PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research
University of Tindouf

Faculty of Sciences and Technology
Department of Computer Science

DATA STRUCTURES AND ALGORITHMS 3

Major: Information systems

Level: second Year

Authored by :
Dr. DRAOUI Abdelghani

Academic Year : 2025-2026

TABLE OF CONTENT

I. GENERAL INTRODUCTION ....

1 **Review and Basic Definitions** .... - 1 -

1.1 Introduction .... - 1 -

1.2 Data structure .... - 1 -

1.3 Algorithm .... - 1 -

1.4 Example of algorithms .... - 1 -

1.5 Classification of algorithms .... - 2 -

1.6 Evaluation of algorithms .... - 2 -

1.7 How to express an algorithm? .... - 2 -

1.8 Data types .... - 2 -

1.9 Reading user input and displaying output .... - 3 -

1.10 Types of Operators .... - 3 -

1.10.1 Assignment .... - 3 -

1.10.2 Arithmetic operators: +, -, *, /, div, mod .... - 3 -

1.10.3 Priority rules .... - 3 -

1.10.4 Logical and relational operators .... - 4 -

1.11 Control flow structures .... - 4 -

1.11.1 Conditional control structures .... - 4 -

1.11.2 Iterative structures .... - 5 -

1.11.3 For loop .... - 5 -

1.11.4 While loop .... - 5 -

1.11.5 Do …. While .... - 5 -

1.12 Arrays .... - 6 -

1.12.1 One Dimensional array .... - 6 -

1.12.2 Multidimensional dimensional array .... - 7 -

1.13 Structure .... - 7 -

1.14 Subprograms (functions and procedures) .... - 8 -

1.14.1 Subprogram syntax .... - 8 -

1.15 Recursion .... - 9 -

1.15.1 Steps to Implement Recursion .... - 9 -

1.16 Pointer .... - 11 -

1.16.1 Pointer syntax .... - 11 -

1.17 Dynamic memory management .... - 11 -

1.18 Exercises .... - 13 -

TABLE OF CONTENT

1.19 Conclusion .... - 14 -

2. **Chapter 01: Time and Space Complexity** .... - 15 -

2.1. Introduction .... - 15 -
2.2. Time Complexity Analysis .... - 15 -
2.3. Best, average and worst case scenarios .... - 16 -
2.4. Measure of order of growth of an algorithm .... - 16 -
2.5. Asymptotic notations (big-Oh, big-theta and big-omega) .... - 17 -
2.5.1. Definition .... - 17 -
2.5.2. General rules .... - 18 -
2.6. Classes of complexity .... - 20 -
2.6.1. Constant complexity O(1) .... - 20 -
2.6.2. Logarithmic complexity O(log(n)) .... - 20 -
2.6.3. Linear complexity O(n) .... - 20 -
2.6.4. Quasi-linear complexity O(n*log(n)) : .... - 21 -
2.6.5. Quadratic complexity $O(n^2)$ .... - 21 -
2.6.6. Polynomial complexity $O(n^k)$ .... - 22 -
2.6.7. Exponential Complexity $O(2^n)$ .... - 22 -
2.6.8. Factorial complexity O(n !) .... - 22 -
2.7. The big omega notation (Ω-notation) .... - 24 -
2.7.1. Definition .... - 24 -
2.8. The big theta notation (Θ-notation) .... - 25 -
2.8.1. Definition .... - 25 -
2.8.2. Theorem .... - 26 -
2.9. Time Complexity for Recursive Functions .... - 26 -
2.10. Calculation of space complexity .... - 29 -
2.10.1. For non-recursive algorithms .... - 29 -
2.10.2. For recursive algorithms .... - 29 -
2.11. Conclusion .... - 29 -
2.12. Exercises .... - 30 -

3. **Chapter 02: Sorting algorithms** .... - 31 -

3.1. Introduction .... - 31 -
3.2. Characterization of sorting algorithms .... - 31 -
3.3. Selection sort .... - 32 -
3.3.1. Selection sort algorithm .... - 32 -

TABLE OF CONTENT

3.3.2. Pseudocode of selection sort algorithm .... - 32 -
3.3.3. Selection sort time complexity calculation .... - 33 -
3.4. Bubble sort .... - 34 -
3.4.1. Bubble sort algorithm .... - 34 -
3.4.2. The pseudocode of bubble sort algorithm .... - 35 -
3.4.3. Bubble sort algorithm time complexity .... - 35 -
3.4.4. Improvement of bubble sort algorithm .... - 36 -
3.5. Insertion sort .... - 37 -
3.5.1. Insertion Sort algorithm .... - 37 -
3.5.2. Pseudocode of insertion sort .... - 38 -
3.5.3. Time complexity Analysis of insertion sort .... - 38 -
3.6. Binary Search (Or bisection search) .... - 40 -
3.6.1. The binary search algorithm .... - 40 -
3.6.2. Pseudocode of binary search (iterative solution) .... - 41 -
.3.7 Merge sort .... - 42 -
3.7.1. The merge sort algorithm .... - 42 -
3.7.2. Merge sort Algorithm .... - 43 -
3.7.3. Time complexity analysis of Merge sort .... - 44 -
3.7.4. Stable or instable ? .... - 47 -
.3.8 Quick sort .... - 48 -
3.8.1. Quick sort algorithm .... - 48 -
3.8.2. Quick sort algorithm pseudocode .... - 50 -
3.8.3. Time complexity of Quick sort algorithm .... - 51 -
3.8.4. Improved version of quick sort .... - 53 -
3.8.5. Space complexity analysis of quick sort .... - 54 -
3.9. Comparison of sorting algorithms .... - 55 -
3.10. Conclusion .... - 56 -
3.11. Exercises .... - 56 -

4. **Chapter 03: Trees** .... - 57 -
4.1. Introduction .... - 57 -
4.2. Definitions and terminology .... - 57 -
.4.3 Generic tree (n-ary tree) to binary tree conversion .... - 59 -
4.4. Types of binary trees (according to its structure) .... - 60 -
4.5. Tree traversal .... - 62 -

TABLE OF CONTENT

4.5.1. Depth First Traversal .... - 62 -
4.5.2. Breadth first traversal .... - 64 -
4.6. Max heap & Min heap .... - 64 -
4.6.1. Heap .... - 64 -
4.6.2. Max heap .... - 64 -
4.6.3. Min heap .... - 64 -
4.6.4. Representation of max heap in memory .... - 65 -
4.6.5. Insertion in a max heap .... - 66 -
4.6.6. The insertion algorithm .... - 67 -
4.6.7. Building max heap from an arbitrary complete binary tree .... - 67 -
4.6.8. Deletion of root node from max heap .... - 69 -
4.6.9. Heap Sort .... - 70 -
4.6.10. Heapifying a complete binary tree .... - 70 -
4.6.11. Building a max heap using heapify .... - 71 -
4.6.12. Searching for a value in max heap .... - 71 -
4.7. Additional definitions .... - 71 -
4.8. Time complexity analysis of operations on tree .... - 73 -
4.8.1. Time complexity of deletion from a max heap .... - 74 -
4.9. Binary Search Tree (BST) .... - 75 -
4.9.1. Advantages of BST .... - 75 -
4.9.2. Implementation of binary trees .... - 76 -
4.9.3. Frequent operations on BSTs .... - 77 -
4.9.4. Searching for a value in a BST .... - 79 -
4.9.5. Operation of finding the node with maximum value .... - 79 -
4.10. Implementation of Traversals .... - 80 -
4.10.1. Depth-first traversals .... - 80 -
4.10.2. Breadth-first traversal implementation .... - 80 -
4.10.3. Time complexity of traversals .... - 81 -
4.11. Deletion operation .... - 81 -
4.12. Operation of counting nodes .... - 83 -
4.13. Operation of search for successor of a given node .... - 83 -
4.14. Conclusion .... - 84 -
4.15. Exercises .... - 84 –

TABLE OF CONTENT

5. **Chapter 04 : Graphs** .... - 85 -
5.1. Introduction .... - 85 -
5.2. Basic Definitions and Terminology .... - 85 -
5.3. REPRESENTATIONS OF A GRAPH .... - 88 -
5.3.1. Array Representation .... - 88 -
5.3.2. Linked List Representation .... - 89 -
5.4. DEPTH-FIRST TRAVERSAL .... - 89 -
5.4.1. Introduction .... - 89 -
5.4.2. Program .... - 90 -
5.4.3. Explanation .... - 91 -
5.4.4. Analysis .... - 91 -
5.5. BREADTH-FIRST TRAVERSAL .... - 92 -
5.5.1. Introduction .... - 92 -
5.5.2. Program .... - 92 -
5.6. Exercises .... - 94 -
5.7. Conclusion .... - 94 -

II. GENERAL CONCLUSION ....
III. REFERENCES ....

General Introduction

General introduction

Data Structures and Algorithms (DSA) form the foundation of efficient problem-solving in computer science and software engineering. These lecture notes are designed to help you understand how data can be organized, processed, and manipulated effectively, and how algorithmic thinking leads to scalable and optimized solutions.

The lecture notes begin with algorithmic complexity, where you will learn how to analyze the efficiency of algorithms in terms of time and space. Concepts such as Big-O notation will help you compare solutions and make informed decisions about performance, especially as input sizes grow.

Another key component of the course is sorting algorithms. You will learn multiple approaches to sorting data, analyze their performance characteristics, and understand the trade-offs between simplicity, speed, and memory usage.

You will then explore core data structures, focusing on how data can be stored and accessed efficiently. Building on this, the course introduces trees and graphs, which are powerful structures used to model hierarchical relationships, networks, and real-world systems such as file systems, social networks, and routing paths. You will study common traversal and search techniques and understand when and why these structures are used.

By the end of the course, you will be able to:

- Analyze the efficiency of algorithms using complexity analysis
- Choose appropriate data structures for specific problems
- Work with trees and graphs to model complex relationships
- Implement and compare common sorting algorithms

Overall, this course builds strong analytical and practical skills that are essential for technical interviews, advanced computer science topics, and real-world software development.

Review of DSA 2

1 Review and Basic Definitions

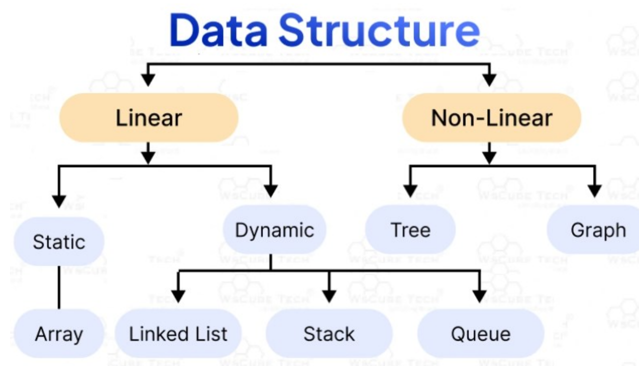
1.1 Introduction

Data Structures and Algorithms 3 (DSA3), focuses mainly on the concept of time and space complexity and how this tool is used to compare several sorting algorithms that will be studied later on. Understanding these concepts requires knowing some fundamental concepts which are covered in this review.

1.2 Data structure

A data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes. Figure below shows the most popular data structures.

Fig 1. Classification of data structures



Note: Stack and Queue data structures can be implemented statically using arrays, as we will see later in this course. Non-linear data structures such as tree and graph are covered in the follow-up course, Data Structures and algorithms 3.

1.3 Algorithm

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a finite amount of time [1]. An algorithm is thus a sequence of computational steps that transform the input into the output.

1.4 Example of algorithms

We use algorithms in our daily life, here are some few examples:

- A food recipe (ingredients(as input) -> prepared meal(result)).
- Looking for meaning of a word in the dictionary (word -> definition).
- Sorting a sequence of elements (unsorted sequence -> sorted sequence).

For a sorting algorithm, if the input sequence <1,9,5,4> is given the algorithm should output the following output sequence <1,4,5,9>.

The input sequence <1,9,5,4> is called a problem instance.

The problem instance consists of a valid input needed to compute a solution to the problem.

1.5 Classification of algorithms

An algorithm is correct if, for every problem instance provided as input, it finishes its computing in finite time and output the correct solution.

An incorrect algorithm might not finish at all on some input instances, or it might finish with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if you can control this error rate.

We will concern ourselves only with correct algorithms.

1.6 Evaluation of algorithms

Algorithms are evaluated according to their use of resources. Running time and consumption of memory (RAM).

To evaluate and potentially improve our algorithm, we should ask ourselves these questions:

- Is my algorithm a correct one?
- How much is its running time?
- How much space in memory is consumed?
- Are there any way to do better?

Note: Estimating the running time and the space of memory consumed are covered in the advanced course DSA3.

1.7 How to express an algorithm?

An algorithm can be specified in natural language (Arabic, French, English …etc.), flow chart, pseudocode, or computer program. The only rule to be respected is that the algorithm should have clear meaning.

A pseudocode, it is a method of expressing algorithm where a mixture of expressions in natural language and expressions used in programming language (assignment, comparison, addition …etc) are used, it is intended for human to focus on understanding algorithms without worrying about the differences in syntax in the various programming languages.

Note: In this course we will use C programming language and the pseudocode to write algorithms.

1.8 Data types

Almost every algorithmic problem requires storage and manipulation of data of a specific type, the most frequently used data types are summarized in the table below :

type	Example of possible values
integer	-400 -6 10000
Real	4.459 -3.1543
character	‘a’ ‘A’ ‘_’ ‘9’
string	‘’University of Tindouf’’
Boolean	True, False

1.9 Reading user input and displaying output

Values of variables -defined in the program- can be read from input of the user. Displaying the output of the program, is usually done, in a window. To read inputs and display output to the user, we use the read and print functions. Table below show an example of use of this functions:

Pseudocode	Example
read	read(a)
print	print('The value of b is : ',b)

1.10 Types of Operators

1.10.1 Assignment

Assigning values to variables can be performed using an assignment operator. In our pseudocode we adopt the combined symbol '**<-**' for this task.

Pseudocode	Explanation
A <- B	Value of B is assigned to A using the assignment operator <-

1.10.2 Arithmetic operators: +, -, *, /, div, mod

We can perform arithmetic operations on numerical data types such as integer and real numbers, the modulo (i.e. the remainder of division) is denoted by '**%**' symbol.

Operation	Result (Explanation)
5+9	14
60-20	40
15/2	7.5
15 **div** 2	7 (Euclidian division)
9 **%** 2	1 (remainder of division of 9 over 2)

Note: Performing arithmetic operation on variables of character type, corresponds to performing these operation on the corresponding ASCII code of these characters.

1.10.3 Priority rules

In an instruction containing more than one arithmetic operation, the following order of performing these operations has to be respected:

Order	Operators
01	(),[],{}
02	Exponents like : x^2 x^3
03	**×**, /
04	+, -

Note: For operators having same priority, we perform calculation from left to right.

Example: 100/4*5 , a is equal to 125 and not 5 !

1.10.4 Logical and relational operators

The logical operator (and, or, negation) along with relational operators leads always to true and false answer, we use 0 to denote false and 1 to denote true.

Operation	Result (Explanation)
5 = 9	0 (False)
60 ≠ 20	1 (True)
1 and 0	0 (False)
1 or 0	1 (True)
>, <, >=, <=	/

Note: In C language, we use 0 to express 'False' value and any other value to express True value.

1.11 Control flow structures

The order of execution of a program is from top instruction to bottom one, an instruction or a group of instructions can be prevented from execution using one of the conditional structures, if instructions are needed to be executed multiple times before proceeding to the execution of the subsequent lines one type of loops is used.

1.11.1 Conditional control structures

The instructions inside the conditional block is executed only if a condition or group of conditions are satisfied.

1.11.1.1 If structure

Pseudocode	Example

If Condition(s) then
Instruction(s)
endif
	If (age > 18) then
	print(' you are an adult')
	endif

1.11.1.2 If-else if –else structure

Execution of instruction can be restricted to satisfaction of one of a group of conditions.

Pseudocode	Example
if Condition(s) **then**	**if** a > b **then**
Instruction(s)	Print('a is larger than b')
Else if Condition(s) **then**	**Else if** a < b
Instruction(s)	Print('b is larger than a')
Else Condition(s) **then**	**Else** if a < b
Instruction(s)	Print('b is equal to a')
endIf	**endif**

1.11.1.3 Switch case structure

Switch-case structure is preferred when dealing with many conditions.

Pseudocode	Example
Switch variable or expression **Case** value 1 : action1 **Case** value 2 : action2 … … **Otherwise** a default action **EndSwitch**	**Switch** Variable 1 **Case** 1: print(‘winter’) **Case** 2: print(‘spring’) **Case** 3: print(‘summer’) **Case** 4: print(‘autumn’) **Otherwise** print(‘Error’) **EndSwitch**

1.11.2 Iterative structures

1.11.3 For loop

If the number of execution of some instructions is known, it is preferable to use the ‘for loop’.

Pseudocode	Example

for i from .. to .. do
Instruction(s)
EndFor
	for i from 1 to 20 do
	Print(‘Hello’)
	EndFor

1.11.4 While loop

In some cases, number of iterations of some group of instructions in not known, therefore, we use the while loop.

Pseudocode	Example
while condition(s) Instruction(s) EndWhile	while grade > 10 print(‘Congratulations’) EndWhile

1.11.5 Do …. While

If a group of instructions is needed to be performed at least once, the ‘do-while’ loop is preferred

Pseudocode	Example
do Instruction Instruction while conditions(s)	do print(‘hi !’) write(‘type N to terminate’) while C = ’N’

Remark: Having a code written using one of the aforementioned loop types, we can rewrite the same code using the other type, below is an example

For loop	While loop

For i <- 2 to n do
    Instruction(s)
EndFor
	i <- 2
	while (i ≤ n)
	        Instruction(s)
	        i <- i + 1
	EndWhile

1.12 Arrays

1.12.1 One Dimensional array

Question 1: How can we save grades of 100 students?

Question 2: Do we have to declare 100 separate variables, each storing a grade of one student?

Answer: The best way to store grades of 100 students is to declare one single variable whose type is one dimensional array.

In general, in order to store multiple values of variables of the same type, we can use one dimensional arrays. The drawbacks of using arrays to store data is that their size cannot be changed during the execution time and the elements of the array must have same data type.

A 1D-array can be declared following the syntax below :

var arrayName [0,..,size] : data type

where :

arrayName : denotes the name of the array.

Data type : denotes the data type of elements of the array.

To access the ith element of the array arrayName, we use the following syntax : T[i].

```
Algorithm Filling_1D_table
Variable T[0,..,4] : array of integers, i :integer
Begin
   For i from 1 to 4 do
            Print('insert element index ',i)
            Read(T[i])
   EndFor
End
```

1.12.2 Multidimensional dimensional array

We would like to write a program that calculates the average of each student knowing that the class contains 20 students who studied 6 courses.

Question: How can we store this type of data efficiently?

Answer: Ideally, we use two-dimensional array, where one dimension represent the students and the other represents the modules, in that case the syntax would be :

Var Grades[0,..,19 ;0,..,5] : array of real nbrs

To Access the jth grade of ith student we adopt the following syntax Grades[i,j]

Example:

```
Algorithm filling of 2D array
Variables T[1,..,4 ;1,..,6] : array of real nbrs, i,j :integer
begin
        for i from 1 to 4 do
            for j from 1 to 6 do
                Print('insert grade', j, ' of student',i)
                Read(T[i ;j])
            EndFor
        EndFor
End
```

1.13 Structure

When dealing with entities having many attributes, where each attribute has different data type, in that case, we declare a variable of type structure. A structure is a compound data type, composed by multiple elementary data types.

In order to declare a variable of type structure, we need to define structure data type. We can access its attributes using the dot notation.

Structure syntax	Example

Type Identifier=structure
   Field1 : Type1
   Field2 : Type2
   ...
   ...
   Field N : Type N
EndStructure
	Type Car=Structure
	  Manufacturer : string
	  Color : string
	  HorsePower : integer
	
	EndStructure

Use of created structure to declare variables	Example
Variable Var1 : Identifier begin Var1.Field1 <- Value1 Var1.Field2 <- Value2 Var1.Field3 <- Value3 End	Variable V : Car Begin V. Manufacturer <- 'BMW' V. Color <- 'Red' V. HorsePower <- 120 End

1.14 Subprograms (functions and procedures)

Subprograms are used to avoid code repetition, therefore, ensuring better maintainability of the code. If a subprogram needs data to perform the instructions, this data is given in a form of input parameters. If the subprogram output is needed in some instructions of the program, the subprogram is defined as a function, otherwise it can be easily defined as a procedure.

