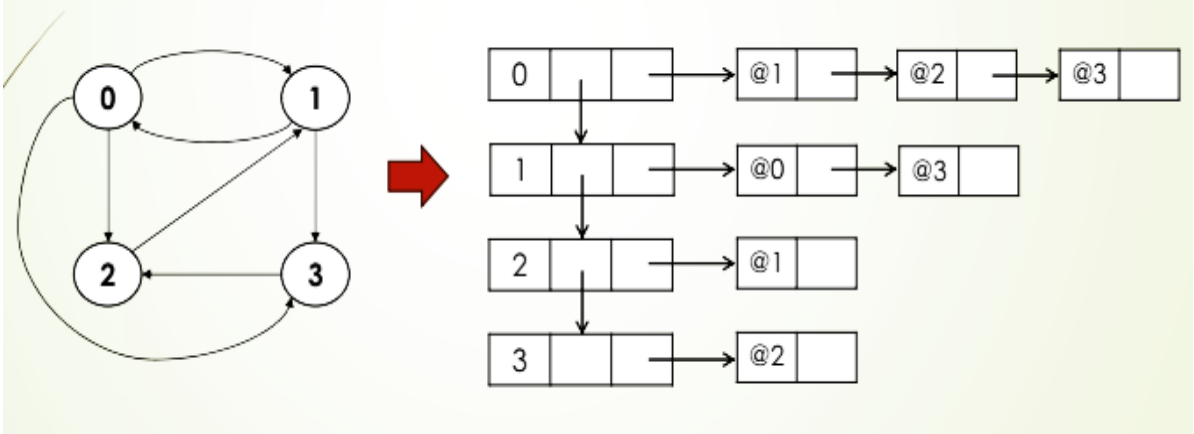


Figure V.3: List of lists representation

6. Graph traversal

Graph traversal consists of visiting each vertex exactly once, starting from an initial vertex and following the edges (or arcs) of the graph.

To ensure that no vertex is visited more than once, we must mark (label) the visited vertices. Initially, all vertices are marked with 0 (unvisited).

There are two main types of graph traversal:

- **Depth-First Traversal (DFS)**
- **Breadth-First Traversal (BFS)**

6.1. Depth First Search (DFS)

6.1.1. Principle

Starting from a given vertex, we mark the current vertex as visited. We visit the **first unvisited neighbour** (successor) of the current vertex and we mark it (e.g., with 1), then we go from that successor to its next successor, and so on.

The idea is to go as far as possible along each path by always choosing an unvisited successor [2].

When we reach a dead end or a vertex that has already been visited, we backtrack to the previous vertex and try the next unvisited successor from there.

Example:

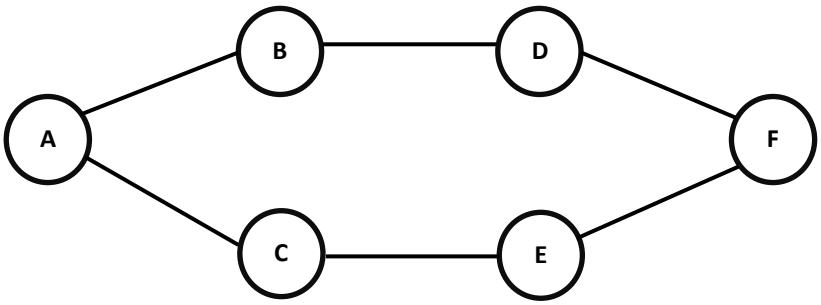


Figure V.4: Example of a cycle graph.

In this example, we have two possible DFS orders starting from vertex **A**:

1. A → B → D → F → E → C
2. A → C → E → F → D → B

6.1.2. Algorithm

The DFS algorithm is based on the use of a **stack** to keep track of the vertices to be visited. In other words:

- At each step, we consider the **last visited vertex** that was pushed onto the stack.
- Then, we **push all its unvisited successors** onto the stack.
- This process **continues until the stack is empty**, meaning all reachable vertices have been visited.

Pseudo-code:

```
Begin
   Initialize an empty stack P;
   Mark all vertices as unvisited (0);
   Mark the starting vertex D as visited (1);
   Push D onto the stack P;

   While P is not empty do
         Let V be the vertex at the top of the stack P;
         If there exists a successor S of V Then
                  If S is unvisited (marked 0) Then
                           Mark S as visited (1);
                           Push S onto the stack P;
                  EndIf
         Else
                  Pop V from the stack P;
         EndIf
   EndWhile
End
```

6.2. Breadth First Search (BFS)

6.2.1. Principle

When a vertex is visited, **all of its neighbors are visited first** before moving on to the next level of vertices.

This process is **repeated for each vertex** until there are **no more unvisited vertices**.

Using the same example from page 50, we have two possible BFS traversal orders starting from vertex A:

1. A → B → C → D → E → F
2. A → C → B → E → D → F

6.2.2. Algorithm

The BFS algorithm can be described as follows:

- When a vertex is visited, it is **inserted into a queue**.
- Then, the **first element of the queue** is removed and replaced with all of its **unvisited successors**.
- This process is **repeated** as long as the queue is not empty.

Pseudo code:

```
Begin
   Initialize an empty queue F;
   Mark all vertices as unvisited (0);
   Mark the starting vertex D as visited (1);
   Enqueue D into F;

   While F is not empty do
      Dequeue a vertex V from F;
      For each successor S of V do
          If S is unvisited (marked 0) then
                    Mark S as visited (1);
                    Enqueue S into F;
        EndIf
      EndFor
   EndWhile
End
```

6.3. Application

Let the following graph [8]:

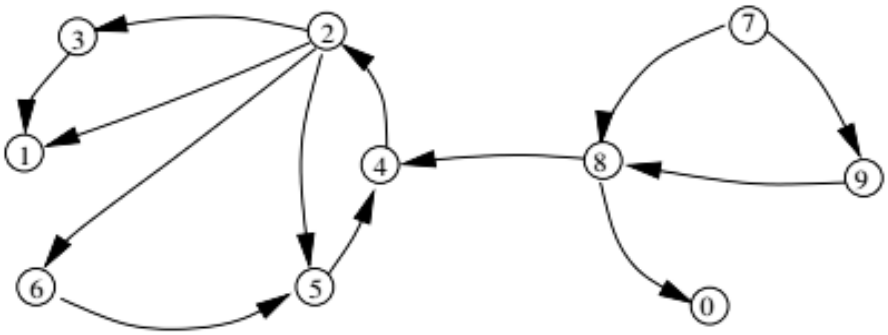


Figure V.5: Example of a directed graph [8]

1. Apply the DFS traversal to the graph above, starting at vertex 2, then 7.
2. Apply now the BFS traversal starting at vertex 4, then 7. In both cases, provide the list of vertices visited in the order of traversal.

DFS Result:

- Starting at vertex 2: 2, 3, 1, 6, 5, 4
- Starting at vertex 7: 7, 9, 8, 0, 4, 2, 3, 1, 6, 5

BFS Result:

- Starting at vertex 4: 4, 2, 3, 1, 6, 5
- Starting at vertex 7: 7, 8, 9, 0, 4, 2, 3, 1, 6, 5

Applying the DFS and BFS traversal algorithms to a directed graph confirms the theoretical properties of each method. DFS explores each branch in depth before backtracking, while BFS proceeds level by level to ensure an orderly traversal based on distance from the starting node. Results obtained from different starting points highlight the impact of the initial node on the visit sequence, particularly in directed graphs where connectivity depends on the origin of the traversal.

Therefore, a solid understanding of graph structures and their underlying algorithms is essential for addressing complex computer science problems, ranging from pathfinding to network analysis. Mastery of these concepts equips learners to tackle algorithmic challenges involving intricate relationships between entities.