1.14.1 Subprogram syntax

Any subprogram has two parts, a subprogram definition and a subprogram call. A subprogram is defined first and then it can be called one or multiple times inside our program.

Subprogram definition contain the following parts : Function name, parameters, return type subprogram header and, subprogram body

Subprogram definition syntax

```
function  add(x ,y : integers) :
integer
Var z : integers
Begin
     z <- x + y
     return z
End
```

If no value is required to be returned the subprogram is defined as a procedure, in that case, we change the keyword function with procedure, and the return type is omitted.

To execute instructions inside a subprogram, this subprogram has to be called with the proper arguments. Calling a subprogram of type procedure is straightforward, the name of the procedure is mentioned in the intended place in the program. If the subprogram is of type function (i.e. it returns a value), this returned value can be exploited by calling the function at the right hand side of the assignment, inside condition of if block, inside while loop condition or inside a display function. Below an example of a program written with and without use of subprogram

A program written without use of subprograms	A program written with the use of subprograms

Algorithm display
Var a<-5, b<-6 ,c : integer
Begin
    c <- a + b
    print('a + b is equal to : ',c)
    a <- 2
    b <- 3
    c <- a + b
    print('a + b is equal to : ',c)
end
	Algorithm display
	procedure add(x ,y : integers)
	Var z : integer
	Begin
	    z <- x + y
	    print('a + b is equal to : ',z)
	End
	Var a<-5, b<-6 ,c : integers
	Begin
	    c <- add(a,b)
	    a <- 2
	    b <- 3
	    c <- add(a,b)
	end

To see the utility of subprogram use, consider this case, If the program is to be modified to compute the product instead of sum, the program written without the use of subprogram requires modification of 4 lines, while the program written using a subprogram demanded modification of only 2 lines.

1.15 Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. If solving a problem requires solving a smaller version of the same problem, recursion can be used to formulate this solution.

A recursive algorithm takes one step toward solution and then recursively call itself to further move. The algorithm stops once we reach the smallest version of the problem, called a base case. Since the called function may further call itself, this process might continue forever. So it is essential to provide **that base case** to terminate this recursion process.

1.15.1 Steps to Implement Recursion

Step1 - Define **a base case**: Identify the simplest (or base) case for which the solution is known or trivial.

Step2 - Define **a recursive case**: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 - Ensure **the recursion eventually reaches the base case**: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

Example : Given a natural number ‘n’, define a recursive subprogram that calculates n! (factorial of n)

n! = 1 * 2 * …….* (n-1)*n

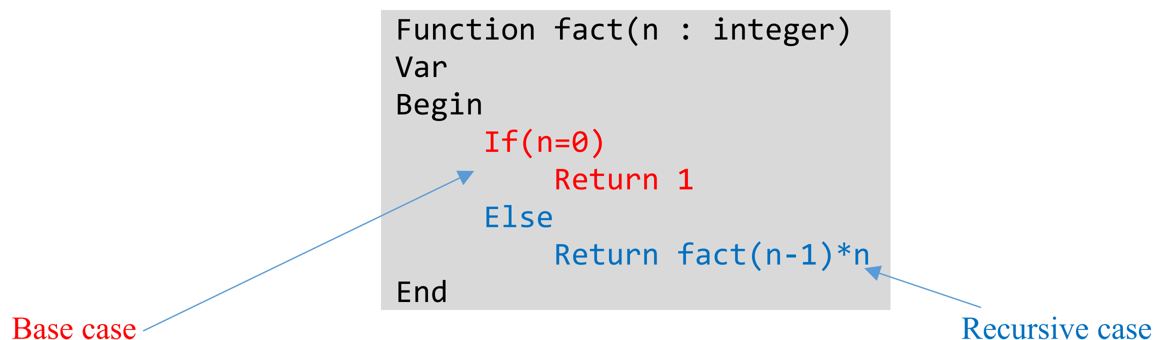
   = (n - 1)! * n

Factorial of the smallest natural number ‘n’ represents the base case, 0! = 1.

We notice that calculating factorial of ‘n’ requires calculating factorial of ‘n-1’. Therefore, we are dealing with an algorithm which is recursive in nature. Fact(n) subprogram can be summarized in mathematical notation as :

$$Fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ Fact(n-1) * n & \text{if } n > 0 \end{cases}$$

Following the above mathematical formula, the recursive subprogram can be easily defined as



```
Function fact(n : integer)
Var
Begin
     If(n=0)
         Return 1
     Else
         Return fact(n-1)*n
End
```


Example: Given a real number X and a natural number n, define a recursive subprogram that calculates X^n

Given that $X^n = X^{n-1} * X$ and $X^0 = 1$, we notice that calculating X^n requires calculating power of same real number X with a reduced parameter n-1. Among many options $X^0 = 1$ can be selected as a base case. Based on these remarks, we can define the required subprogram as follows.

```
Function pow(x: real, n : integer)
Var
Begin
     If(n=0)
          Return 1
     Else
          Return x*pow(x,n-1)
     EndIf
End
```

1.16 Pointer

A pointer is a variable defined to store the address of another variable of a same type. Any subprogram having the address of a variable in memory, it can be accessed/modified via the pointer holding its address.

1.16.1 Pointer syntax

```
Algorithm pointerExample1
Var a : integer
    P : pointer on integer
Begin
   1    a <- 294
   2    p <- &a
   3    *p = *p + 6
   4    Print('a = ',*p)
End
```

A variable can be modified through the pointer holding its address. In the following example:

- An integer variable ‘a’ and a pointer on integer ‘p’ are declared.
- **Line 1**: An integer variable ‘a’ is declared
- **Line 2**: Address value of the integer ‘a’ is stored in ‘p’. In other words, address of any variable ‘x’ is specified using the ‘&’ symbol as ‘&a’.
- **Line 3**: Variable whose value is stored in ‘p’ is updated.
- **Line 4**: the integer variable on which ‘p’ is pointing is displayed .

The line p = &a represents a **referencing operation** which consist of assigning the pointer as reference to the variable. While the syntax (*p) represents the opposite operation referred as **dereferencing operation**, in which the variable pointed by ‘p’ is targeted.

In dereferencing operation, a number of bytes is deduced from the size of the pointer datatype. This number of targeted bytes is considered as a single value and returned. That is the main reason requiring the pointer having same type of the variable to which it is pointing.

1.17 Dynamic memory management

Local variables are created in the stack frame of the function in which it is used. The memory size to be allocated for storing variables in pre-defined in the compilation process. On the other hand, a space in a memory can be allocated dynamically for a variable with a desired size (i.e. in the run time of the program) and that is in the heap section of memory. The allocated space in memory can be accessed/modified through a pointer holding its address.

In C language this allocated space of memory has to be freed 'manually' by explicitly stating it the program, otherwise , a memory leak will occur due to occupation of an inaccessible space in memory. In other programming language such as Java, freeing occupied memory is performed automatically by the garbage collector (further details about this concept will be studied in Object Oriented Programming module).

In the example below, a simple program demonstrating the dynamic memory management is illustrated. Two slots of memory are created to manipulate. One slot via a variable while manipulating the other slot via a pointer. The latter slot is manipulated using a pointer because it is dynamically allocated in the heap section of memory. The scope of the second slot of memory is global as it can be changed by any subprogram having its address.

```
Algorithm pointerExample1
Var P : pointer on integer
Begin

   1    p <- allocate[sizeOf(integer)]
   2    *p = *p + 6
   3    Free(p)
End
```

For dynamic allocation of memory in the heap section of the memory, the allocate function is used. It allocates a memory of size given as an argument then it returns the address of that allocated memory. This function returns the address of the allocated block of memory. The free() function is used to free up the memory slot previously allocated.

In this program, a space for use for storing an integer number is desired. Since that size allocated for variables of type 'int' varies depending on the compiler used, sizeof() function is used to specify the correct size of 'integer' as set by the used compiler.

In summary, the example above shows how to allocate, exploit and, free a space in memory and that is in the run time. The size of the allocated memory space can be specified by the user input as well, an entry that cannot be predicted for the compiler to allocate it statically.

A space in memory can be allocated dynamically for more complex data types such as structures of a certain entity. Below is an example of a program along with the memory sections involved in the execution of the program.

```
Wilaya : structure
     matricule : int
     classemnt : float
     categorie : character
EndOfStructure
Algorithm pointerExample1
Var p : pointer on wilaya structure
Begin

   1    p <- allocate[sizeOf(wilaya structure)]
   2    p.matricule  <- 37
   3    p.classement <- 8
   4    p.categorie  <- 'd'
   5    Free(p)
End
```

An entity 'wilaya' with attributes 'matricule', 'classement' and, 'category' is defined as a structure. A space having size of this structure is allocated in heap section of memory using the alloc() function. Attributes of this entity represented in a form of a structure are accessed/modified through the pointer 'p' which holds the address of this structure. The access to any attribute is achieved using the dot notation ('.'). This notation is a shortcut for dereferencing operation.

In short, for dynamic memory usage (i.e. for usage in the run time), memory is allocated using the allocate() function. That memory block is accessed through a pointer, and it is freed (deallocated) through the same pointer using the free() function.

1.18 Exercises

1. Write an algorithm that asks the user for two integer numbers to output the remainder of division. (hint: use conditional structure to avoid division over zero).
2. Write a program that tells if an integer number given by the user is odd or even.
3. Write a program that display only the even numbers of a pre-defined array of integers.
4. Write a program that display all the divisors of an integer number given by the user.
5. Write a program that gives the roots of a quadratic equation.
6. Write a program that receives the three sides of a triangle, to check if this triangle is a right one.
7. Write a program that calculate the area of a disc of a given radius 'r'.
8. Define another subprogram that calculate the volume of a cylinder of a given radius 'r' and height 'h'.

9. We want to store data of student of our department, show how you can use structure to store data of each student with the following pieces of information: name, age, gender.

1.19 Conclusion

In this introductory part, a quick and brief review of fundamental concepts are exposed. These concepts represents the pre-requisites of this course. For a detailed explanation, reader may consult of the references mentioned in the references section.

Chapter 01 : Time and Space complexities

2. Chapter 01: Time and Space Complexity

2.1. Introduction

Each algorithm is evaluated in terms of time that it takes to execute and in terms of memory it consumes during its execution. In other words, we evaluate an algorithm by calculating its **time and space complexity**.

In general the time of execution of an algorithm depends on the following factors:

- Input size.
- Instructions of the algorithm itself.
- The compiler used for the given programming language.
- Machine characteristics (CPU, RAM, operating system) on which the algorithm is executed.

Since the last two factors are varying from one user to the other (i.e. we cannot know characteristics of machine and compiler used by each user), we measure the complexity using only the two first factors.

2.2. Time Complexity Analysis

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. You might consider resources such as memory or energy consumption. Most often, however, you'll want to measure computational time [1].

Question: how to estimate the running time, without taking into consideration compiler and machine specifications?

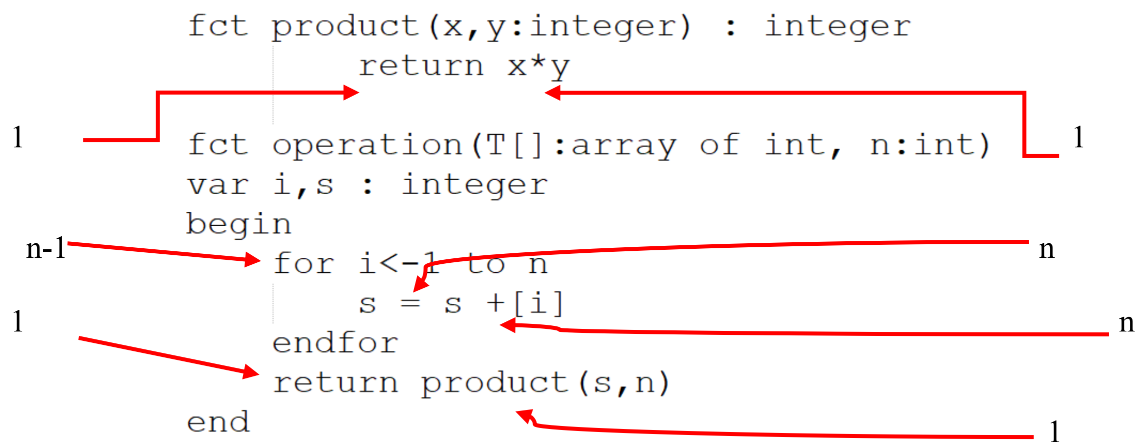
Answer: We suppose that the algorithm is executed in Random Access Machine where:

- Operations are executed in sequential order (machine has one single CPU, no parallelism or concurrency).
- Each basic operation (arithmetic operations, logic, relational, assignment …etc) takes one unit of time to execute.
- Calling a subprogram takes a constant time plus the time taken to execute instructions inside the body of the subprogram.
- The global running time is the sum of running time of all code blocks composing our program.

Note: Although it is often straightforward to analyze an algorithm in the RAM model, sometimes it can be quite a challenge. You might need to employ mathematical tools such as combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula [1].

Chapter 01 : Time and Space Complexity

Measuring execution time of an algorithm when run on a *particular* machine using a *particular* compiler with *particular* programs running in parallel in the machine, leads to value that cannot be adopted as a general measure. Instead, we assume that our program is run on a random access machine with an arbitrary input size of 'n'. In other words, we compute the cost of execution of each instruction times its frequency of execution (if this instruction is inside a loop). To demonstrate how these rules are followed, consider the example shown below:



$T(n) = 1 + n\text{-}1 + 1 + 1 + n + n + 1 = 3n + 3$

2.3.Best, average and worst case scenarios

The expression T(n) depends on the input size 'n'. If this expressions has different forms depending on the truth or falsity of some condition. T(n) is expressed in terms of best and worst case scenarios. We'll usually concentrate on finding only the worst-case running time because:

- This measure gives a guarantee that the algorithm never takes any longer.
- In practice, the probability of occurence of Worst-case scenario is high (take the search for inexistent data in a table in a database as an example).
- The average case is roughly as bad as the worst case.

2.4.Measure of order of growth of an algorithm

It would be difficult to compare different algorithms for each value of the input size 'n'. We study new notations that helps us to compare algorithms asymptotically, in other words, we compare the rate of growth of algorithms when the input size 'n' tends to infinity. If we have multiple algorithms for solving a given problem, the algorithm that takes the least time to execute when 'n' tends to infinity is considered to be the best solution. The new notations, helps knowing the upper, lower and tight bound of execution time of the algorithm.

2.5. Asymptotic notations (big-Oh, big-theta and big-omega)

The big-oh notation (O-notation) is the most frequently used notation to evaluate performances of an algorithm. This notation expresses **the asymptotic upper bound** of a function within a constant factor.

2.5.1. Definition

For a given function g(n), we denote O(g(n)) (pronounced big O of g of n) the set of functions f(n) for which there exist positives constants c and n_0 such that : $0 \leq f(n) \leq c.g(n)$ for all $n \geq n_0$,

In mathematical notation we can write, **$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c.g(n)$ for all $n \geq n_0\}$** c and n_0 are referred as multiplicative constant and a threshold [1].

Intuitively, $f(n) \in O(g(n))$ means that the function f(n) is growing at a lower rate compared to the function g(n). Figure shows that starting from the value n_0, f(n) is always less than c.g(n).

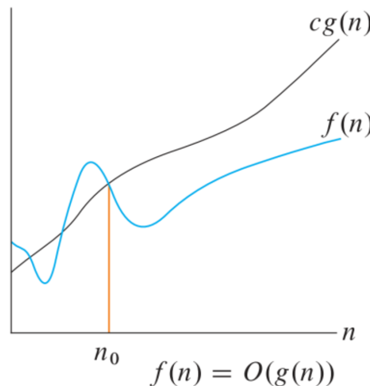


Figure 1 Graphical interpretation of Big-O Notation

Example: Let's consider some examples, taking $f(n) = 5n^2+400n+ 9$.

Although this function is higher than n^2, we can prove that $f(n) \in O(n^2)$

$f(n) = 5n^2 + 400n + 9$

$5n^2 + 400n + 9 \leq 5n^2 + n.n + n.n$ for all $n \geq 400$

$5n^2 + 400n + 9 \leq \mathbf{7}n^2$ for all $n \geq \mathbf{400}$

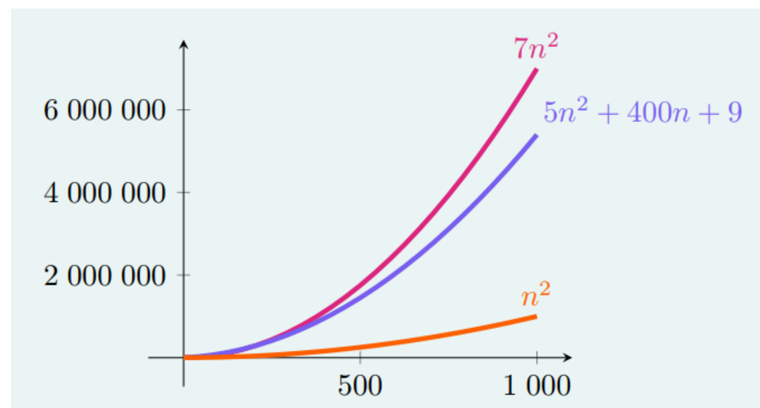


Figure 2 Comparison of three functions -example 1

Therefore, taking $c = 7$ as multiplicative constant et $n_0 = 400$ as a threshold, we conclude that $f \in O(n^2)$.

Example : $f(n) = 4n^2 + 100n + 500$, $f(n) \in O(n^2)$?

$4n^2 + 100n + 500 \leq cn^2$

If we divide both sides on n^2, we obtain

$4 + 100/n + 500/n^2 \leq c$

Multiple combinations of n_0 and c can be used to prove that the inequality is valid for all values of $n \geq n_0$. As examples:

$n_0 = 1 \quad c = 604$

$n_0 = 10 \quad c = 19$

$n_0 = 100 \quad c = 5.05$

Therefore, $f(n) \in O(n^2)$

Example : $f(n) = n^3 - 100n^2$, $f(n) \in O(n^2)$?

$n^3 - 100n \leq cn^2$

if we divide both sides on n^2, we obtain

$n - 100 \leq c$

For any value chosen for c, the inequality is not valid for values of $n > c + 100$, therefore, f(n) does not belong to $O(n^2)$.

2.5.2. General rules

In order to deduce the big O notation, we can follow the rules below:

- Multiplicative constants are omitted : $O(c.T) = O(T)$
- Addition is deduced by taking the biggest term :
  $O(T1) + O(T2) + O(T3) = \max[O(T1), O(T2), O(T3)]$
- Multiplication is not affected $O(T1)*O(T2) = O(T1*T2)$

Example : Calculate O(T(n)) for $T(n) = 7*2^n + 3n^3 + 6n^2 + 1000$

$T(n) = T1 + T2 + T3 + T4$

$O(T1)+O(T2)+O(T3)+O(T4) = O(T1+T2+T3+T4)$

$= \max[O(T1), O(T2), O(T3), O(T4)]$

$= O(T1)$
$= O(7x2^n)$
$= O(2^n)$

$T(n) \in O(2^n)$=> we say that the time complexity is in the order of $O(2^n)$

Example : Assuming the execution time of a given algorithm is given as $T(n) = 3n^2+10n+10$, How much is O(T(n))?

$$O(T(n)) = O(3n^2 + 10n + 10)$$

$$= O(\max (3n^2, 10n, 10))$$

$$= O(3n^2) = O(n^2)$$

Remark:

For n = 10 we have :

Contribution of the term $3\ n^2$: $3(10)^2 / 3(10)^2+10(10)+10$ = **73.2%**

Contribution of the term 10n : 10(10) / 3(10)2+10(10)+10 = **24.4%**

Contribution of the term 10 : 10 / 3(10)2+10(10)+10 = **2.4%**

The weight of $3n^2$ becomes bigger when n = 100, it becomes 96.7% and therefore we can ignore the quantities 10n and 10.

Example:

In the worst case scenario, the condition1 is not satisfied leading to execution of two nested loops, each for 'n' times. Therefore we only take the code leading to largest T(n).

$O(T1) + O(T2) = \max[O(T1), O(T2)]$

T(n) belongs to $O(n^2)$.

```
Var i, j : integer
begin
  if Condition1   then              }
    for i =1 to n do                } T1(n)=c.n+c'
       Instruction(s)               }
    End for

  else
    for i=1 to n do                 }
      for j=1 to n do               }
        Instruction(s)              } T2(n)=b.n²+b'.n + b''
      EndFor                        }
    EndFor                          }
  EndIf
end
```

2.6.Classes of complexity

2.6.1. Constant complexity O(1)

Running time does not increase when the input size of the problem increases.