PART 2 : Exercises

1. Introduction

Having covered the fundamental concepts and theoretical foundations of algorithms and data structures in the first part of this handout, this second part is dedicated to putting that knowledge into practice. Mastering algorithms and data structures requires active problem-solving, and this section provides you with the essential opportunity to solidify your understanding and develop your analytical and algorithmic thinking.

This part contains a collection of exercises designed to help you apply the principles discussed in the lectures. You will find problems of varying difficulty. Solutions and detailed corrections are provided for many of these exercises, allowing you to check your work, identify areas for improvement, and understand effective solution strategies. Additionally, some exercises are included without immediate corrections to encourage independent practice and self-assessment.

We strongly encourage you to attempt each exercise thoroughly on your own before consulting the provided solutions. This active engagement is crucial for truly grasping the material and building your problem-solving skills in algorithms and data structures.

Note:

It should be noted that some exercises are inspired by the following resources: [11], [12], [8], [2], [9], and [10].

2. Recursion

Exercise 2.1:

The **GCD** (Greatest Common Divisor) is the largest positive integer that divides two or more integers without a remainder. To calculate it, the **Euclidean algorithm** can be used, which is based on the following principle:

- The GCD of a and b (where a > b) is the same as the GCD of b and (a mod b). If b = 0, then the GCD of a and b is equal to a.
- This operation is repeated until the remainder is 0.
- The last non-zero remainder is therefore the GCD.

1. Applying this method, write an (iterative) function that calculates the GCD of two positive integers.

2. Provide the recursive version of this function.

Exercise 2.2:

Write a recursive function that checks for the existence of an element x in an array of integers T, by applying:

a) Sequential search.
b) Binary search (Dichotomy).

Exercise 2.3:

Write a recursive function that calculates the maximum element of an array of real numbers, applying the **dichotomy principle** (using a **divide-and-conquer approach**).

Exercise 2.4: (without solution)

Suppose you want to reverse an array T in the manner shown in the example below:

Before:

12	5	10	3	16	9

After:

9	16	3	10	5	12

Write the corresponding recursive function.

Solution 2.1:

1. Iterative GCD:

```
Function GCD(a, b: integer): integer
Var    r: integer;
Begin
    While (b <> 0) do
            r ← a mod b;
            a ←  b;
            b ←  r;
    EndWhile
    Return a;
End
```

2. Recursive GCD:

```
Function GCD(a, b: integer): integer
Begin
    If (a < b) Then
            Return GCD(b, a);
    Else If (b = 0) Then
            Return a;
        Else
            Return GCD(b, a mod b);
        EndIf
    EndIf
End
```

Solution 2.2:

a) Sequential search:

```
Function seq_search(T: array of integers, strt, end, x: integer): boolean
Begin
    If (strt <= end) Then
            If (x = T[strt]) Then
                    Return True;     // Element found
            Else     // Element not found at 'strt', search in the rest of the array
                    Return seq_search(T, strt + 1, end, x);    // Recursive call
            EndIf
    Else     // Base case: search range is empty (strt > end)
            Return False;              // Element not found in the range
    EndIf
End
```

b) Binary search (Dichotomy):

```
Function bin_search(T: array of integers, strt, end, x: integer): boolean
Var m: integer;
Begin
        If (strt <= fin) Then              // Check if the search range is valid
                m ← (strt + fin) div 2;            // Calculate the middle index (integer division)
                If (x = T[m]) Then         // Check if the middle element is the target
                        Return True;     // Element found
                Else If (x < T[m]) Then    // If target is smaller, search the left half
                         Return bin_search(T, strt, m - 1, x);      // Recursive call on left sub-array
                    Else          // If target is larger, search the right half
                        Return bin_search(T, m + 1, end, x);      // Recursive call on right sub-array
                    EndIf
                EndIf
        Else  // Base case: search range is empty (strt > end)
                Return False;    // Element not found in the range
        EndIf
End
```

Solution 2.3:

```
Function Max(T: array of reals, strt, end: integer): real
Var  m: integer;   u, v: real;
Begin
        If (strt = end) Then               // Base case: sub-array contains a single element
                Return T[strt];  // The maximum is the element itself
        Else  // Sub-array contains more than one element
                m ← (strt + end) div 2;  // Calculate the middle index (integer division)
                // Recursively find the maximum in the two halves:
                u ← Max(T, strt, m);      // Maximum of the left half ( from strt to m)
                v ← Max(T, m + 1, end);  // Maximum of the right half ( from m+1 to end)
                // Combine results: Return the maximum of the two halves found:
                If (u > v) Then
                        Return u;
                Else
                        Return v;
                EndIf
        EndIf
End
```

3. Algorithm complexity

Exercise 3.1

Determine if the function *f* is in the order of *g* (*f = O(g)*?) in the following cases:

a) *f(n)=n2+10n, g(n)=n2*

b) *f(n)=3n3+2n2+n+1, g(n)=n3*

c) *f(n)=2n2+n+1, g(n)=n3*

d) *f(n)=n2+n, g(n)=n*

Exercise 3.2

Determine the number of iterations of the following module and derive its asymptotic complexity.

```
For i from 1 to n
        For j from i+1 to n
                If (T[i] > T[j]) Then
                        tmp ← T[i] ; T[i] ← T[j] ; T[j] ← tmp ;
                EndIf
        EndFor
EndFor
```

Exercise 3.3

Determine the complexity of the following algorithm:

```
Var n, total, i, j, k: integer;
Begin
        total ← 0;
        For i from 1 to n-1
                For j from i+1 to n
                        For k from j+1 to n
                                total ← total + 1;
                        EndFor
                EndFor
        EndFor
        Write (total);
End
```

Exercise 3.4 (without solution)

Consider two square matrices A and B of size n.

a) Write the algorithm that calculates matrix C such that C = A + B.

b) Deduce its asymptotic complexity (*Big O*).

Solution 3.1:

a) f(n)=n²+10n, g(n)=n²

f(n)=O(g(n)) because for $n \geq n_0=1$ and c=1+10=11, we have $n^2+10n \leq 11n^2$.

b) f(n)=3n³+2n²+n+1, g(n)=n³

f(n)=O(g(n)) because for $n \geq n_0=1$ and c=3+2+1+1=7, we have $3n^3+2n^2+n+1 \leq 7n3$.

c) f(n)=2n²+n+1, g(n)=n³

f(n)=O(g(n)) because for $n \geq n_0=1$ and c=2+1+1=4, we have $2n^2+n+1 \leq 4n^3$.

d) f(n)=n²+n, g(n)=n

f(n)≠O(g(n)) because for $n \geq n_0=1$ and c=1+1=2, we have $n^2+n > 2n$.

Solution 3.2:

For the number of iterations, we have:

The 1st loop: $\sum_{i=1}^{n} x$ (Referring to the outer loop controlled by i) such that *x* is the 2nd loop.

The 2nd loop: $x = \sum_{j=i+1}^{n} iter$ (iter: iteration)

Therefore, we will have:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} iter$$

The calculation of this sum results in the total number of iterations:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n} \sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n} n - i = \sum_{i=1}^{n-1} i = \frac{n\,(n-1)}{2}$$

From the number of iterations, we deduce that the complexity of the module is ***O(n²)***.

Solution 3.3:

By applying the following rule: The total complexity of a module (or code block) within a group of nested loops is the complexity of the operations inside the module multiplied by the product of the sizes (number of iterations) of all loops.

Therefore, while the rule is a simplification, applying it in this case correctly leads to the correct asymptotic complexity of ***O(n³)***.