Example:

```
function sum (a : integer, b : integer) : integer
begin
  return(a+b)  ⇚ 1 unit of time for addition.
               1  unit of time for returning.
end
```

Only two operations to execute (addition and returning), T(n)=2 => T(n) => T(n) α k where k is a constant => we say that the algorithm's time complexity is constant, O(1).

2.6.2. Logarithmic complexity O(log(n))

A slow increase of running time when the input size n is increased.

Example :

```
proc display(n : integer)
var i : integer
begin
        i <- 0
        while(i>0)
            print('i = ',i)
            i <- i/2
        endWhile
end
```

value of i after each iteration :

$$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \dots \dots, \quad \frac{n}{2^k} = 1$$

Or $n, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}, \dots \dots, \frac{n}{2^k} = 1$

K represent the number of iterations.

$$k = \frac{\log n}{\log 2} \Rightarrow O(\log(n))$$

2.6.3. Linear complexity O(n)

Linear increase of the running time with respect to increase of input size 'n'.

Example : An algorithm that fills an array.

```
procedure fill(T[] : array of integers, n :integer)
var i : integer
Begin
        For i = 0 to n do   ⇚ instruction executed n+2 times
           Print('insert element ',i) ⇚ instruction executed n+1 times
           Read(T[i]) ⇚ instruction executed n+1 times
        EndFor
End
```

T(n)= n+2 + n+1 + n+1 = 3n+4 => O(n)

2.6.4. Quasi-linear complexity O(n*log(n)) :

Example : Having two nested loops, the instructions inside the for loop will execute 'n' times, while the instructions inside the while loop will execute log (n) times as shown in the previous section.

```
proc display(n : integer)
var i,j : integer
begin
     for j<-0 to n step 1
         i <- 0
         while(i>0)
             print('i = ',i)
             i <- i/2
         endWhile
     EndFor
end
```

2.6.5. Quadratic complexity O(n²)

When the input size is doubled, the running time is multiplied by a factor of 4.

Example : filling a 2D table using two nested loops.

```
Example (filling a 2D array)
fct fill(T[][] : array of integers, n : integer)
var i,j :integer
Begin
   For i = 0 to n do  ⇚ instruction executed n+2 times
       For j = 0 to n do ⇚ instruction executed n+2 times
          Print('insert element i=',i ,'et j= ',j) ⇚ executed (n+2)*(n+2) times
          Read(T[i;j]) ⇚ executed (n+2)*(n+2) times
       EndFor
   EndFor
End
```

$T(n)= 2(n+2) + 2(n+2)(n+2) = \alpha n^2+\beta n+\gamma \quad => \quad O(n^2)$

2.6.6. Polynomial complexity $O(n^k)$

As an example, with might consider an algorithm containing k nested loops, each running approximately ‘n’ times.

2.6.7. Exponential Complexity $O(2^n)$

When the input size ‘n’ is doubled, the running time will be raised to power 2.

Example : The recursive solution to find the Fibonacci number of n (We will explain how to compute T(n) for recursive functions)

2.6.8. Factorial complexity O(n !)

Example : The famous Travelling Salesman Problem

The travelling salesman problem, is an optimization problem, where, given a number of cities, the shortest circuit has to be found and each city has to be visited only once.

In graph theory, a path is a sequence of vertices where each vertex is connected to the next by an edge, and no vertex or edge is repeated. A circuit is a type of path that starts and ends at the same vertex, with no edges being repeated.

For 7 cities, the starting point(a city) is selected. The salesman has to choose the second city to visit among 6 cities. The third city to be choosen among 5 cities, and we continue on the same manner.

- ⇨ We have 6x5x4x3x2x1 combinations or (7**-1**) !
- ⇨ Having this problem for only 4 cities, we notice that the circuit **A -> B -> C -> A** is the same as the circuit **A-> C -> B -> A** , therefore, for a problem with n cities we have (n-1) ! **/ 2**
- ⇨ For example for the TSP with 7 cities we have $(7-1)! / 2 = 360$ combinations

Tableau 1 Execution Time in terms of input size of factorial fct

2.0GHz CPU

2,000,000,000
calculation per second

To calculate all possible circuits composed of n cities is time consuming even if we use a normal computer whose CPU speed is 2 billion operations per second.

n	*n*!	Time (s,d,y)	
3	3! = 6	1.5e-9	seconds
5	5! = 120	6e-8	seconds
10	10! = 3628800	0.0018144	seconds
11	11! = 39916800	0.0199584	seconds
12	12! = 479001600	0.2395008	seconds
13	13! = 6227020800	3.1135104	seconds
14	14! = 87178291200	43.5891456	seconds
15	15! = 1.3076744e+12	653 s ≈ 10	minutes
16	16! = 2.092279e+13	10461 ≈ 2.9	hours
17	17! = 3.5568743e+14	177843 ≈ 2	days
18	18! = 6.4023737e+15	3201186 ≈ 37 days	
19	19! = 1.216451e+17	≈ 1.2	years
20	20! = 2.432902e+18	≈ 38	years

For large values of 'n', the running times of algorithms having different complexities will be very distinct.

Example : execution time of algorithms with different complexities for different values of n

Tableau 2 Execution Time in term of input size and diffrent complexities

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	O(1)	O(log *n*)	O(*n*)	O(*n* log *n*)	O(n^2)	O(n^3)	O(2^n)
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Figure below illustrates plots of functions 1,log(n), n, n*log(n), n^c , c^n, n! against different values of n.

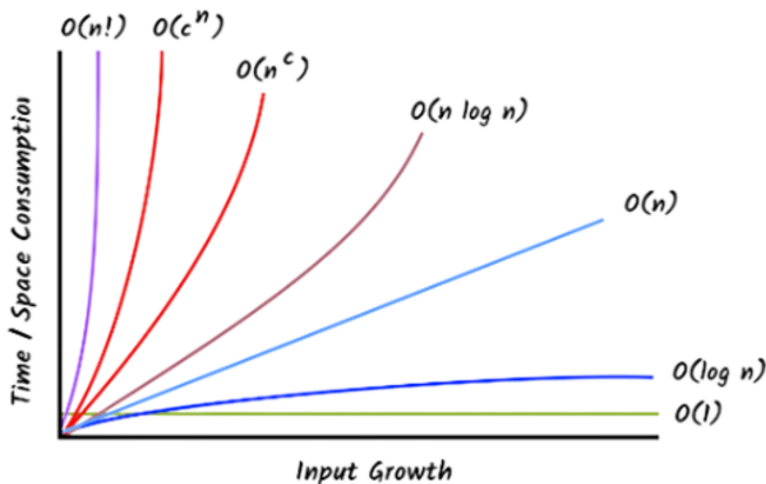


Figure 3 Graphical comparison between diffent fcts used in complexity classification

2.7. The big omega notation (Ω-notation)

While the O-notation gives us the asymptotic upper bound, the Ω-notation allow us to know the asymptotic lower bound.

2.7.1. Definition

We say that f(n) belongs to Ω(g(n)), if we can find constants c and n_0 such that : $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0$.

In other words, Ω(g(n)) is the set of functions f(n) that satisfies the above condition.

The intuition behind the Ω-notation is that for all values of n that are bigger than n_0, the value of f(n) is always bigger or equal to c.g(n).

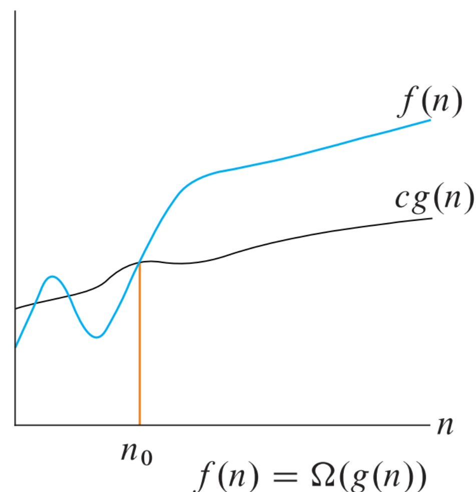


Figure 4 Graphical interpretation of big-Omega notation

Example : Given f(n) := n^2 - 400n + 5 that is shown in purple in figure below:

Even if f(n)is lower than n^2, f ∈ Ω(n^2).

$f(n) = n^2 - 400n + 5$

$\geq n^2 - 400n$

$\geq n^2 - \frac{n}{2}n$ For all n ≥ 800

$= \frac{1}{2}n^2$

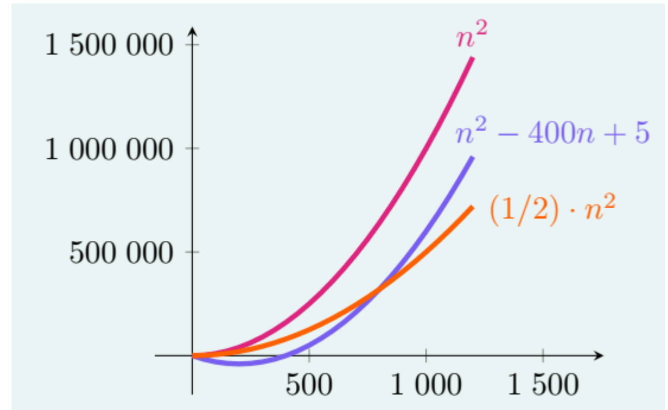


Figure 5 Graphical interpretation of Big-Omega an example

Taking c = 1/2 as a multiplicative constant and n_0= 800 as a threshold, we conclude that f ∈ Ω(n^2).

2.8.The big theta notation (Θ-notation)

2.8.1. Definition

A function f(n) belongs to the set Θ(g(n)) if positive constants c_1 and c_2 such that : $c_1g(n) \leq f(n) \leq c_2g(n)$, for highly sufficient values of n.

As Θ(g(n)) represent a set, we can write « f (n) ∈ Θ(g(n)) »

Figure on the right hand side, represents the geometrical interpretation of the Θ-notation. For all values of n that are greater of equal to n_0, the value of f(n) is greater or equal to c1g(n) and less than c2.g(n). We say that g(n) is **the asymptotic tight bound** of f(n).

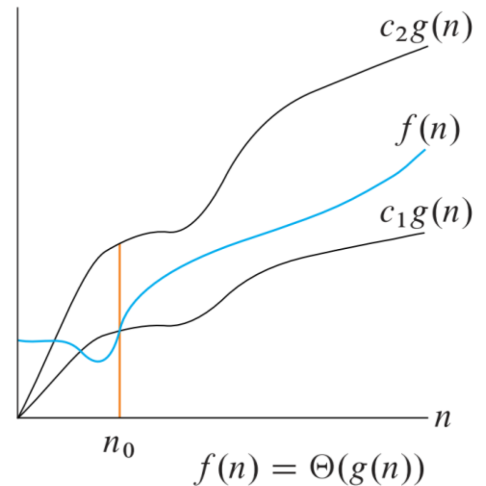


Figure 6 Graphical interpretation of big-Theta notation

2.8.2. Theorem

For any two functions f(n) and g(n), we have f(n)=Θ(g(n)) if and only if f(n) ∈ O(g(n)) and f(n) ∈ Ω(g(n))

Example f(n) = 42n+ 5

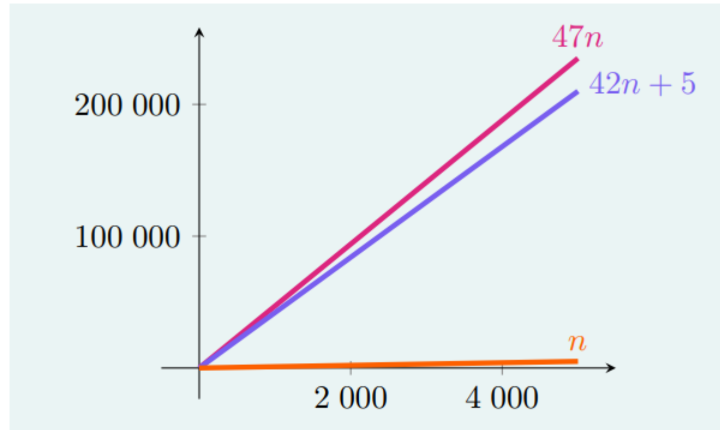


Figure 7Example of calculation of big-Theta

For higher values of n :

$$n \leq f(n) \leq 47n$$

f(n) ∈ Ω(g(n))

2.9. Time Complexity for Recursive Functions

Analyzing a recursive algorithm results in a recurrence equation of the general form

$$T(n) = \begin{cases} d & \textit{if base case} \\ T\left(\frac{n}{a} + b\right) + c & \textit{else} \end{cases}$$

Where parameters a, b, c and, d are real numbers.

The calculation of T(n) for recursive functions is done in various methods. The analytic method, where T(n) -in terms of n- is deduced from the formula of T(n).

Example : Let's compute T(n) for an algorithm that calculates the factorial of n, where n is a natural number.

Pseudocode	Time complexity

fct fact(n :integer) : integer
Begin
if n=0 then (1)
return 1 (1)
else
(1) return **fact(n-1)***n
End T(n-1) 1
	$\begin{cases} T(0) = C_1 \\ T(n) = T(n-1) + 4 \end{cases}$
	$T(n) = T(n-1) + C_2$ or $C_2=4$
	$T(n-2)= T(\mathbf{n-1}) + C_2$
	⇨ $T(n)=T(\mathbf{n-2})+\mathbf{2}C_2$
	⇨ $T(n)=T(\mathbf{n-3})+\mathbf{3}C_2$

	In general $T(n)=T(\mathbf{n-k})+\mathbf{k}*C$ When $n-k=0 \Rightarrow k=n$ $T(n)=T(0)+n*C \Rightarrow T(n)= C_2n+ C_1$ T(n) belongs to $\Theta(n)$

Example : Let's examine the six first values of Fibonacci

n	0	1	2	3	4	5	6
Fib(n)	0	1	1	2	3	5	8

The mathematical formula for calculating the Fibonacci number of n is :

$$Fib(n) = \begin{cases} n & if\ n = 0,1 \\ Fib(n-1) + Fib(n-2) & if\ n > 1 \end{cases}$$

Pseudocode

fct fib(n :integer) : integer
begin ← 1
if n <= 1 then
return n ← 1
else
return **fib(n-1)+ fib(n-2)**
T(n-1) 1 T(n-2)
End

<table>
<tr><td colspan="2">T(0)=0 and T(1)=1

T(n)=T(n-1)+ T(n-2)+C C=3</td></tr>
<tr><td>Given that T(n-1)> T(n-2) if we assume that T(n-2) = T(n-1)

$T(n) \simeq 2*T(\mathbf{n-1})+C$
$\simeq 2[2*T(\mathbf{n-2})+C]+C$
$\simeq 4T(n-2)+3C$
$\simeq 4[2*T(n-3)+C]+3*C$
$\simeq 8T(\mathbf{n-3})+7C$

In general</td><td>Given that T(n-1)> T(n-2) if we assume that T(n-1) = T(n-2)

$T(n) \simeq 2*T(\mathbf{n-2})+C$
$\simeq 2[2*T(\mathbf{n-4})+C]+C$
$\simeq 4T(n-4)+3C$
$\simeq 4[2*T(n-6)+C]+3*C$
$\simeq 8T(\mathbf{n-6})+7C$

In general</td></tr>
</table>

$T(n) \simeq 2^k T(n-\mathbf{k})+(2^k -1)*C$ When $n-k=0 \Rightarrow n=k$ $T(n) \simeq 2^n T(0)+(2^n-1)*C$ $\simeq 2^n[T(0)+C]-C$ $T(0)=0$ et $C=3$ $T(n) \simeq 3*2^n -3$	$T(n) \simeq 2^k T(n-\mathbf{2*k})+(2^k -1)*C$ When $n-2k=0 \Rightarrow k=n/2$ $T(n) > 2^{n/2} T(0)+(2^{n/2}-1)*C$ $> 2^{n/2}[T(0)+C]-C$ $T(0)=0$ et $C=3$ $T(n) \simeq 3*2^{n/2} -3$

$\Rightarrow 3*2^{n/2} -3 < T(n) < 3*2^n -3$ Therefore T(n) belongs to $\mathbf{\Theta(2^n)}$

2.10. Calculation of space complexity

2.10.1. For non-recursive algorithms

We measure the space allocated for variables and function calls in terms of the input size n.

When calling the 'display' function, only two variables are created in memory (i and n) therefore, we say that the space complexity is constant.

```
proc display(n : integer)
var i : integer
begin
        i <- 0
        while(i>0)
            print('i = ',i)
            i <- i/2
        endWhile
end
```

2.10.2. For recursive algorithms

We draw recursion trees for some values of n. then try to deduce the relationship between the number of function calls and the input size n.

Draw the recursion tree of Fibonacci when n = 5, deduce complexity.

To reach the base case in the longest branch, 5 function calls are performed.

For n = 5 we perform 5 calls.

For n = 10 we perform 10 calls.

In general for a given value of n, we perform n calls, therefore S(n) = n or S(n) ∈ Θ(n)

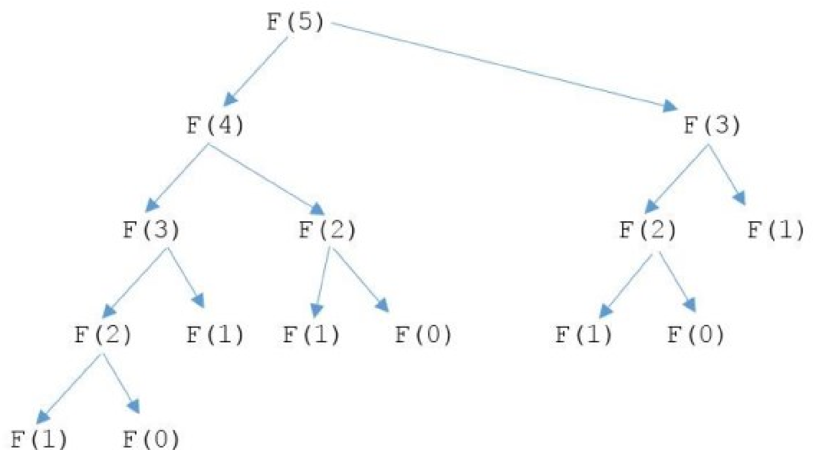


Figure 8 Recursion Tree of Fibonacci fct with input size n=5

2.11. Conclusion

Time and Space complexities are two powerful tools that helps the programmer to evaluate different solutions of a given problem. Further details about the complexity analysis are exposed in the next chapter, where we study different sorting algorithm and their corresponding complexities.

2.12. Exercises

1) Design an algorithm that calculates the sum s, where s = (1 + 2 + 3 + …….. + n)/5 and the time complexity should be constant T(n) belongs to Θ(1).
2) Give the possible algorithms for finding the prime number of a given natural number.
3) Calculate the time and space complexities of the following algorithms

(a)

```
int function1(int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + function1(n-1);}
```

(b)

```
int function2(int n)
{   if (n <= 0)
        return 1;
    else
        return 1 + function2(n-5);}
```

(c)

```
int function3(int n)
{   if (n <= 0)
        return 1;
    else
        return 1 + function3(n/5);}
```

(d)

```
int function4(int n)
{  for(int i=0; i<n; i++)
   { function4(n-1);}
}
```

(e)

```
i ← n;
S ← 0;
Tant que (i > 0) faire
    j ← 2*i;
    Tant que (j > 1) faire
        S ← S+(j-i)* (S+1);
        j ← j-1;
    Fin TQ;
    i ← i div 2;
Fin TQ;
```

(f)

```
i ← 1;
j ← 0;
Pour k de 1 à n faire
    j ← i+j;
    i ← j-i;
Fin Pour;
```

Chapter 02 : Sorting Algorithms

3. Chapter 02: Sorting algorithms

3.1.Introduction

Sorting a sequence of elements consists of ordering them according to a given **key**. For example, assuming that we have a collection of data on students, where each element of this collection represents pieces of information (full name, age, average) of a certain student. We can sort this collection according to one of the student attributes, the selected attribute for sorting is referred as **key** while the rest of attributes are called **satellite data**. Key with satellite data forms a **record.**

Operating on sorted elements helps improving the running time of other operations like the search operation. In addition, studying sorting algorithms helps better understand the concept of time and space complexities.