4. Sorting algorithms

Exercise 4.1:

a) Considering the Selection Sort algorithm, illustrate the execution when sorting the array A=[22,11,34,5,8] by giving the content of the array at the end of each iteration of the main loop.

b) Determine the number of comparisons and swaps in the best, worst, and average cases. Deduce its asymptotic complexity (Big O).

Exercise 4.2:

Considering an ascending insertion sort algorithm, illustrate its execution on the following array: B = [17,7,99,75,21,31], by writing the content of the array at each iteration of the main (outer) loop of the algorithm.

Exercise 4.3:

Using a procedure that takes an array of reals as a parameter and performs a Bubble Sort in descending order.

a) Illustrate the execution on the array: C = [25, 5, 8, 3, 12, 9], by giving the content of the array at the end of each iteration of the main loop.

b) Calculate the number of comparisons in the best and worst cases.

Exercise 4.4: (without solution)

Adapt any sorting algorithm to sort an array of strings alphabetically.

Solution 4.1:

a) Execution illustration for A=[22, 11, 34, 5, 8]:

Iteration	Swap	A				
		22	**11**	**34**	**5**	**8**
1	22 and 5	5	11	34	22	8
2	11 and 8	5	8	34	22	11
3	34 and11	5	8	11	22	34
4	22 and 22	5	8	11	22	34

b) Complexity analysis:

For an array of size n, we have:

- **Comparisons:**

  $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$ = **n(n-1)/2** in best, worst, and average cases.

- **Swaps:**

  $\sum_{i=0}^{n-2}$ = **n−1** in best, worst, and average cases.

The number of comparisons dominates the execution time, growing as $O(n^2)$. Swaps contribute O(n).

Since the total time complexity is determined by the highest order term, Selection Sort has a consistent $O(n^2)$ time complexity across all cases due to its fixed number of comparisons.

Solution 4.2:

Execution illustration for B=[17,7,99,75,21,31] :

i	key	B					
		17	**7**	**99**	**75**	**21**	**31**
1	7	7	17	99	75	21	31
2	99	7	17	99	75	21	31
3	75	7	17	75	99	21	31
4	21	7	17	21	75	99	31
5	31	7	17	21	31	75	99

Solution 4.3:

a) Execution illustration for C = [25, 5, 8, 3, 12, 9]:

Iteration	C						Final position
	25	**5**	**8**	**3**	**12**	**9**	
1	25	8	5	12	9	3	3
2	25	8	12	9	5	3	5 and 3
3	25	12	9	8	5	3	8, 5 and 3
4	25	12	9	8	5	3	9, 8, 5 and 3
5	**25**	**12**	**9**	**8**	**5**	**3**	12, 9, 8, 5 and 3

b) Number of comparisons:

Whatever the initial configuration of C, the instructions of the two loops are always executed, so we have:

$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i =$ **n(n-1)/2** comparisons in both cases.

5. Tree structures

Exercise 5.1:

Considering an **n-ary tree**:

1) Define a node structure. The node must be labeled with an alphabetic letter and may have multiple children.

2) Write a function to compute the size of the n-ary tree.

3) Write a function to search for a node labeled with a given letter.

Exercise 5.2:

Considering a **BST (Binary Search Tree)**:

1) Define the corresponding node structure of this tree.

2) Write a function to compute the height of the BST.

3) Write a function to count the total number of nodes in this BST.

Exercise 5.3:

Consider the following list of keys, given in sequential order:

20, 10, 30, 5, 15, 25, 12

1) Construct the corresponding **Min-Heap** by sequentially inserting each key into an initially empty heap, following the order of the list and using an **array representation.** Show the final state of the resulting Min-Heap

2) Starting from the final Min-Heap obtained in Question 1, perform the standard operation to **delete the minimum element** (which is located at the root of the heap). After removing the minimum element and reorganizing the heap to restore its property, show the final state of the resulting heap, again using an **array representation**.

Exercise 5.4:(without solution)

Based on the previous exercise (5.3):

1) Write a function to insert a new element into the Min-Heap.

2) Write a function to extract the minimum element (which corresponds to deleting from the root).

Solution 5.1:

1) Structure definition:

```
Type Node = Structure
    Info: character;          // The label or data of the node (a character)
    NbChild : integer;        // The number of children of this node
    Child : Array[1..NbChild] of Pointer to Node;  // An array of pointers to the child nodes
EndStructure
Var Root : Pointer to Node ;          // A variable to point to the root of the tree
```

2) Function Size :

```
Function Size (N: Pointer to Node) : integer
Var i,s : integer ;
Begin
    If (N = Null) Then
        Return 0 ;  // Base case: If the node is null, size is 0
    EndIf
    s ← 1 ;
    For i from 1 to N.NbChild
        s ← s + Size (Child(i,N)) ;        // Child(i,N) is a predefined primitive returning the // i-th child of N
    EndFor
    Return s ;
End
```

3) Function Search: (We assume that the function returns the node being searched for)

```
Function Search ( N : Pointer to Node, c : character) : Pointer to Node
Var result : Pointer to Node;
Begin
    //Base case: If the current node is null or it is found
    If (N = Null Or Value(N) = c) Then Return N ;
    EndIf

    // Recursively search in each child's subtree
    For i from 1 to N.NbChild
        result ← Search (Child (i,N), c) ;
        If (result <> Null) Then
            Return result ;
        EndIf
    EndFor

    Return Null ; // If the loop finishes without returning, the value wasn't found in this subtree.
End
```

Solution 5.2:

1) Structure definition:

```
Type NodeABR = Structure
        key : integer ;
        lc, rc, pa : Pointer to Node ; // Pointers to the left child, right child, and parent
EndStructure
Var Root : Pointer to Node ;
```

2) Function Height : Knowing that the height is the maximum level in the tree

```
Function High (Node: Pointer to Node): integer
Begin
    If (Node = Null) Then
            Return -1 ;  // Base case: Height of a null tree/subtree is -1
    Else  Return 1 + Max (High (LC(Node)) , High (RC(Node)) );
    EndIf
End
```

3) Function Size :

```
Function NbNodes ( Node: Pointer to Node) : integer
Begin
    If (Node = Nul) Then
            Return 0 ;        // Base case: An empty tree/subtree has 0 nodes
    Else  Return 1 + NbNodes (LC(Node)) + NbNodes (RC(Node));    // Recursive step
End
```

Solution 5.3:

1. Min-Heap construction (Sequential insertion)

We insert the keys one by one into an empty heap

Array representation (0-based index):

- Initial: [ ]
- Insert 20: [20]
- Insert 10: [20, 10] -> swap (10 , 20) -> [10, 20]
- Insert 30: [10, 20, 30]
- Insert 5: [10, 20, 30, 5] -> swap (5 , 20) -> [10, 5, 30, 20] -> swap (5 , 10) -> [5, 10, 30, 20]
- Insert 15: [5, 10, 30, 20, 15]
- Insert 25: [5, 10, 30, 20, 15, 25]
- Insert 12: [5, 10, 30, 20, 15, 25, 12] -> swap (12 , 30) -> [5, 10, 12, 20, 15, 25, 30]

The final state of the Min-Heap: **[5, 10, 12, 20, 15, 25, 30]**

2. Deleting of the minimum element

The minimum element (the root) is 5.