3.2.Characterization of sorting algorithms

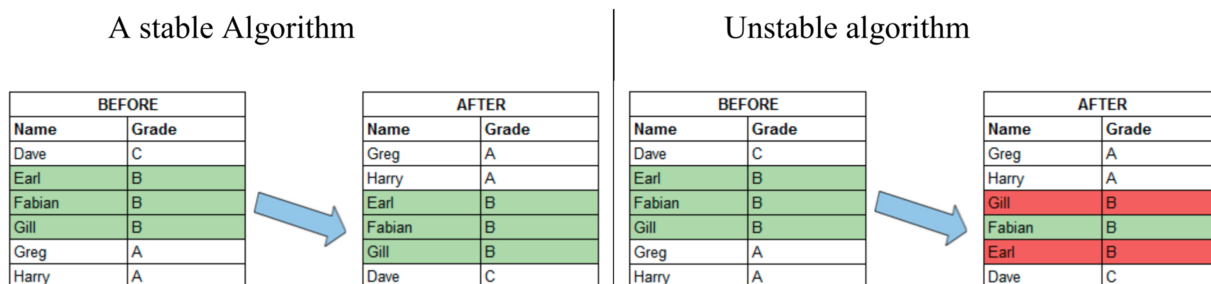
We say a sorting algorithm is:

- **In place**, when this algorithm use a constant size of memory during its execution (in addition to the space allocated for storing the sequence to be sorted).
- **By comparison**, when this algorithm involves comparison of elements as one of the steps.
- **Stable** : A stable sorting algorithm respects the order of elements that have equal keys. This property is useful when we want to sort elements multiple times, each time according to a different key. For example, having a list of students, if we sort it according to names of students and in the second time according to their average. We want students having same average to be sorted according to their name.

Example : We have a list of students sorted according to the alphabetic order of their names. If we sort this list according to their grades, we have two possibilities.

A stable Algorithm

Unstable algorithm



A stable Algorithm — BEFORE

Name	Grade
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

A stable Algorithm — AFTER

Name	Grade
Greg	A
Harry	A
Earl	B
Fabian	B
Gill	B
Dave	C

Unstable algorithm — BEFORE

Name	Grade
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

Unstable algorithm — AFTER

Name	Grade
Greg	A
Harry	A
Gill	B
Fabian	B
Earl	B
Dave	C

3.3.Selection sort

3.3.1. Selection sort algorithm

Given a collection T of 'n' elements, where each element have several attributes. Having selected one of the attributes as a key. Selection sort consist of finding the element with the smallest key value and then swapping it with the first element in the collection. Process is repeated with the new collection T' which is the old collection T without the first element. The iteration is stopped when the new collection is composed of only one element.

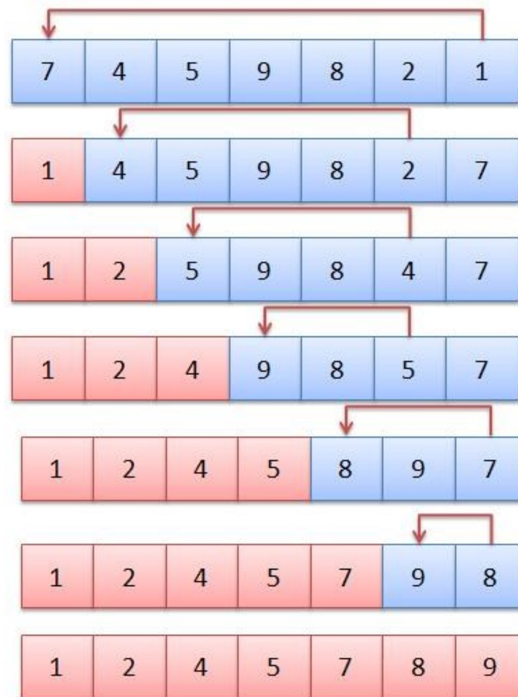


Figure 1Demonstration of selection sort algorithm

3.3.2. Pseudocode of selection sort algorithm

```
PROCEDURE SelectionSort(T[0..n-1] :array of int, n :int)
    j, i, imin, temp : integers
    begin
      for i ← 0 to n-2 do
           imin ← i;
           for j ← i+1 to n-1 do
              if (T[j] < T[imin]) then
                 imin ← j;
              EndIf
           EndFor
           temp ← T[i];
           T[i] ← T[imin];
           T[imin] ← temp;
      EndFor
   end
```

3.3.3. Selection sort time complexity calculation

```
PROCEDURE SelectionSort(T[0..n-1] :array of int, n :integer)
    j, i, imin, temp : integer
    begin;
      For i ← 0 to n-2 do
          imin ← i;                          ← C1*(n-1)
          for j ← i+1 to n-1 do
             if (T[j] < T[imin]) then   ┐
                imin ← j;               ├ C2*k
             endIf                      ┘
          endIf
          temp ← T[i];                  ┐
          T[i] ← T[imin];               ├ C3*(n-1)
          T[imin] ← temp;               ┘
      EndIf
   End
```

Question : How to calculate the k value which represent the total number of iterations of the inner for loop ?

For the 1st iteration of the outer loop, the inner loop will be executed n-1 times.

For the 2nd iteration of the outer loop, the inner loop will be executed n-2 times.

…….

For the (n-2)th iteration of the outer loop, the inner loop will be executed 1 time.

Therefore, $K = (n-1) + (n-2) + \ldots + 1 = (n-1)*n/2$

$T(n) = (n-1)*(C_1 + C_3) + C_2(n-1)*n/2$

$= \alpha*n^2 + \beta*n + \lambda$

Even if the algorithm is given an already sorted collection to sort, it will take the same amount of time for the algorithm to sort an unsorted collection ☹. We say that in all cases, $T(n) \in \Theta(n^2)$ (Time complexity is quadratic)

For the execution of the selection sort algorithm, a constant number of variables has to be declared, therefore, we say the selection sort algorithm is an in-place algorithm.

Referring back to the pseudocode, we can notice that selection sort, at a given moment compares between two elements. We say that the selection sort algorithm is a by comparison sorting algorithm.

Assuming that the letters below are already sorted according to the color key. If we want to sort this collection of colored letters according to their alphabetical order using the selection sort algorithm:

Iteration number	Items to be sorted			
1	D	M	D	C
2	C	M	D	D
3	C	D	M	D
4	C	D	D	M

At the beginning the D appears before the D. At the end of sorting the D moves after the D. => The classical version of selection sort is unstable.

3.4.Bubble sort

3.4.1. Bubble sort algorithm

Given a collection T of n elements. For sake of simplicity, we assume that these elements has only one attribute and therefore can be stored in an array. Steps of bubble sort algorithm are as follows:

- Compare element T[0] with the element T[1]. If element T[0] is greater than T[1], we swap those two elements. We repeat the same process with the pairs {T[1] and T[2]} and {T[2] and T[3]} , till the pair { T[n-2] and T[n-1]}. Eventually, the element with the highest key value will be placed as the last element in the array.
- We repeat same steps with the new array which consists of the old array with the last element removed.
- We stop the iteration when the newly obtained table is composed of just one element.

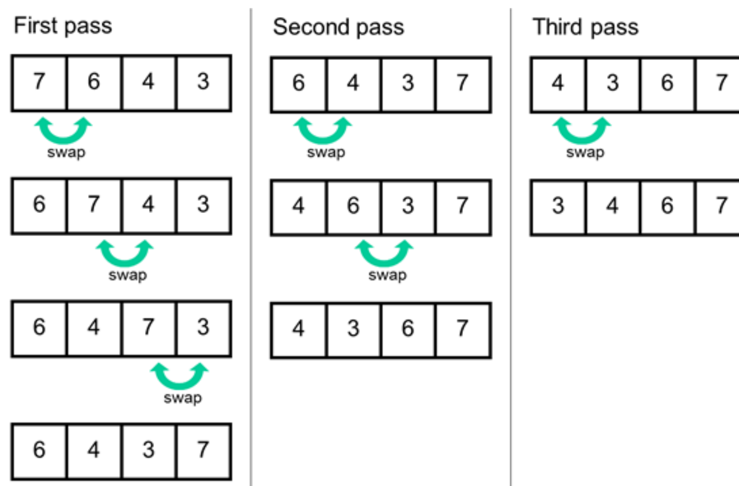


Figure 2Graphic demonstration of bubble sort algorithm

3.4.2. The pseudocode of bubble sort algorithm

```
PROCEDURE BubbleSort(T[0..n-1] :array of int, n :integer)
    j, i, temp : integers
    begin;
      for i ← 1 to n-1 do
           for j ← 0 to n-2 do
              if (T[j] > T[j+1]) then
                 temp ← T[j]
                 T[j] ← T[j+1]
                 T[j+1] ← temp
              EndIf
           EndFor
      EndFor
   end
```

3.4.3. Bubble sort algorithm time complexity

```
PROCEDURE BubbleSort(T[0..n-1] :array of int, n :int)
    j, i, temp : integers                                  n-1
    begin;
      for i ← 1 to     n-1   do                            n-1
           for j ← 0 to     n-2 do
              if (T[j] > T[j+1]) then    }
                 temp ← T[j]             }
                 T[j] ← T[j+1]           }   C1
                 T[j+1] ← temp           }
              endIf
           EndFor
      EndFor
   fin
```

$T(n) = C_1*(n-1)*(n-1)$

$= \alpha*n^2 + \beta*n + \lambda$

Even if the algorithm is supplied with an already sorted list to sort, it will take same time for that algorithm to sort an unsorted list, therefore, in all cases $T(n) \in \Theta(n^2)$ (Time complexity of the classical bubble sort is quadratic)

For a list of any size n, a constant number of variables are created in the memory during the execution of bubble sort algorithm. Therefore we can say that the bubble sort algorithm is an 'in-place' one.

During the execution of bubble sort algorithm, comparison between elements has to be performed at each iteration of the outer loop. Hence, the bubble sort algorithm is a by comparison sorting algorithm.

3.4.4. Improvement of bubble sort algorithm

Even if the supplied list to be sorted by the bubble sort algorithm is already sorted, this algorithm takes almost the same amount of time required for sorting an unsorted list !

If the bubble sort algorithm is supplied with an already sorted list to sort, in that case, the condition (if (T[j] > T[j+1])) inside the algorithm will be always false. Therefore, we can assume at the beginning of each iteration of the outer loop that the list is sorted. If the list is unsorted, the condition will not be satisfied and our assumption will be wrong. In that regard, we use a Boolean variable is_sorted to know when the list is already sorted and therefore use it to stop the iteration.

```
PROCEDURE bubbleSort(T[0..n-1] :array of int, n :int)
    j, i, temp : int
    is_sorted : boolean
    begin;
      for i ← 1 to n-1 do
           is_sorted ← true
           for j ← 0 to n-2 to
              if (T[j] > T[j+1]) then
                 temp ← T[j]
                 T[j] ← T[j+1]
                 T[j+1] ← T[j+1]
                 is_sorted ← false
              EndIf
            EndFor
            if (is_sorted = true) then
                 break
            Endif
      EndFor
   fin
```

In the best case (When the list is already sort), the time complexity for bubble sort is linear ($T(n) \in \Theta(n)$). Otherwise, for all other cases, the time complexity is quadratic ($T(n) \in \Theta(n^2)$).

To check if the bubble sort is stable or not, we assume that the colored letters has been already sorted according to their colors. Then, we try to sort the list of letters according to their alphabetical order.

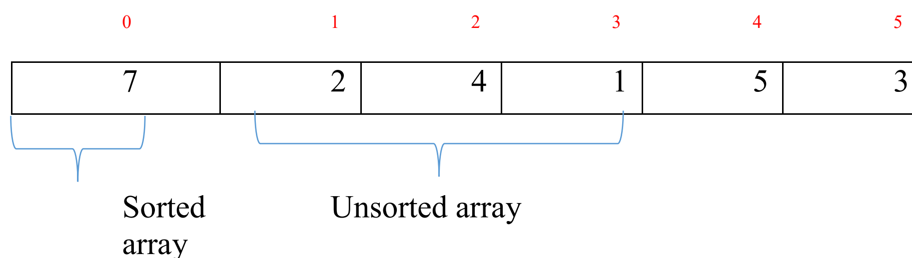
D	M	D	C
D	D	C	M
D	C	D	M
C	D	D	M

At the beginning the D was appearing before D. After sorting, the D remains always before the D. We conclude that the bubble sort algorithm is stable.

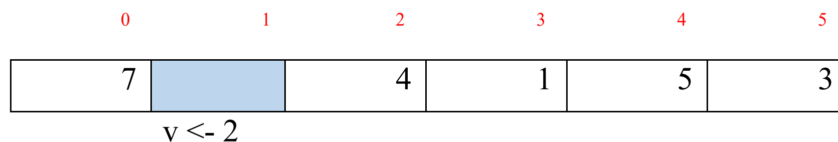
3.5. Insertion sort

3.5.1. Insertion Sort algorithm

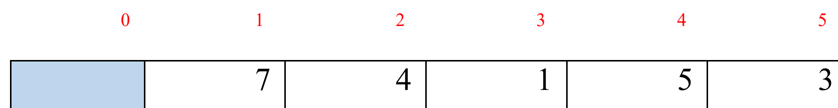
For sake of simplicity, we assume that the list that we want to sort is composed of elements with one single attributes. Therefore, this attribute is the sorting key and this list can be stored in form of an array. We can view this array as two separate arrays. The first array, a sorted one and that is composed, initially, of only the first element. The second array, an unsorted array that composed of the remaining elements.



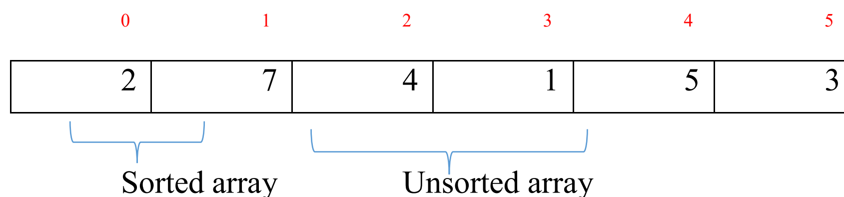
- We store the first element in the unsorted array in a variable v.



- In the sorted array, We shift to the right all the elements that are larger than the value stored in v.



- We insert the value of v in the place empty cell in the array.



At the end of the first iteration, we obtain a sorted array composed of 2 first elements and an unsorted array composed of n-2 elements.

We repeat the same process n-1 times to obtain a completely sorted list.

3.5.2. Pseudocode of insertion sort

```
PROCEDURE InsertionSort(A[0..n-1] :array of int, n :int)
    i, i_hole, v : integers
    begin;
      for i ← 1 to n-1
           v ← T[i]
           i_hole ← i
           while (i_hole > 0 AND A[i_hole - 1] > v)
               A[i_hole] ← A[i_hole -1]
               i_hole ← i_hole - 1
           endWhile
           T[i_hole] ← v
      EndFor
   end
```

3.5.3. Time complexity Analysis of insertion sort

```
PROCEDURE InsertionSort(T[0..n-1] :array of int, n :int)
    i, i_hole, temp : integers
    begin;
      for i ← 1 to n-1 do
           v ← T[i]         }  ——  c1*(n-1)
           i_hole ← i       }
           while (i_hole > 0 ET A[i_hole - 1] > v)
               A[i_hole] ← A[i_hole -1]  }  ——  c2*K
               i_hole ← i_hole - 1       }
           endWhile
           T[i_hole] ← v  ←———————  c3*(n-1)
      EndFor
   end
```

In the best case scenario (When the array is already sorted), each element is larger than the elements that are on its left. Therefore, the while loop will never execute. In other words, k=0 in this case => $T(n) = \alpha n + \beta \Rightarrow T(n) \in \Theta(n)$

In the worst case scenario (when the array is reverse ordered), meaning that each element is less than all the elements on its left:

For the first iteration of the for loop, The while loop will run 1 time.

For the second iteration of the for loop, the while loop will run 2 times.

…….

For the $(n-1)^{th}$ iteration of the for loop, the while loop will be executed n-1 times.

K = 1 + 2 + ……. + (n-2)+(n-1)

=(n-1)*n/2

T(n) = c1*(n-1)+ c2*(n-1)*n/2+ c3*(n-1) => In the worst case, The time complexity is in the order of $\Theta(n^2)$ (big theta n squared)

	Time Complexity
Best case (When the array is already sorted)	$\Theta(n)$
Worst case (When the array is reverse sorted)	$\Theta(n^2)$
average case (all other cases.)	$\Theta(n^2)$

Assuming that the colored letters are sorted at first, according to their colors, if we try to sort the list according to the alphabetical order of the letters we obtain :

D	M	D	C
D	D	C	M
D	C	D	M
C	D	D	M

At the begining D was appearing before D. after sorting with insertion sort D remains always before D. => insertion sort is a stable sorting algorithm.

For sorting an array with a n number of elements, we need additional constant space for storing a defined number of variables, in other words, the insertion sort is an *in-place algorithm*. (auxiliary space complexity is constant)

In addition, for sorting a list using insertion sort algorithm, at an intermediary step, we need to compare elements to each other, hence, insertion sort is a *by comparison algorithm*.

3.6.Binary Search (Or bisection search)

Given a sort array T of n elements.

Indice	0	1	2	3	4	5	6	7	8	9	10
T[indice]	3	4	6	9	11	13	15	18	23	24	26

Write an efficient algorithm that look for value val in the array, return its index if it exist in the array, return -1 otherwise.

In general, checking a value whether it exists or not in an unsorted array, requires scanning the entire array (element by element), resulting in algorithm whose time complexity is linear. But in our case the array is already sorted, we can exploit this fact, to suggest a faster running algorithm.

3.6.1. The binary search algorithm

Given a sorted array of size n :

- We check if the value we are looking for is in the middle of the table. If it exists in the middle, we stop since we found the value in our array.
- Otherwise if our value is less than the value in the middle, then it is less than all values of elements at the right of the middle table, so we should look for that value if it exists in the table at the left of the middle element.
- Otherwise if our value is greater than the value in the middle, then it is greater than all values of elements at the left of the middle table, so we should look for that value if it exists in the table at the right of the middle element.
- The search space is halved at each iteration, eventually reaching a case where the search space is an array composed of one single element.

Since the array is sorted, telling whether the value 20 exists in the array or not, will take only 4 iterations instead of 15 iterations that is required for scanning the entire array using linear seach, if 20 does not exist in the table.

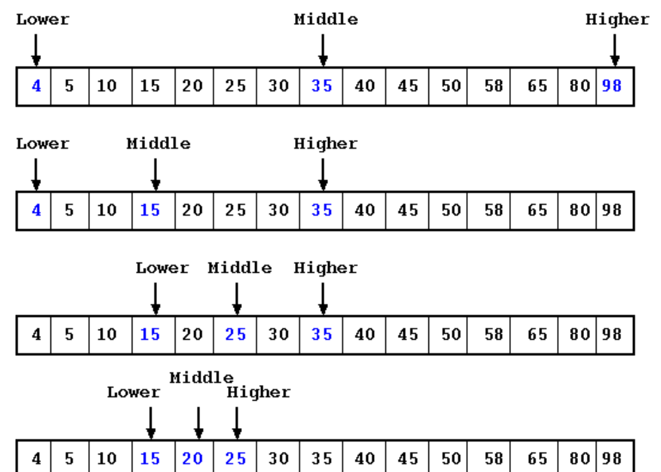


Figure 3Mechnism of Binary Search

3.6.2. Pseudocode of binary search (iterative solution)

```
func binary_search(T[] :array of int, n :int, val :int) :int
Var min, max, mid : integers
Début
    mini <- 0
    maxi <- n - 1
    do
      mid <- (mini + maxi) div 2
      if ( val < T[mid]) then
          maxi <- mid - 1
      else if (val > T[mid]) then
          mini <- mid +1
      else
          return mid
      FinSi
    while ( mini < maxi )
    return -1
Fin
```

The number of elements at each iteration of loop :

$$n \; then \; \frac{n}{2} \; then \; \frac{n}{4} \; then \; \frac{n}{8} \; then \; \frac{n}{16}, \dots \dots, \qquad \frac{n}{2^k} = 1$$

Written in terms of powers of 2 :

$$n, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}, \dots \dots, \qquad \frac{n}{2^k} = 1$$

K here represents the number of iterations of while loop.

$k = \frac{\log n}{\log 2}$ => T(n) belongs to $\Theta(\log(n))$

Notice that the same algorithm can be re-written in a recursive form.

3.7.Merge sort

Merge sort belongs to the category of divide and conquer algorithms, meaning that we decompose the problem to a small sub-problems which are easy to solve. Then, we solve each sub-problem and combine the solutions to obtain the solution of the bigger problem.

3.7.1. The merge sort algorithm

for sake of simplicity, we will assume that the ‘n’ elements we are seeking to sort are stored in one-dimensional array. Recursively, we perform the following operations :

We divide the array to two equal size sub-arrays or almost equal(i.e. almost equal is when the size of the array is an odd number).

we sort each sub array.

we merge the two sub-arrays to obtain a sorted array.