- We remove the root (5).
- We replace the root with the last element of the heap (30).
- We reduce the size of the heap.
- Array state before adjustment: [30, 10, 12, 20, 15, 25] (size 6)
- Now, we perform the adjustment operation on the element at the root (30) to restore the Min-Heap property. We compare 30 with the smaller of its children and swap if necessary, then continue recursively.
    - Compare 30 (index 0) with its children: 10 (index 1) and 12 (index 2). The smaller child is 10.
    - Swap 30 and 10: [10, 30, 12, 20, 15, 25]
    - Now at index 1 (where 30 is). Compare 30 with its children: 20 (index 3) and 15 (index 4). The smaller child is 15.
    - Swap 30 and 15: [10, 15, 12, 20, 30, 25]
    - Now at index 4 (where 30 is). Compare 30 with its children: 25 (index 5). No right child.
    - Compare 30 and 25. 25 is smaller.
    - Swap 30 and 25: [10, 15, 12, 20, 25, 30]
    - Now at index 5. The element (30) has no children. The adjustment operation is complete.

The final state of the Min Heap after deletion: **[10, 15, 12, 20, 25, 30]**

6. Graphs

Exercise 6.1:

Using the example in Figure V.1 (page 53), report the adjacency matrix of each graph.

Exercise 6.2:

Write a recursive function for the DFS (Depth First Search) algorithm.
The main idea is to define a function that **visits all vertices** starting from a given vertex G. For each visited vertex, the function is then **recursively called** on all of its **unvisited successors**.

Exercise 6.3:

Give an algorithm that takes as input a directed acyclic graph ***G*** and two vertices ***vs*** and ***vt,*** and returns the number of simple paths from ***vs*** to ***vt***.

For example, the DAG in the figure below contains exactly three simple paths from vertex ***p*** to vertex ***t***:

p → s → r → u → t, p → o → r → u → t, and ***p → o → s → r → u → t.***

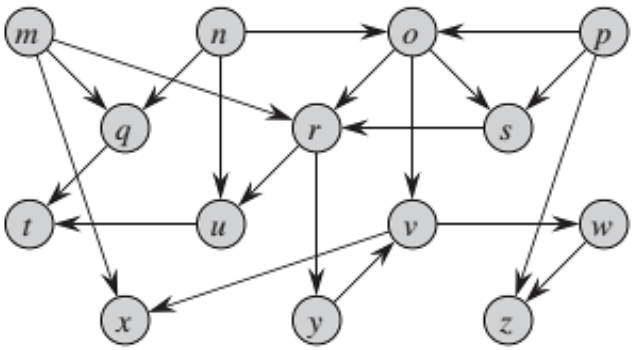


Figure V.6 : Example of a directed acyclic graph [2].

Exercise 6.4: (without solution)

Modify the algorithm of Exercise 6.3 so that it **stores previously computed results** to avoid redundant computations of subpaths.
For example, in the DAG shown in Figure V.6, the subpath from r to t (**r → u → t**) is shared by the three simple paths.
Hint: Use memoization to avoid recomputing the number of paths from a node that has already been processed.

Solution 6.1:

G1: directed graph

1	**1**	**1**	**0**	**0**	**0**
0	**0**	**0**	**1**	**0**	**0**
0	**1**	**0**	**1**	**0**	**0**
0	**0**	**1**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**1**	**0**

G2: indirected graph

0	**1**	**0**	**1**	**0**	**0**
1	**0**	**0**	**1**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**
1	**1**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**1**
0	**0**	**0**	**0**	**1**	**0**

G3: weighted directed graph

4	**3**	**3**	**∞**
∞	**5**	**∞**	**∞**
∞	**∞**	**1**	**2**
∞	**7**	**∞**	**6**

Solution 6.2:

```
Function Rec_DFS (CurrentNode)
Begin
        // To keep track of visited nodes, suppose the node has a boolean field called 'Visited'
        Mark CurrentNode as visited in 'Visited'

        // Do something with the current node

         // Look at all the successors (directly connected nodes) of the current node
        For each successor S of CurrentNode
                If S is not marked as visited Then
                        Rec_DFS(S);      // Recursively call this same function on the successor S
                EndIf
         EndFor
End
```