Recursion stops when we reach sub-arrays composed only of one single element.

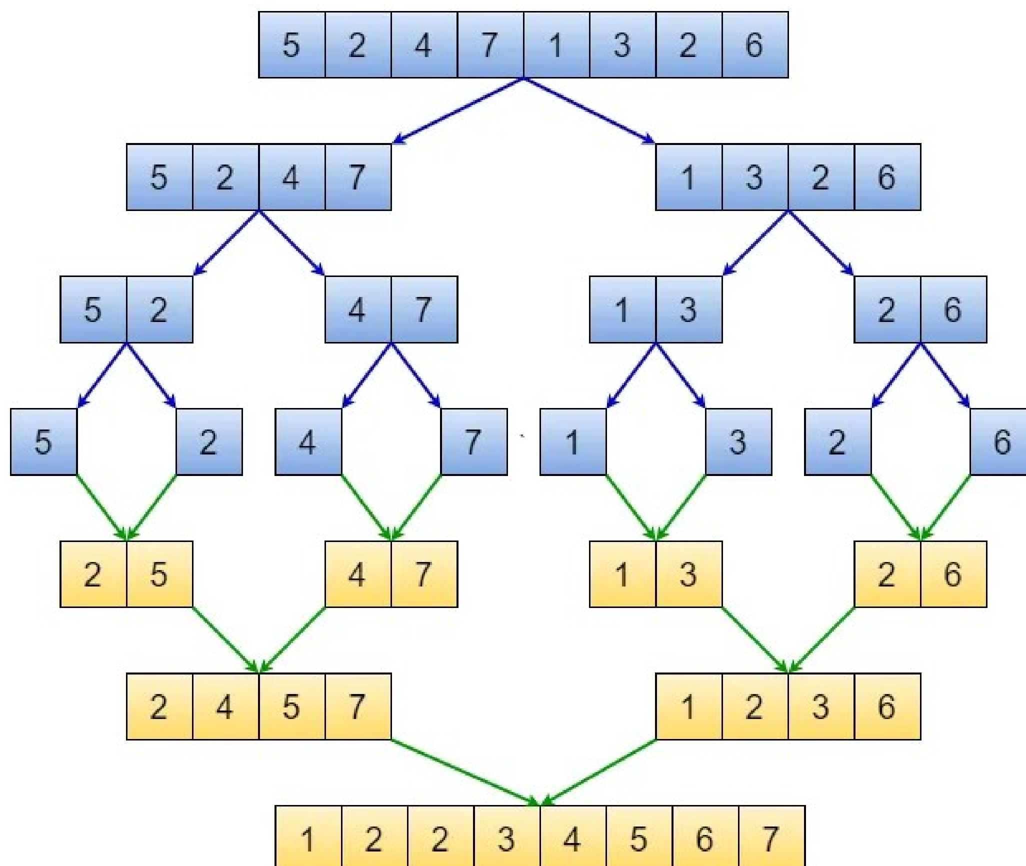
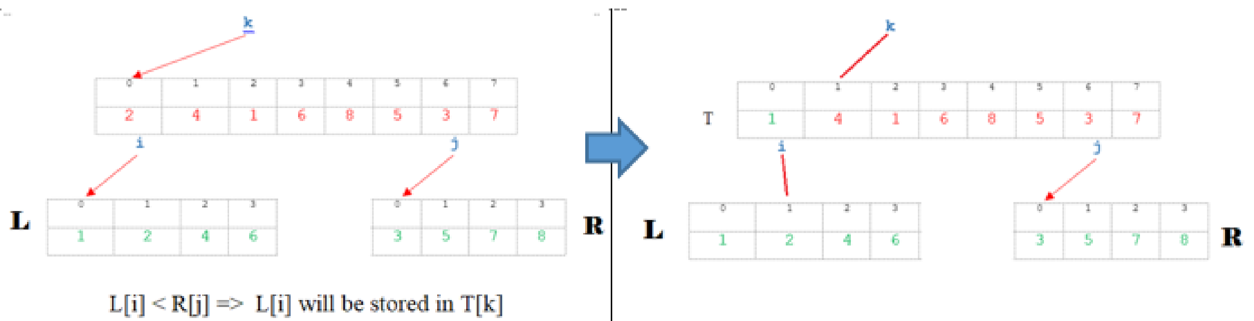


Figure 4 Steps of Merge sort

3.7.2. Merge sort Algorithm

Suppose we have already sorted two sub-arrays. How can we merge this two sub-arrays into one single sorted array?

- If L[i] < R[j] , store L[i] in T[k], otherwise , store R[j] in T[k]
- Increment the index with the smallest value (i or j)
- If one of the indices i or j exceed the size of its related array, fill the remaining elements of the other array directly in the array T.



L[i] < R[j] => L[i] will be stored in T[k]

3.7.2.1. Merge algorithm

```
PROC Merge(T[],n , Left[], nL, Right[], nR : integers)
    i, j, k : integers
Begin
      i ← j ← k ← 0
      While( i < nL AND j < nR ) do
             if( Left[i] < Right[j]) then
                T[k] ←   Left[i]
                i ← i + 1
                k ← k + 1
             else
                T[k] ←   Right[j]
                j ← j + 1
                k ← k + 1
             EndIf
       EndWhile

       While(i <  nL ) do
                T[k] ←   Left[i]
                i ← i + 1
                k ← k + 1
       finTanque

       while(j < nR ) do
                T[k] ←   Right[j]
                j ← j + 1
                k ← k + 1
       endWhile

End
```

Will not be executed only if all elements of the right sub-array are stored in the array T, in that case, we move the remaining elements in the left sub-array directly in the array T.

Will not be executed only if all elements of the left sub-array are stored in the array T, in that case, we move the remaining elements in the right sub-array directly in the array T.

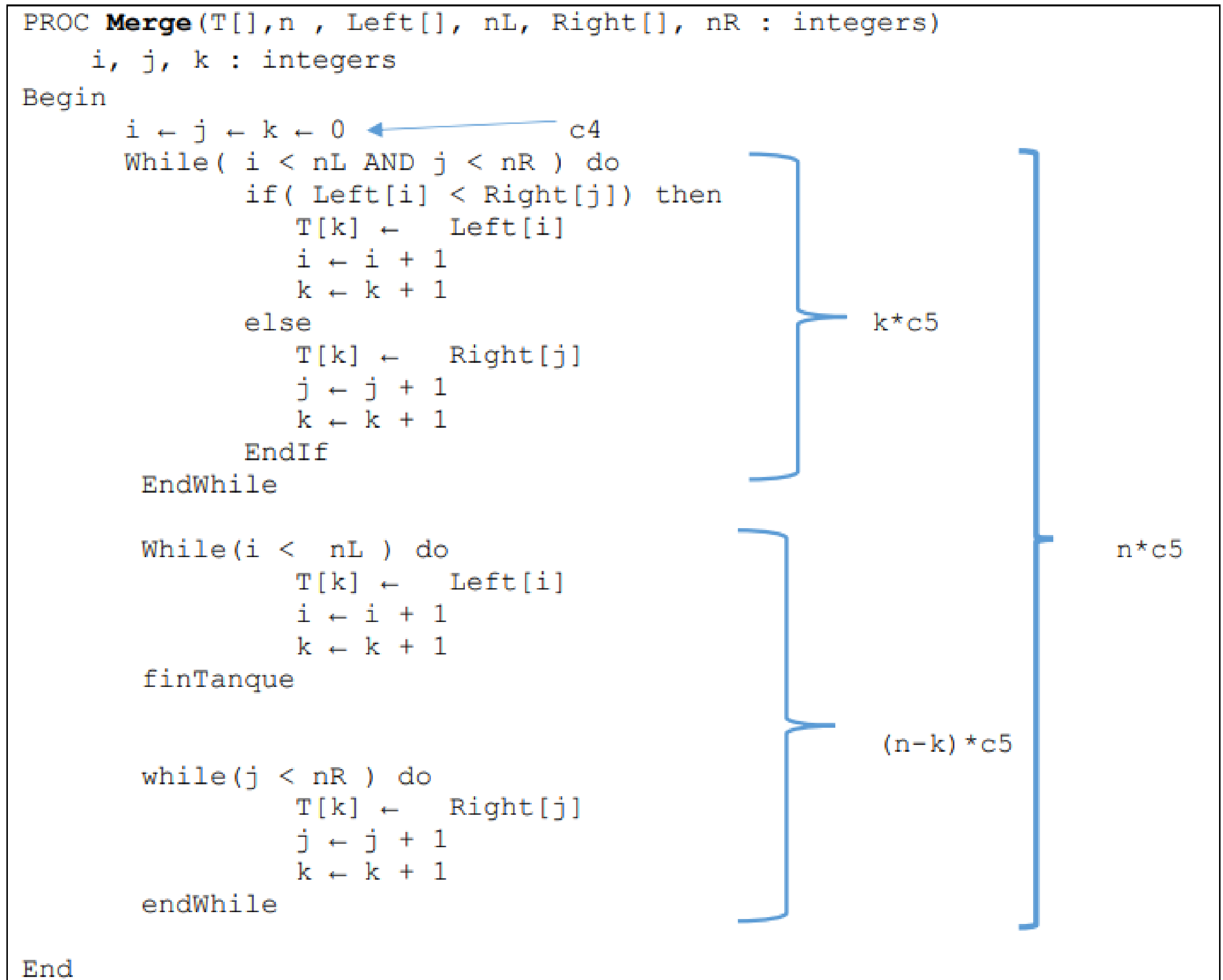
```
PROCEDURE MergeSort(T[]:, n : integers)
    Left, Right : array of integers
Begin
    if (n < 2) then
        return
    Endif
    mid <- n/2
    Left[mid], Right [n- mid]: integers

    for i<-0 to mid-1
        Left[i] <- T[i]
    EndFor

    For j <- mid to n-1
        Right[j- mid] <- T[j]
    EndFor

    MergeSort (Left, mid)
    MergeSort (Right, n- mid)
    Merge(T, Left, mid, Right, n- mid)
End
```

Decomposition of the array T into two subarrays. Then filling the two sub-arrays will elements of the array T

3.7.3. Time complexity analysis of Merge sort

```
PROCEDURE MergeSort(T[]:, n : integers)
    Left, Right : array of integers
Begin
    if (n < 2) then
        return
    Endif
    mid <- n/2                                    } c0
    Left[mid], Right [n- mid]: integers

    for i<-0 to mid-1
        Left[i] <- T[i]
    EndFor
                                                  } n*c1
    For j <- mid to n-1
        Right[j- mid] <- T[j]
    EndFor

    MergeSort (Left, mid)  <-----------  T(n/2)
    MergeSort (Right, n- mid)  <-----------  T(n/2)
    Merge(T, Left, mid, Right, n- mid)  <-----------  K
End
```

$T(n) = c_0 + n^* c_1 + 2^*T(n/2) + K$ for $n > 1$



```
PROC Merge(T[],n , Left[], nL, Right[], nR : integers)
    i, j, k : integers
Begin
      i ← j ← k ← 0
      While( i < nL AND j < nR ) do
             if( Left[i] < Right[j]) then
                T[k] ←   Left[i]
                i ← i + 1
                k ← k + 1
             else
                T[k] ←   Right[j]
                j ← j + 1
                k ← k + 1
             EndIf
       EndWhile

       While(i <  nL ) do
                T[k] ←   Left[i]
                i ← i + 1
                k ← k + 1
       finTanque

       while(j < nR ) do
                T[k] ←   Right[j]
                j ← j + 1
                k ← k + 1
       endWhile

End
```


K = c_4+c_5*n

$$T(n) = \begin{cases} c1 & n = 1 \\ 2 * T(n/2) + c2 * n \end{cases}$$

For n > 1 :

$T(n) = 2*T(n/2) + c_2*n$ and

$= 2*[2*T(n/4) + c_2*(n/2)] + c_2*n$

$= 4*T(n/4) + 2*c_2*n$

$= 4*[2*T(n/8)+ c_2*(n/4)] + 2*c_2*n$

$= 8*T(n/8)+ 3*c_2*n$

In general :

$= 2^K*T(n/2^K)+ K*c_2*n$

When $n/2^K = 1 \Rightarrow k = \log_2(n)$

$T(n) = 2^{\log_2(n)} * T(1) + \log_2(n) * c_2 * n$

$= n * T(1) + \log_2(n) * c_2 * n$

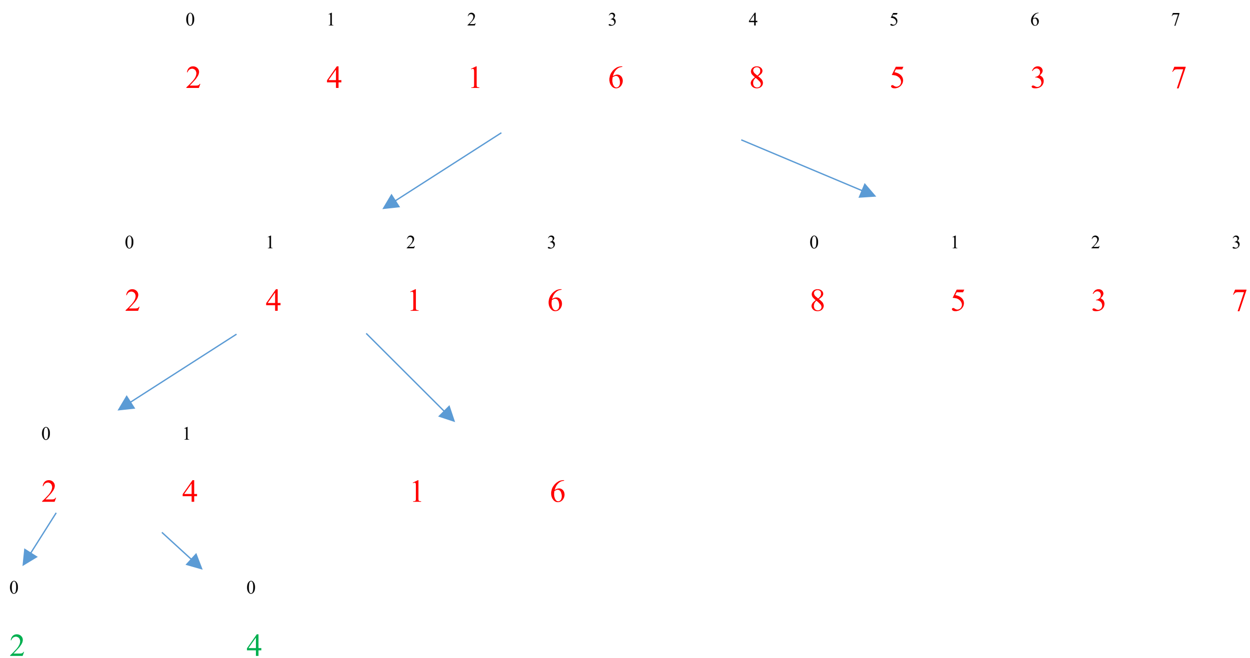
$= n * c_1 + \log_2(n) * c_2 * n$

For $n \to \infty \quad n * \log(n) > n$

In all cases, the time complexity is : $\Theta(n*\log(n))$

In order to sort a sequence using merge sort algorithm, many other sub-arrays will be created in memory, therefore, merge sort is not an in place sorting algorithm.

Since the merge sort is a recursive algorithm, we can estimate its space complexity by drawing the recursion tree and then try to deduce the relationship between the input size 'n' and the number of calls of this function in the longest branch. We notice that the small arrays containing elements 2 and 4 will be removed from memory before the creation of two sub-arrays each containing elements 1 and 6.



In general, to sort an array of size n, the arrays with the following size will be created in memory.

$n, n/2, n/4, \ldots\ldots, n/2^K$

The sum of the elements stored in this array represent the maximum amount of memory consumed during the execution of the merge sort.

$S(n) = n + n/2 + n/4 + \ldots\ldots \ , n/2^K$

$= n * (1 + ½ + ¼ + \ldots\ldots + n/2^K)$

$= n * c$

Where c is a geomertrical sequence, therefore, the sum of these terms is :

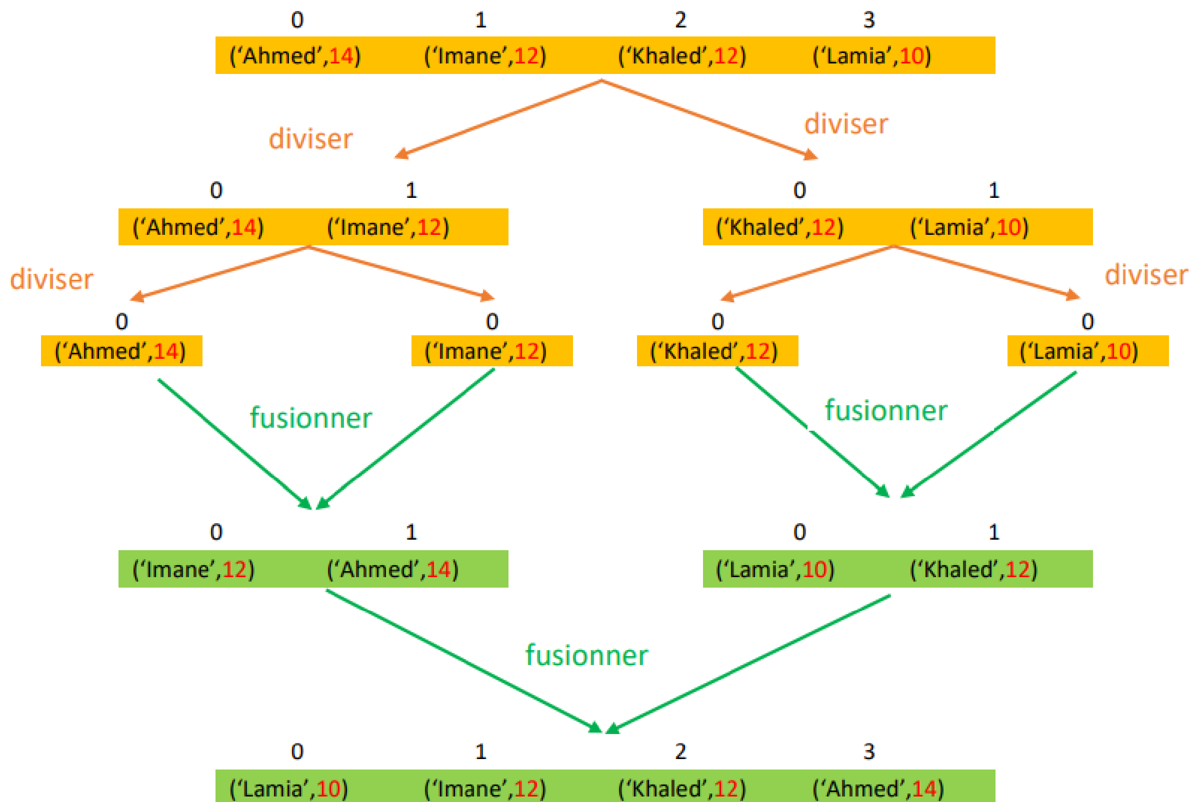
$c = (1 + ½ + ¼ + \ldots\ldots) \ = \frac{1-\frac{1}{2}^k}{1-\frac{1}{2}}$

$0 \leq k < +\infty \ \Rightarrow \ 1 \leq c < 2 \ \Rightarrow \ 1*n \leq S(n) < 2*n$

=> Space complexity of the merge sort algorithm is Θ (n).

3.7.4. Stable or instable ?

Given a collection of elements, each with two attributes (name, grade). This collection is already sort according the attribute name (i.e. name is the sorting key). If we sort this collection according to the key grade using merge sort :

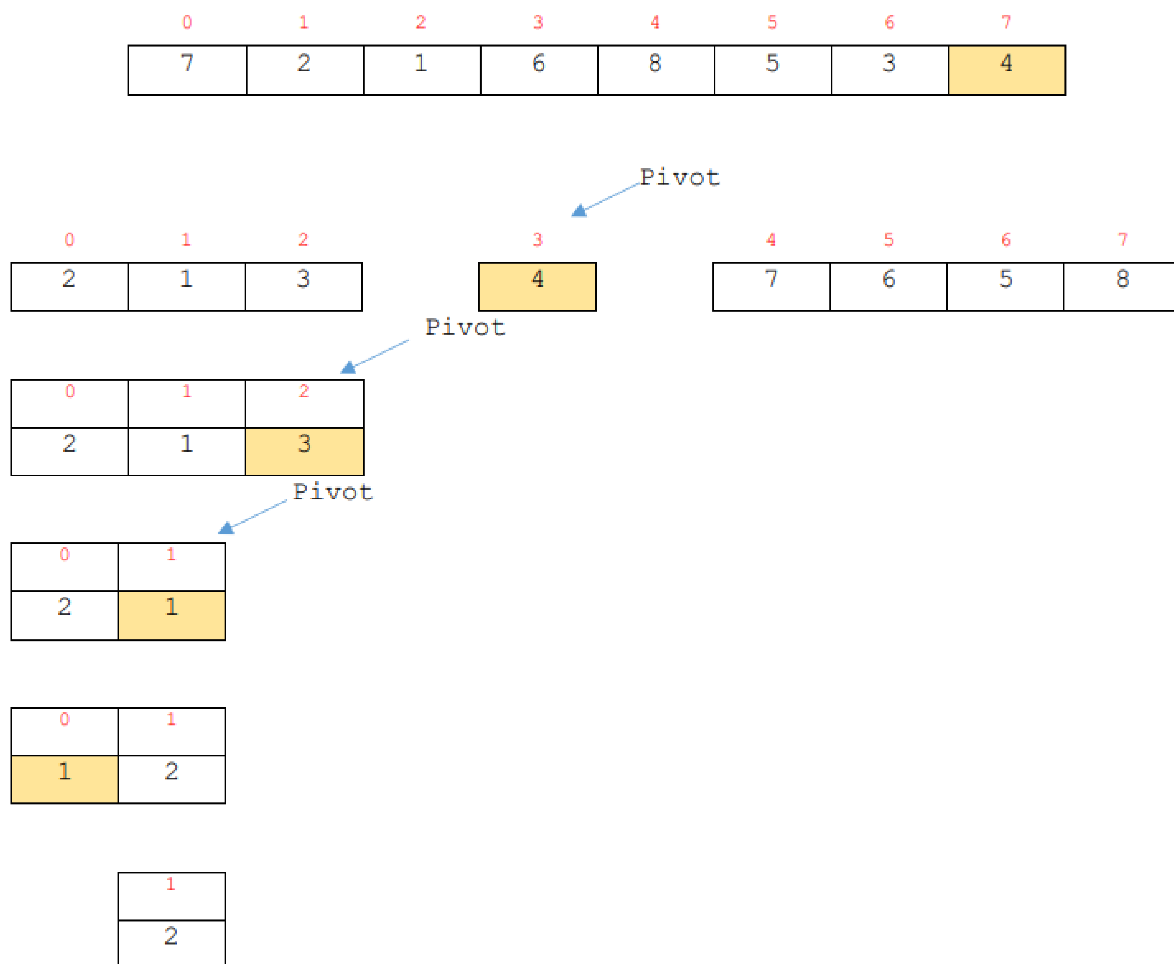


We notice that the merge sort perserve the order of appearance of elements having same key, Therefore merge sort is stable.

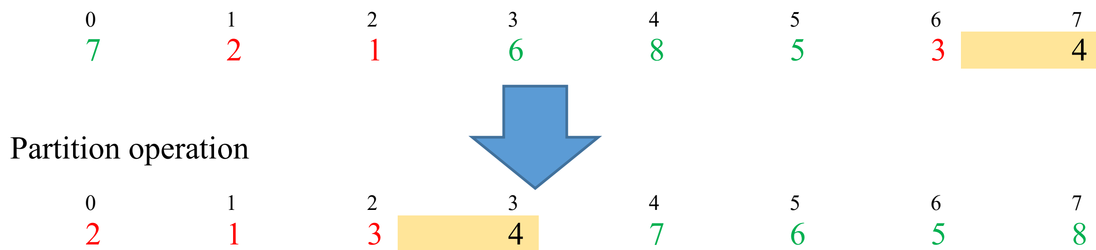
3.8.Quick sort

3.8.1. Quick sort algorithm

Given an array of n elements. One of the element will be referred as pivot will be moved to its final position. The elements of the array will be re-arranged such that all elements whose values are less than the value of the pivot will be at the left hand side of the pivot and all elements whose value is greater than the pivot will be placed at the right hand side of it. This operation is called the partition. Recursively, partition process will be repeated for each of the sub-arrays located at the right and left sides of the pivot. Recursion will stop when the sub-arrays shrink to one single element

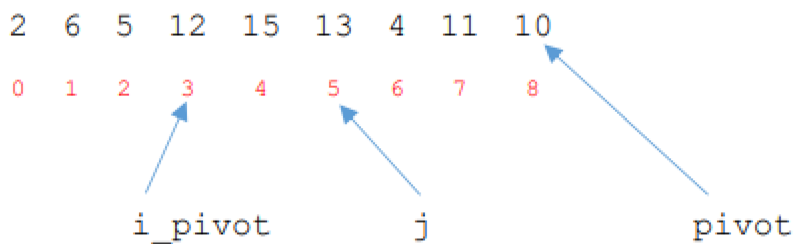


3.8.1.1. Partition algorithm



The operation of partition consist of selecting the last element in the array as a pivot. Moving the elements whose values is less than the pivot value to the left of the pivot. Finally, moving the elements whose values are greater than the value of the pivot to the right of the pivot.

In order to know how partition algorithm operates, we consider the array at an intermediary state during the execution of the algorithm.



- 10, the last element is selected as pivot.
- Elements whose index is from index 0 to i_pivot -1 are the elements that are less than the pivot.
- Elements whose index is from index i_pivot + 1 to j are greater than the pivot.
- After incrementing j, we compare T[j] with the pivot (10).
- if T[j] <= pivot. swap T[i] with T[j] increment i_pivot
- else (T[j] > pivot). Do nothing
- repeat until j = n-2 (n-2 is the index of before the last element)
- At the end, all is left is to swap T[i_pivot] with T[n-1] (where T[n-1] is the pivot)
- Return the index of the pivot (i_pivot), to know the starting and the ending indices of each sub-array.

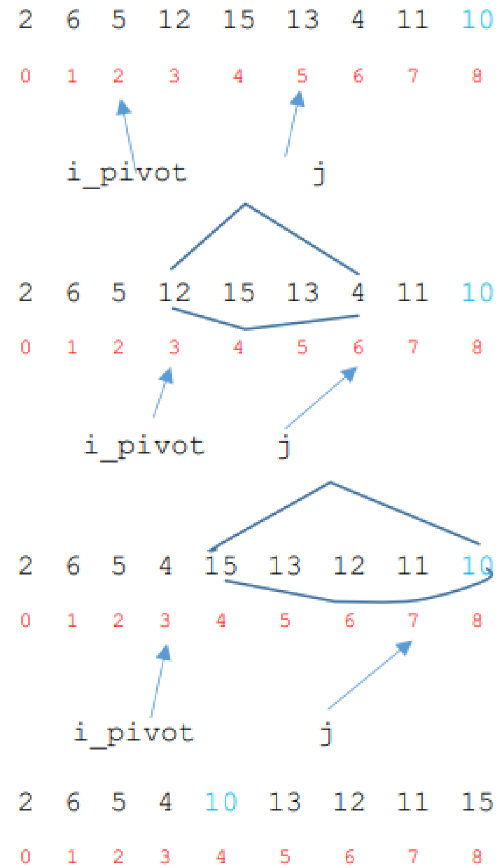


Figure 5 Steps of partition algorithm for the given array

3.8.2. Quick sort algorithm

```
fct partition(T[] :array of int ; start, end :integers) : integer
Var temp,i_pivot,pivot :integer
Begin
     pivot <- T[end]
     i_pivot <- start
     for j <- start to end-1 do
             if (T[j] <= pivot) then
                  Swap(T[i_pivot],T[j])
                  i_pivot++
             EndIf
      EndFor
      Swap(T[ i_pivot],T[end])
      return  i_pivot
End
```

Remark : In order to know the starting and the ending indices of each sub-array at the sides of the pivot. The partition algorithm should be designed to return the value of the pivot.

```
Procedure QuickSort (T[] :array of integers, start, end :integers)
Var pivot_idx : integer
Begin
   if (start >= end)
       return
   Endif
   pivot_idx  <- Partition (T, start, end)
   QuickSort (T , start, pivot_idx  -1 )
   QuickSort (T , pivot_idx +1 , end )
End
```

3.8.3. Time complexity of Quick sort algorithm

```
fct partition(T[] :array of int ; start, end :integers) : integer
Var temp,i_pivot,pivot :integer
Begin
     pivot <- T[end]                  } a
     i_pivot <- start
     for j <- start to end-1 do                               }
             if (T[j] <= pivot) then      }                   }
                  Swap(T[i_pivot],T[j])   } c                 } c*(n-1)
                  i_pivot++               }                   }
             EndIf                                            }
      EndFor
      Swap(T[ i_pivot],T[end])        } b
      return  i_pivot
End
```

```
Procedure QuickSort (T[] :array of integers, start, end :integers)
Var pivot_idx : integer
Begin
   if (start >= end)        }
       return               }  c1
   Endif                    }
   pivot_idx  <- Partition (T, start, end) <------- c2*n + c3
   QuickSort (T , start, pivot_idx  -1 ) <------- g
   QuickSort (T , pivot_idx +1 , end ) <------- h
End
```

$T(n) = c1 + c2*n + c3 + g + h \qquad T(1) = c_4$

In the best case scenario(when the pivot is always placed at the middle of the array after the end of each partition operation), the size of sub-arrays at the left and the right of the pivot will be almost half of the original array. In this case, g = T(n/2) and h = T(n/2)

$T(n) = c1 + c2*n + c3 + 2*T(n/2)$

$T(n) = 2*T(n/2) + c_2*n$

$= 2*[2*T(n/4) + c_2*(n/2)] + c_2*n$

$= 4*T(n/4) + 2*c_2*n$

$= 4*[2*T(n/8)+ c_2*(n/4)] + 2*c_2*n$

$= 8*T(n/8)+ 3*c_2*n$

In general :

$= 2^K*T(n/2^K)+ K*c_2*n$

Recursion stops when the sub-array becomes one single element, in other words, when $n/2^K =1 \iff k = \log_2(n)$

$T(n) = 2^{\log_2(n)}*T(1)+ \log_2(n)*c_2*n$

$= n^*T(1)+ \log_2(n)*c_2*n$

$= n^*c_1+ \log_2(n)*c_2*n$

For $n \to \infty \quad n*\log(n) > n$

In the best case, the time complexity is $\Theta(n*\log n)$

In the worst case scenario(i.e. when the pivot is placed always at one of the corners of the arrays after the end of each partition operation). In this case, the size of one of the sub-arrays is almost equal to n, while the other contains at most one element. In other words, g = T(n-1) and h = 1

$T(n) = T(n-1) + c*n$ $\quad\quad T(1)=c_1$

$= T(n-2)+ c*(n-1) + c*n$

$= T(n-2)+ 2*c*n -c$

$= T(n-3)+3*c*n -3c$

$= T(n-3)+4*c*n -6c$

$= T(n-k) + k*c*n –[1+2+3…+(k-1)]*c$

$= T(n-k) + k*c*n – c*k(k-1)/2$

When n-k = 1,

$T(n) = T(1) + (n-1)*c*n –c*(n-1)*(n-2)/2$

In the worst case, the time complexity is $\Theta(n^2)$

3.8.4. Improved version of quick sort

For quick sort, we can reduce the chance of occurence of the worst case scenario, by selecting randomly one of the elements of the array as a pivot instead of always selecting the last element as the pivot.

```
fct partition_random(T[] :array of integers ; start, end :integers) : integer
Var indiceP, temp : integer
begin
        pivot_idx <- random(start,end)
        temp <- T[pivot_idx]
        T[pivot_idx] <- T[end]
        T[end] <- temp
        partition (T,start,end)
end
```

Where the function random(x,y) returns a random integer number in the interval [x y]. notice that the old partition function is exploited in the new function (partition_random).

```
Procedure QuickSort_V2(T[] :array of integers, start, end :integers)
Var pivot_idx : integer
begin
   if (start >= end)      }
       return             }  C4
   Fin                    }
   pivot_idx  <- partition_random(T, start, end)       c2*n+c3
   QuickSort_V2(T, start, pivot_idx-1 )   <---- h
   QuickSort_V2(T, pivot_idx+1 , end )    <---- g
End
```

T(n) = c1 + c2*n + c3 + g + h   T(1) = c_4

When the pivot is randomly selected among the elements of the array, assuming its index is 'i', therefore, $0 \leq i < n$ => if g takes T(n-i) to execute then h takes T(i) and vice versa.

T(n) = $\frac{1}{n}\sum_{i=0}^{n-1}[T(n-i)+T(i)]$ + c2*n + c1 * c3

$\frac{1}{n}\sum_{i=0}^{n}[T(n-i)+T(i)] = n * \log(n)$ => T(n) belongs to Θ(n*log(n)), if the pivot is choosen randomly.

3.8.5. Space complexity analysis of quick sort

To compute the space complexity for quick sort, which is a recursive algorithm, we need to draw the recursion tree to know the relationship between the input size 'n' and the number of function calls of quick sort algorithm, in the longest branch of the recursion tree.

In the worst case scenario(when the array is reverse sorted), the pivot, after each partition operation, will be always placed as first element of the array. In that case, we would have 'n' function calls to sort the array, as illustrated in the figure below.

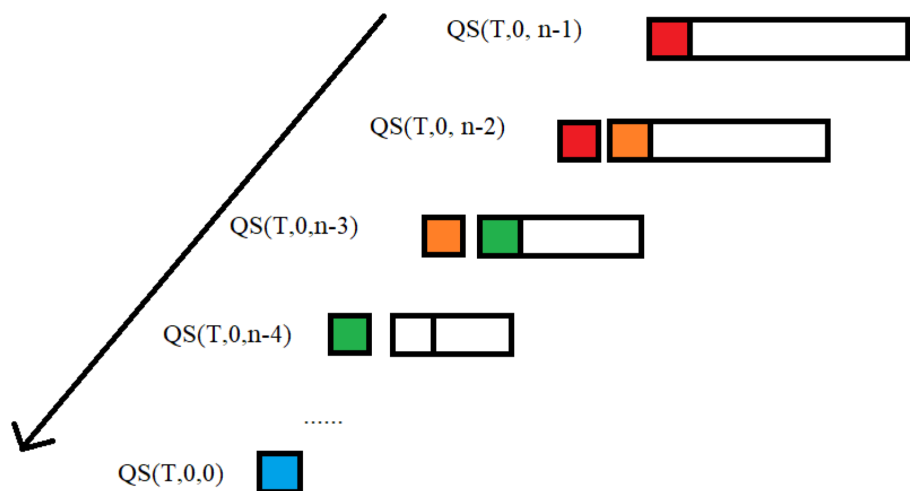


Figure 6 Quick sort in the best case

In the average case, after the end of each partition, the pivot will be almost always placed at the middle of the array, resulting in the following function calls.

QS(T,0,n-1) -> QS(T,0,n/2) -> QS(T,0,n/4) -> … -> QS(T,0,$n/2^k$)

Where k represents the number of function calls. Recursion stops when reaching the base case, when the array to sort is composed of just one element(i.e. when $n/2^k = 1$) => $k = \log_2(n)$.

We conclude that, the space complexity is:

- Θ(n) in the worst case,
- Θ(log(n)) in the average and best cases.

Unlike in Merge sort, in quick sort, only one array is created in memory with some constant number of additional variables. Therefore, the quick sort algorithm is an <u>in place algorithm</u>.

3.9.Comparison of sorting algorithms

For ease of comparison, table below summarizes the characteristics of the studied sorting algorithms.

	Time complexity		Space complexity	Other properties
Selection sort	All cases	$\Theta(n^2)$	Θ(1)	Instable In place
Bubble sort (improved version)	Best case	Θ(n)	Θ(1)	Stable In place
	Average case	$\Theta(n^2)$		
	Worst case	$\Theta(n^2)$		
Insertion sort	Best case	Θ(n)	Θ(1)	Stable In place
	Average case	$\Theta(n^2)$		
	Worst case	$\Theta(n^2)$		
Merge sort	All cases	Θ(n*log n)	Θ(n)	Stable Not in place
Quick sort	Best case	Θ(n*log n)	Θ(n)	instable in place
	Average case	Θ(n*log n)	Θ(log n)	
	Worst case	$\Theta(n^2)$		

Although it is mentioned that the worst case time complexity of quick sort is $\Theta(n^2)$, the probability of occurrence of that worst case is extremely low.

3.10. Conclusion

Sorting is a frequent operation performed in various programs, however, the reason for studying several sorting algorithms, is that these algorithms are excellent examples for learning time and space complexity in depth. Armed with the different techniques examined in this chapter the student will be well prepared for the analysis of other algorithms.

3.11. Exercises

1) A certain sorting algorithm is used to sort the pairs (x,y) according the value of the product x*y, at the beginning the list was :

```
(2,6), (1,2), (2,10), (2,2), (3,4), (5,4)
```

After sorting the list became:

```
(1,2), (2,2), (3,4), (2,6), (5,4), (2,10)
```

- Is this a stable algorithm?, justify your answer

2) Given the collection of pairs (name, grade) which is already sorted according to the alphabetical order of names.

2.1) Resort this collection according to the grade values using :
- Quick sort
- Merge sort

2.2) For each algorithm, deduce its stability

0	1	2	3	4	5	6
(‘**A**hmed’,14)	(‘**B**achir’,8)	(‘**D**onia’,12)	(‘**H**alim’,12)	(‘**I**mane’,15)	(‘**K**amal’,13)	(‘**L**amia’,9)

3) Calculate the time complexity formula T(n)

```
i ← 1;
Tant que (i < n) faire
    j ← 1;
    Tant que (j < 2*n) faire
        j ← j*2;
    Fin TQ;
    i ← i+1;
Fin TQ;
```

```
Pour i de 2 à n faire
    k ← i-1;
    x ← T[i];
    Tant que (T[k] > x et k > 0) faire
        T[k+1] ← T[k];
        k ← k-1;
    Fin TQ;
    T[k+1] ← x;
Fin Pour;
```

Chapter 03 : Trees

4. Chapter 03: Trees

4.1.Introduction

Tree as a data structure are used extensively in storing data in memory (RAM) and in storage devices such as the hard disk drive. It is used to store data that are related to each other in hierarchical fashion. In addition, storing data in some types of trees improves the speed of access and modification.

When dealing with data stored in an array:

- Access to each element is fast, where it suffices to mention the index of the element in question.
- Modification (insertion or deletion) of one of the elements is slow, most of the time, since it requires shifting other elements.

When operating on data stored in a Linear Linked List (LLL):

- Access is sequential and therefore, may necessitate performing a number of operations directly proportional to the number of nodes of that LLL.
- Modification (insertion or deletion of a node), can be made faster by performing this operation at the beginning of this LLL.

In addition, new data can be stored in the LLL, dynamically(In the runtime) and it does not require prior selection of size, as it is the case of arrays.

Storing data in some types of trees(balanced binary search tree), makes the speed of both access and modification relatively fast as compared to storage in arrays and linear linked lists.

4.2.Definitions and terminology

A tree is a collection of **nodes** and **links**. Each node has a **key**. Each node has only one **parent** node, except, one node which is referred as **root node**. Node '100' is the parent of nodes '50' and '20'. On the other hand, we can say nodes '50' and '20' are **child** nodes of '100' (or in short, we say '50' and '20' are children of '100').

There are several types of trees, later on, we will consider the implementation of some types of tree and the corresponding operations.

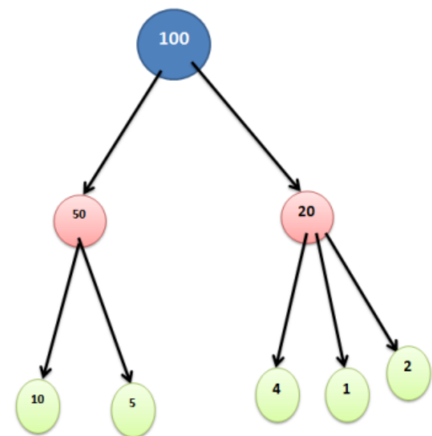


Figure 1 example of a tree

Leaf node (or external node) is a node that does not have child nodes(10,5, 4, 1, 2 are keys of leaf nodes).

Internal node is a node that has at least one child node, nodes with keys 100, 50, 20 are internal nodes.

A tree composed of nodes ‘50’, ‘10’, ‘5’ is a **subtree** of the tree shown above, the root node of this subtree is ‘50’. Therefore, each tree can be viewed as a root node plus the subtrees whose root nodes are children of the root node of that given tree. This recursive nature of the tree, will help coding operations on this data structure in a form of recursive functions.