***Note**: To start, we call the function: Rec_DFS(StartingNode)*

Solution 6.3:

```
// The function CountPaths takes as parameter: the graph, the source and target vertices.
Function CountPaths (G, Vs, Vt)
Begin
        // Base case: If the source node is the target, we have found one path
        If (Vs = Vt) Then
                Return 1
        EndIf

        // Initialize the total count of paths starting from Vs and start a recursive step
        total_paths = 0
        For each successor S of Vs
                // recursively count the number of paths from S to the Vt
                total_paths = total_paths + CountPaths (G, S, Vt)
        EndFor

        Return total_paths
End
```

Conclusion

This document has explored fundamental concepts in algorithms and data structures, offering a solid foundation for understanding the principles of algorithmic problem-solving. From analyzing complexity and exploring common sorting methods to examining tree and graph structures, it equips readers with the essential tools to design and implement efficient computational solutions.

Theoretical foundations, reinforced by practical examples, highlight the importance of choosing appropriate data structures and algorithms based on specific problem requirements. The progression from basic to advanced topics—such as graph traversal—demonstrates how these core concepts interconnect to support sophisticated problem-solving strategies.

In conclusion, the study of algorithms and data structures goes beyond theoretical interest; it is a vital skill for developing real-world software solutions. These concepts underpin many advanced areas of computer science, including artificial intelligence, machine learning, and systems design. Mastery of these fundamentals empowers students to tackle complex challenges with confidence and make informed, performance-conscious design decisions.

As learners advance in computer science, it will be key to build on this foundation through continued practice and exploration to meet the evolving demands of modern computing.

Bibliography

[1] D. Berthet and V. Labatut, *Algorithmique & programmation en langage C - vol.1 : Supports de cours. Licence. Algorithmique et Programmation*, Université Galatasaray, 2014. [Online]. Available: https://hal.science/cel-01176119v2

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, Third edition. Cambridge, Massachusetts London, England: MIT Press, 2009.

[3] J.-P. Préaux, *Complexité d'un algorithme :* Support de cours. [Online]. Available:https://www.i2m.univ-amu.fr/perso/jean-philippe.preaux/PDFenseignement/InfoPCSI/sup11_Complexite.pdf [Accessed: Jan. 12, 2025].

[4] *Complete Guide on Complexity Analysis - Data Structure and Algorithms Tutorial*. [Online]. Available:https://www.geeksforgeeks.org/dsa/complete-guide-on-complexity-analysis [Accessed: Apr. 21, 2025].

[5] K. Zampieri, S. Rivière, and B. Amerein-Soltner, *Complexité des algorithmes :* Support de cours. [Online]. Available: https://ressources.unisciel.fr/algoprog/s34plexite/emodules/cx00macours1/res/cx00cours-texte-xxx.pdf [Accessed: Dec. 4, 2024].

[6] A. Djeffal, *Algorithmique et Structures de Données 3 :* Support de cours. [Online]. Available: https://www.abdelhamid-djeffal.net/web_documents/introalgo2122.pdf [Accessed: Nov. 12, 2024].

[7] T. H. Cormen, *Algorithmes notions de base*. Paris: Dunod, 2013.

[8] R. Malgouyres, R. Zrour, and F. Feschet, *Initiation à l'algorithmique et à la programmation en C: cours avec 129 exercices corrigés*. Paris: Dunod, 2014.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Algorithmique : cours avec 957 exercices et 158 problèmes*, 3e édition. Paris: Dunod, 2010.

[10] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Boston, MA, USA: Addison-Wesley Professional, 1997.

[11] D. Berthet and V. Labatut, *Algorithmique & programmation en langage C - vol.2 : Corrigés de travaux pratiques. Licence. Algorithmique et Programmation*, Université Galatasaray, 2014. [Online]. Available: https://hal.science/cel-01176120

[12] D. Berthet and V. Labatut, *Algorithmique & programmation en langage C - vol.3 : Sujets de travaux pratiques. Licence. Algorithmique et Programmation*, Université Galatasaray, 2014. [Online]. Available: https://hal.science/cel-0117612