Descendants of a node ‘i’ are all the nodes contained in the subtrees of the tree whose root node is ‘i’. descendants of D are F, G, H, I, J, K and, L

Ancestors of a node ‘i’ are all nodes in the branch leading from that node to the root node. ‘G’, ‘D’, and, ‘A’ are ancestors of ‘K’.

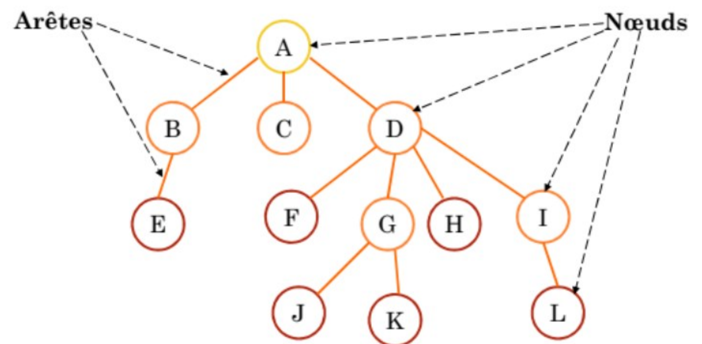
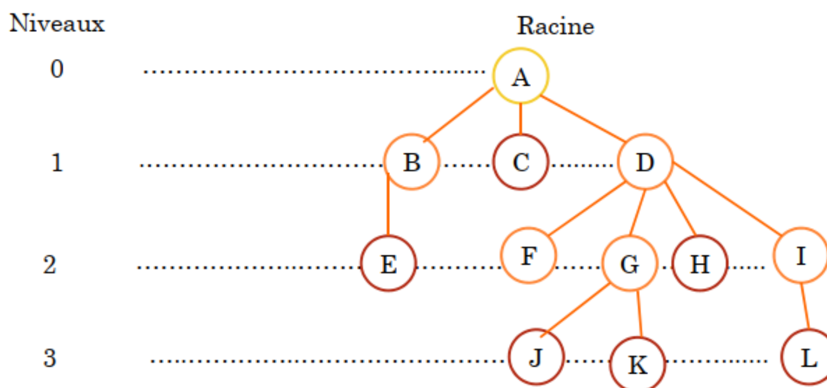


Figure 2 A tree whose keys are alphabets

A level of a node ‘I’ is the distance that separates that node from the root node:

- Level 0 contains only one node(the root node).
- If the degree of tree is known, the maximum number of nodes in any level can be known.



Height of node ‘I’ is the distance separating that node from the root node. Height of node ‘F’ is 2.

Height of a tree is the maximum heights of its nodes. Height of tree illustrated above is 3. Height of a tree composed of only one node is 0. For ease of coding, the height of an empty tree will be taken as -1.

Degree of a node, is the number of its child nodes. Degree of node '20' is 3, degree of node 100, is 2.

Degree of a tree is the maximum of degrees of its nodes. Degree of tree illustrated above is 30.

n-ary tree is a tree whose degree is 'n'. tree shown just above is a 3-ary tree.

A binary tree is a tree whose degree is 2. Each child node of a given node in the binary tree is either a left child or right child.

4.3.Generic tree (n-ary tree) to binary tree conversion

We are interested in binary trees. Therefore, given a generic tree, operations on this tree will be easier if it is converted first to binary tree. The rules for transforming a generic tree to a binary tree are as follows:

1- The root of the binary tree is the same root of the generic tree.
2- For each node in the generic tree :
    a. The first child node(counting from left) of that node in the generic tree is its left node in the binary tree.
    b. The first sibling node (counting from left) of that node in the generic tree is its right left node in the binary tree.

Example: Following the rules mentioned above, we can transform the 3-ary tree on the left to the binary tree on the right:

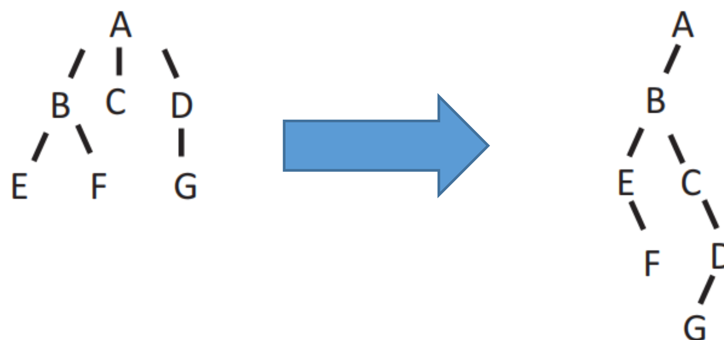


Figure 3 Converting a ternary tree to it corresponding binary tree

Figure below shows how to convert a generic tree to a binary tree in a quick manner. With a link connect each node to its first child as shown in figure b. then, connect each node to its first sibling as shown in c. remove old link to obtain the converted tree.

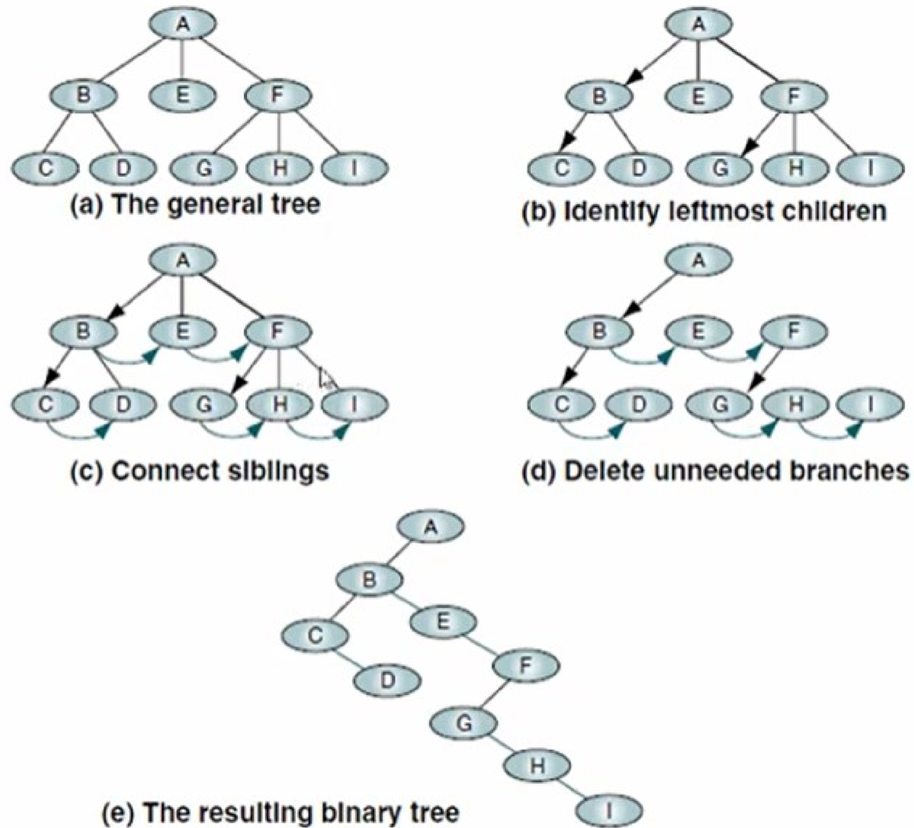


Figure 4 A trick to quickly convert to a binary tree

4.4.Types of binary trees (according to its structure)

Full binary tree is a binary tree where each node has exactly 0 or 2 children.

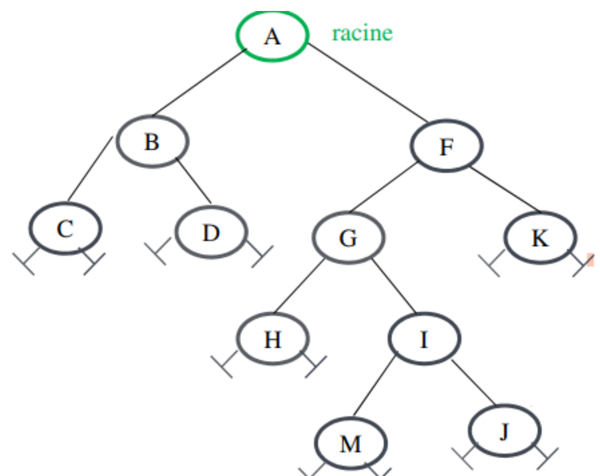


Figure 5 Full binary tree

A perfect binary tree is a full tree where all leaf nodes are in the same level.

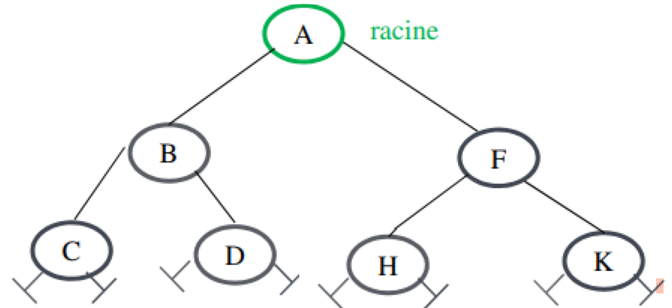


Figure 6 a perfect binary tree

A binary tree is a **complete binary tree** if all the levels are completely filled except possibly the last level and the last level is filled from left to right.

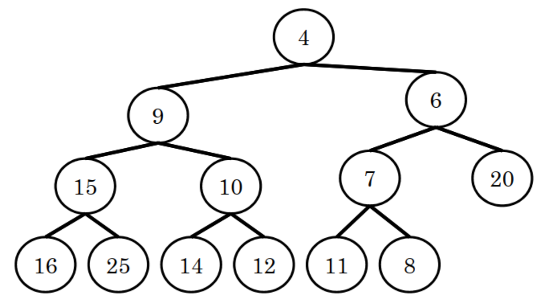


Figure 7 A complete binary tree

The binary tree shown on the right side is not a complete tree since the second level is not completely filled.

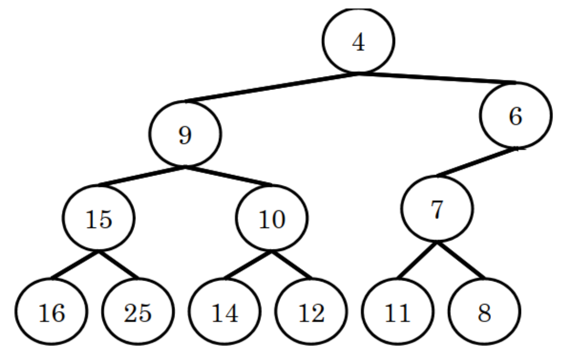


Figure 8 an ordinary binary tree

The binary tree on the left hand side is not completely filled since the last level is not filled from left to right.

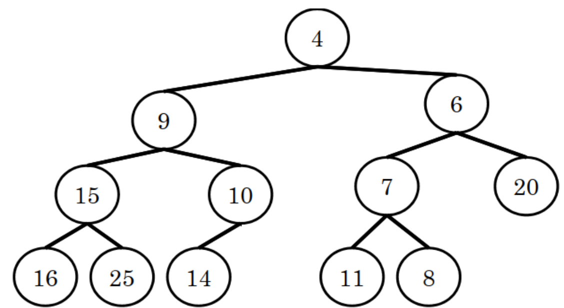


Figure 9 a complete binary tree

4.5. Tree traversal

Traversal consists of displaying all nodes in a specific order. We distinguish two types of traversals

4.5.1. Depth First Traversal

display keys of nodes branch by branch, we count tree instances of Depth tree traversal, namely, preorder tree traversal, inorder tree traversal, postorder tree traversal. It is qualified as depth, since we descend from root node to a leaf node before going back to root.

4.5.1.1. Preorder(Prefix) traversal

We display the root node, then the nodes of the Left Sub-Tree (LST), then the nodes of the Right Sub-Tree (RST). When displaying nodes of each subtree we, recursively, apply the same rule. In short we can right the rule as : root->LST->RST. The prefix 'pre' in the naming of that traversal refers to the position of the root with respect to left and right subtrees. In other words, root is visited before the subtrees. Left subtree is always visited before the right subtree.

Root -> LST -> RST

result of preorder traversal is :

12,1,91, 67, 7, 82, 61

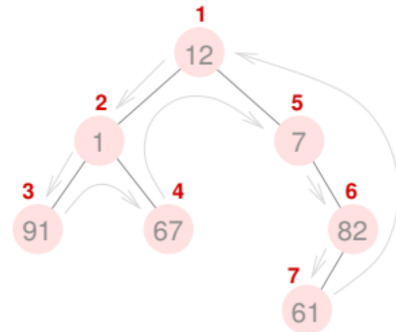


Figure 10 Preorder traversal

4.5.1.2. Inorder(Infix) traversal

We display the nodes of the left subtree, then the root node, then the nodes of the Right Sub-Tree (RST). When displaying nodes of each subtree we, recursively, apply the same rule. In short we can right the rule as : LST->root->RST.

Left subtree is always visited before the right subtree. The prefix 'in' in the naming of that traversal refers to the position of the root with respect to left and right subtrees. In other words, root is visited after the left subtree and before the right subtree.

LST->root->RST.

Result of inorder traversal is:

91, 1, 67, 12, 7, 61, 82

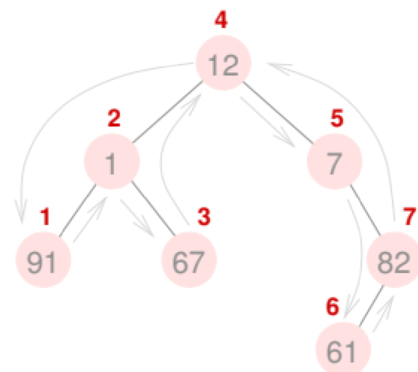


Figure 11 Inorder traversal

4.5.1.3. Postorder(Postfix) traversal

We display the nodes of the left subtree, then the nodes of the Right Sub-Tree (RST), the root node. When displaying nodes of each subtree we, recursively, apply the same rule. In short we can right the rule as : LST-> RST-> root.

Left subtree is always visited before the right subtree. The prefix 'post' in the naming of that traversal refers to the position of the root with respect to left and right subtrees. In other words, root is visited after visiting the left subtree and the right subtree.

LST-> RST-> root.

Result of postorder traversal :

91, 67, 1, 61, 82, 7, 12

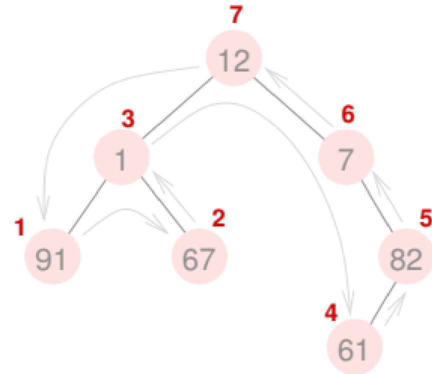


Figure 12 Postorder traversal

When applying one of the depth first traversals, there is quick way to know the order of display of keys. For the preorder traversal we draw a dash(or a dot) the left side of each node. We start at the top of the root node and start descending from the left side, each time we cross a dash of given node, we display the key value of that node, we stop when reaching the top of the root node from the right side. For the inorder and postorder traversal we follow the same steps. But by considering the new position of the dash(or the point) as illustrated in figure below.

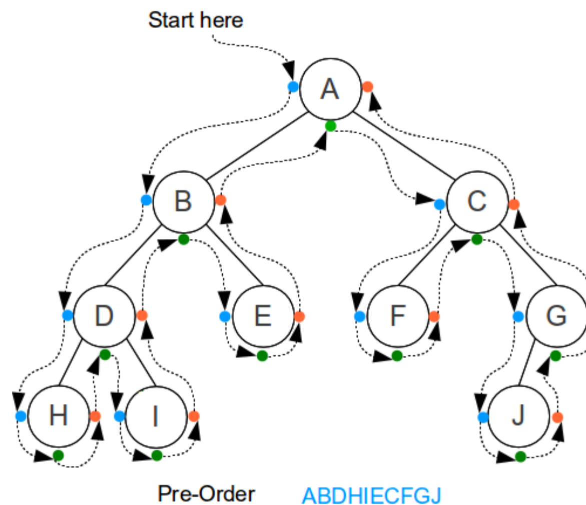


Figure 13 Quick way to perform depth-first traversals

4.5.2. Breadth first traversal

We display nodes in a level by level order, starting from level zero and for each level we display nodes starting from left.

Result of Breadth first traversal:

12, 1, 7, 91, 67, 82, 61

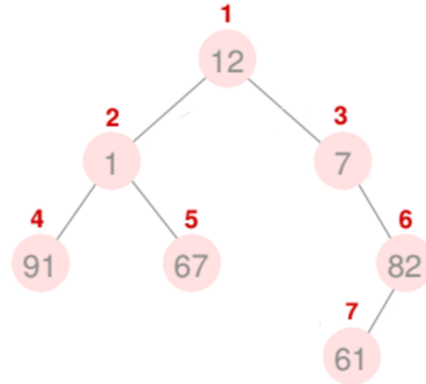


Figure 14 Breadth-first traversal

4.6. Max heap & Min heap

4.6.1. Heap

max heap and min heap can be used to sort a sequence. In addition, max heap and min heap can be exploited to implement priority queues.

4.6.2. Max heap

A binary tree that has two properties :

1- **Structural property**: it is a complete binary tree. In other words, all levels of the binary tree are completely filled, except possibly, the last level which has to be at least filled from left to right.
2- **Order property**: The key of each node is <u>greater</u> than the values of its descendants.

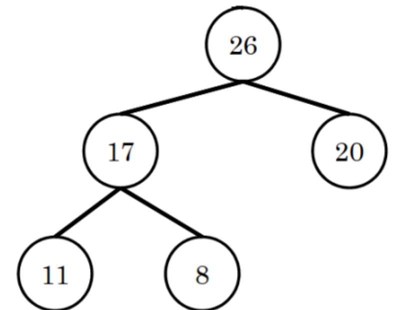


Figure 15 max heap

4.6.3. Min heap

A binary tree that has two properties :

1. **Structural property**: it is a complete binary tree. In other words, all levels of the binary tree are completely filled, except possibly, the last level which has to be at least filled from left to right.
2. **Order property**: The key of each node is <u>less</u> than the values of its descendants.

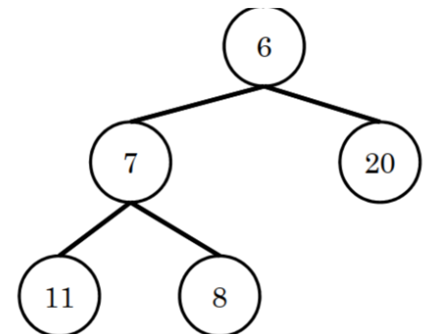


Figure 16 min-heap

Note: We will see how to implement max heap and its related operations. Then, min heap and its related opertions can be deduced easily.

4.6.4. Representation of max heap in memory

Given that the max heap is a complete binary tree, it can be stored in an array (starting from element index 1), where for each node having index i

- Its parent node is at **└ i/2 ┘**.
- Its left child node is at 2*i.
- Its right child node is at 2*i + 1

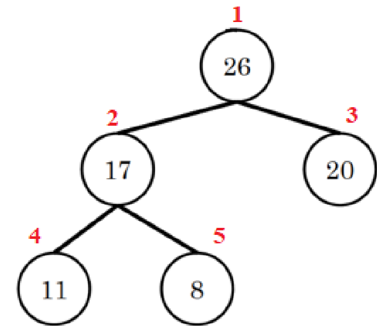
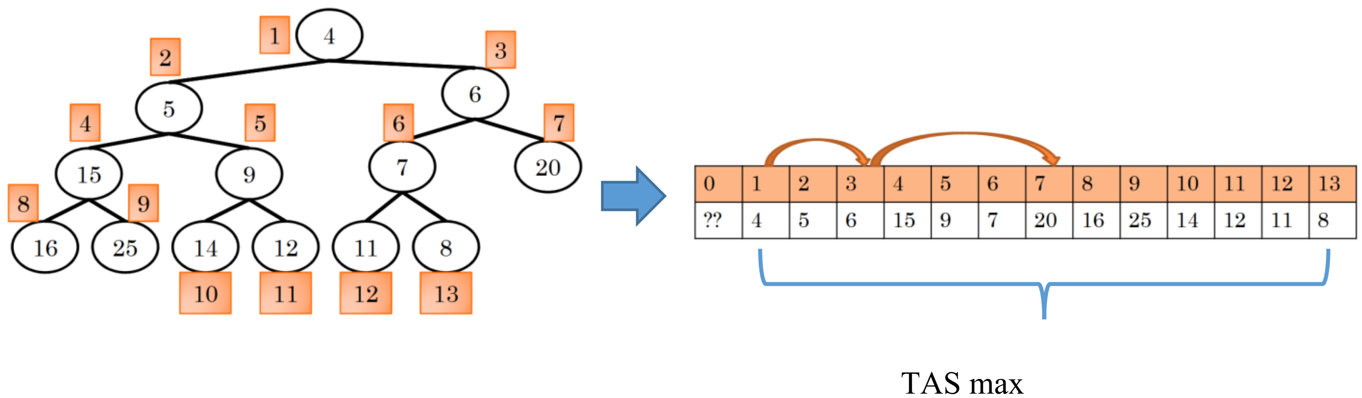


Figure 17 Maxheap nodes ordering

Note: └ x ┘ is the floor value of the real number x.

example └ 3,999 ┘ = 3 and └ 3,0001 ┘ = 3.

Note: In c language, indices of the array start from 0. In order to deduce the relationships between elements of the array, we ignore the element with index 0. Therefore, when dealing we a max heap of size n, an array with size of n+1 is created while ignoring the element at index 0.



4.6.5. Insertion in a max heap

To insert a value 'v' in a max heap :

1- Insert the element 'v' as a last leaf node.
2- Swap this element if it greater than its parent.
3- Repeat step two, until this element is less than its parent or until this element become a root node.

Step 1 is required to keep the tree, a complete one.

Steps 2 and 3 are performed to make sure that the final tree represents a max heap.

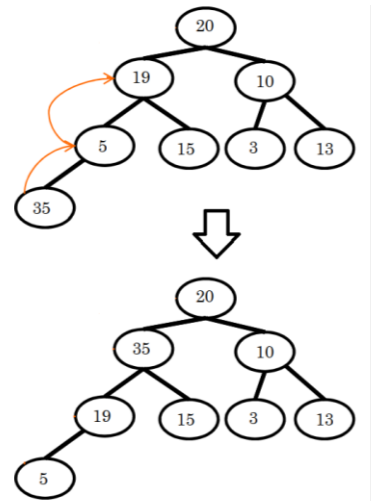
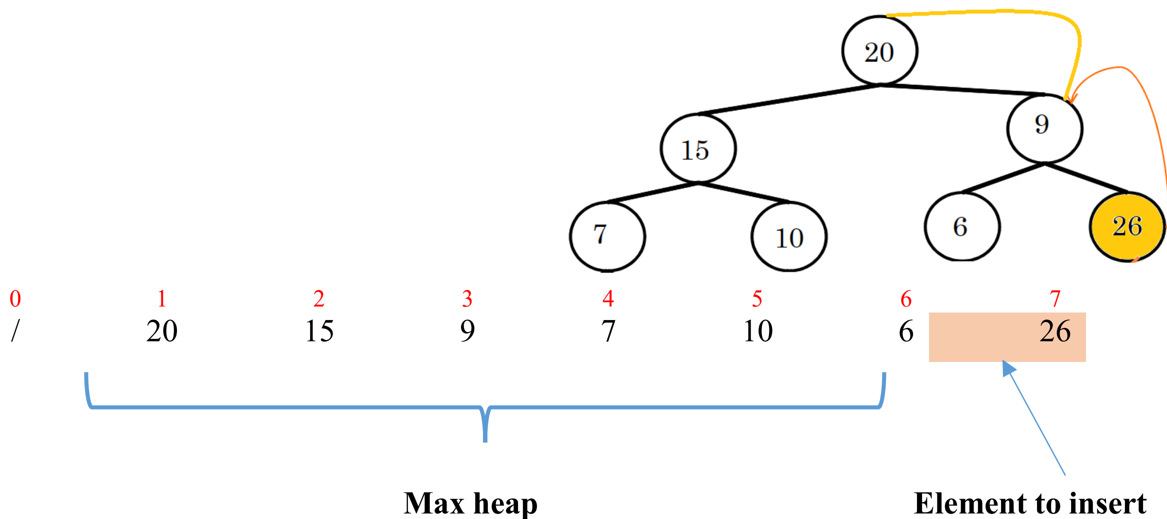


Figure 18 Insertion of node 35 to max-heap

Given an array T of size n, where the first 1 … (n-1) elements represent a max heap. We want to insert the last element with index 'n' in the max heap.



4.6.6. The insertion algorithm

We compare this element(node) with its ancestors, starting from its parent. We swap the node with its parent, if it is greater than its parent. We repeat the same process with that swapped node with its new parent. We stop iteration if node value is less its parent value or when the node after being swapped becomes a root node.

```
Procedure Insert(T[] : array of int, index : integer)

Start
        While (index  > 1 AND T[index ] > T[index/2]) do
            Swap(T[index ], T[index/2])
            index  <- index/2
        EndWhile
End
```

4.6.7. Building max heap from an arbitrary complete binary tree

Given a complete binary tree represented in a form of an array T of n elements. Using the insert procedure defined above, we can transform this arbitrary complete binary tree to a max heap :

- We can consider the first node alone as a max heap.
- We insert the second node to the max heap. (We obtain a max heap composed of two nodes.)
- In the same manner, we insert the remaining nodes.

```
Procedure BuildMaxHeap(T[] : array of integers, n : integer)
Var i : entier
Begin
    For i <- 2 to n do
        Insert(T,i)
    EndFor
End
```

Example Creation of max heap from an arbitrary complete binary tree reprsented in a form an array 15, 35, 10, 20, 5, 40, 25

Current node
Its parent

	0	1	2	3	4	5	6	7
i=2	/	15	35	10	20	5	40	25
	/	35	15	10	20	5	40	25
i=3	/	35	15	10	20	5	40	25
	/	35	15	10	20	5	40	25
i=4	/	35	15	10	20	5	40	25
	/	35	20	10	15	5	40	25
i=4	/	35	20	10	15	5	40	25
	/	35	20	10	15	5	40	25
i=5	/	35	20	10	15	5	40	25
	/	35	20	40	15	5	10	25
i=6	/	35	20	40	15	5	10	25
	/	40	20	35	15	5	10	25
i=7	/	40	20	35	15	5	10	25
	/	40	20	35	15	5	10	25

35>15 => swap

node 35 become a root node, we stop.

10<35 do nothing

20>15 => swap

20<35 => do nothing

40>10 => swap

40>35 => swap

Node value 40, become root node => we stop.

25<35 => do nothing.

Max heap

Figure 19 Steps array transformation to a max-heap

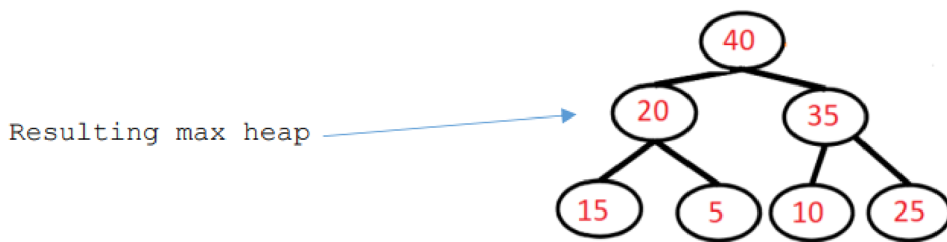


Figure 20 Resulting max-heap by seccussive insertion

4.6.8. Deletion of root node from max heap

In a max heap represented in a form of an array:

1. We delete the root node (save it in a separate temporary variable). We move the last leaf node as the new root to keep the sub-array from index 1 to n-1 representing a complete tree.
2. We transform the new complete binary tree obtained from step 1, to a max heap.
   - **a.** We compare the new root node (call it current node) with its child nodes. If one of the child nodes has the biggest value, We swap that child node with this current node.
   - **b.** We repeat the same process with the current node and its new child nodes.
   - **c.** Process is stopped when, this current node has higher value than the values of its children. Or when this current node becomes a leaf node.
   - **d.** Deleted root node in step 1, will be placed as the last element of the array.(this will not affect the new reduced size max heap since the last element does not belong to it.)

```
Procedure DeleteRoot(T[1..n] : array of integers, n : integer)
Var temp, i, indiceMax : integer
Begin
            temp <- T[1]
            T[1] <- T[n]
            i <- 1
                     If current node has at least one child

            While (2*i < n)
                  i_max <- i
                  if (2*i < n AND T[2*i]> T[i]) Then
                     i_max <- 2*i
                  Endif

                  if (2*i+1 < n AND T[2*i+1]>T[i_max]) then
                     i_max <- 2*i+1
                  Endif

                  if (i_max ≠ i)
                     Swap(T[i],T[i_max])
                     i <- i_max
                  else
                     Break (break out of the loop)
                  Endif
            EndWhile
            T[n] <- temp
End
```

Removed root stored in the empty last cell in the array

4.6.9. Heap Sort

Given a max heap of 'n' nodes and which is represented in a form of array, if we keep removing root of each resulting max heap -using the DeleteRoot function mentioned above- we will obtain an array sorted in ascending order. Building a max heap from an arbitrary array then deleting each time the root of the resulting max heap is referred as max heap, steps can be detailed as follows :

1- Given a complete binary tree represented as an array T of n elements. Transform this array to a max heap.
2- Exploiting the DeleteRoot function, remove root of each resulting max heap.

The array T must be representing a max heap.

```
Procedure HeapSort(T[] : array of integers, n : integer)
Var i : integer
Begin
    For i <- n  downto 2 do
        DeleteRoot(T,i)
    EndFor
End
```

4.6.10. Heapifying a complete binary tree

To transform a complete binary tree to a heap, is to heapify a complete binary tree. Starting from the last leaf node up to the root node following the breadth first traversal in the reversed direction. We build a max heap from each node with its left and right child nodes. If the value of current node is less than one of its child nodes, they will be swapped and we recursively, repeat the same operation with the current node along with its new child nodes. We stop recursion, if the current node becomes a leaf node.

```
Procedure Heapify(T[] : array of integers, n , i : integer)
Var i_max : integer
Begin
        i_max <- i
        if (2*i <= n AND T[2*i] > T[i]) then
            i_max  <- 2*i
        Endif

        if (2*i+1 <= n AND T[2*i+1]> T[i_max]) then
            i_max  <- 2*i+1
        Endif

        if (i_max ≠ i)then
            Swap(T[i],T[i_max ])
            Heapify(T, n, i_max )
        Endif
End
```

4.6.11. Building a max heap using heapify

As mentioned before, we ‘heapify’ a complete binary tree starting from the last leaf node, however, since a leaf node does not have child nodes, the leaf node is always considered as max heap. In that regard, we can start heapifying the complete binary tree starting from the last internal node instead of the last leaf node.

For a given complete binary tree saved in a form of an array, of n elements. The indices of internal nodes goes frmo 1 to └ n/2 ┘ and the indices of exertnal nodes (leaf nodes) goes from └ n/2 ┘ +1 to n.

```
Procedure BuildMaxHeap_V2(T[] : array of integers, n : integer)
Var i : integer
Begin
    For i <- └ n/2 ┘ downto 1 do
        Heapify(T, n, i)
    EndFor
End
```

4.6.12. Searching for a value in max heap

To look for a value ‘v’ in the max heap, we scan the tree using breadth first traversal(level by level starting from root node). We pass to level ‘i’ only if the value ‘v’ is less than the the values of nodes in level ‘i-1’.

4.7. Additional definitions

4.7.1.1. Balanced or Unbalanced Tree ?

If we consider each node in the binary tree as a root node of subtree composed of that node and its descendants. For each resulting subtree we check if the difference between heights of its left subtree and of its right subtree is less or equal to k, where k is an natural number set most of the time to 1. If this condition is satisfied we say that the entire tree is a balanced tree otherwise, it is an unbalanced tree. In other words, a binary tree is balanced if :

$$|\text{height(left subtree)} - \text{height(right subtree)}| \leq k$$

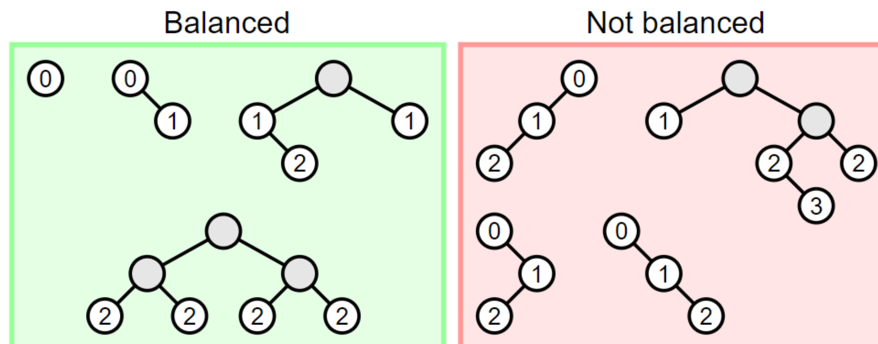


Figure 21 examples of balanced and unbalanced trees

A degenerated tree is a tree where each internal node has exactly one child node. A degenerated tree is always an unbalanced tree.

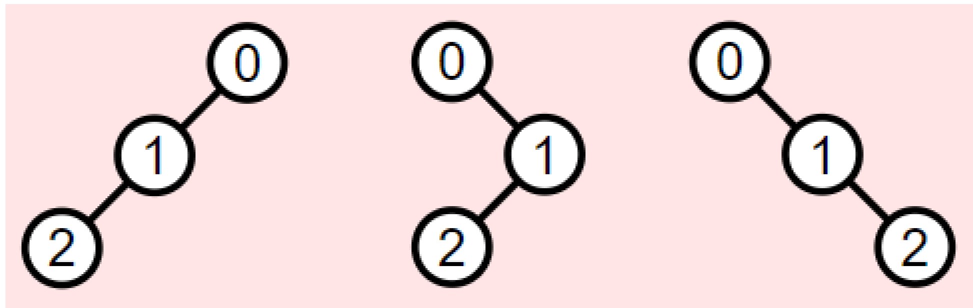


Figure 22 Degenerated trees

4.7.1.2. Relationship between height of a tree and the number of its nodes.

The height of a tree is the distance between the root node and a leaf node in the longest branch, (we count the links not the nodes !)

Given a binary tree of height 'h', the minimum number of nodes is $\mathbf{n_{min} = h+1}$ (the case of a degenerated tree) and the maximum number of nodes is $\mathbf{n_{max} = 2^{h+1} - 1}$ (the case of a perfect number).

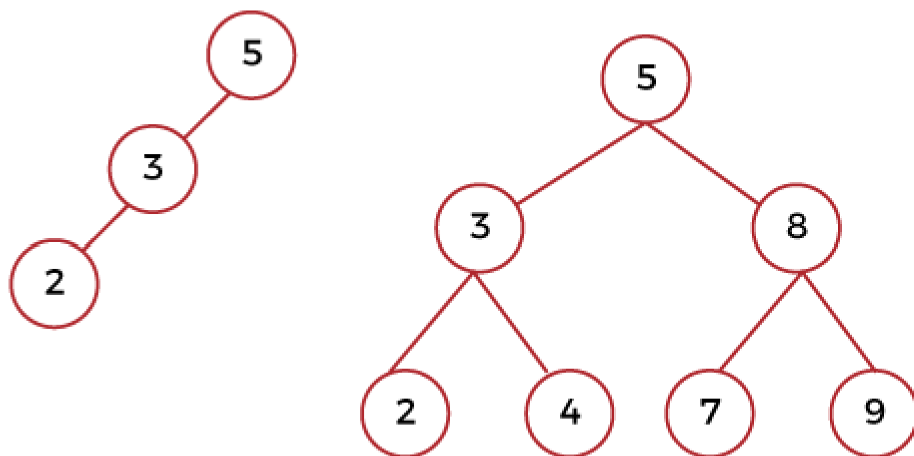


Figure 23 Minimum and maximum number of nodes for height = 3

On the other hand, if the number of nodes is given the height will be $\mathbf{n-1 \leq h \leq \log (n)}$

h = n-1 in case of a degenerated tree.

h α log n, case of any other balanced tree.

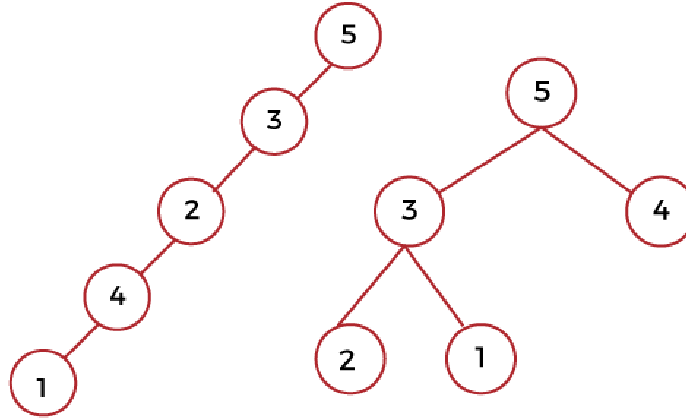


Figure 24 minimum and maximum height given 5 nodes

4.8. Time complexity analysis of operations on tree

Most operations on trees are directly proportional to the height of that tree.

Given that a max heap is a complete tree and knowing that every complete tree is a balanced tree, we can conclude the height 'h' of max heap is directly proportional to the number of nodes 'n'.

In case of insertion of a new node in a max heap, in the worst case, the value of that node will be higher than the values of its ancestors. In that case, we need to swap that node 'h' times with its ancestors. In other words, the time complexity of insertion operation, in the worst case, is Θ(h) or Θ(log n) since h α log(n).

To build a max heap from a complete tree represented in an array, initially we consider only the first element representing a max heap and we proceed to insert the remaining array elements starting for the second element to the last one. Therefore, the time complexity for building a max heap is Θ(n)* Θ(log n) or Θ(n log n).

The time complexity of building a max heap using 'heapify' algorithm can be estimated by considering the figure illustrated below:

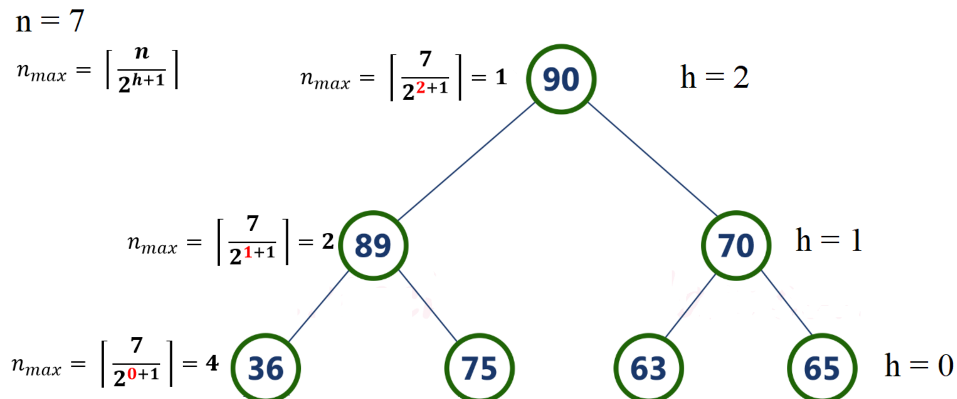


Figure 25 max number of nodes having same height

- The maximum number of nodes having same height h, given the total number of nodes n is **⌈ n / 2^{h+1} ⌉.**
- Each node can be swapped at max ‘h’ times with its ancestors, then, ‘heapify’ applied on only one node, costs O(h).
- The height of a max heap is directly proportional to log n.
- Assuming that we call ‘heapify’ n times starting from the last leaf node and not only from the last internal node.

- We compute the cost at each height and sum over all possible heights :

$$T(n) = \sum_{h=0}^{\log n} \left\lceil \frac{\boldsymbol{n}}{\mathbf{2}^{\boldsymbol{h}+\mathbf{1}}} \right\rceil * \mathbf{O(h)}$$

$$T(n) = \sum_{h=0}^{\log n} \left\lceil \frac{\boldsymbol{n}}{\mathbf{2} * \mathbf{2}^{\boldsymbol{h}}} \right\rceil * \mathbf{c} * \mathbf{h}$$

$$T(n) = \frac{c * n}{2} \sum_{h=0}^{\log n} \left\lceil \frac{\boldsymbol{h}}{\mathbf{2}^{\boldsymbol{h}}} \right\rceil$$

$\sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil = \mathbf{2}$ => $\sum_{h=0}^{\log n} \left\lceil \frac{h}{2^h} \right\rceil < \mathbf{2}$ => T(n)= c*n => T(n) belongs to O(n).

4.8.1. Time complexity of deletion from a max heap

In case of deletion of the root node from a max heap, In the worst case the leaf node replacing the root node, will be swapped ‘h’ times with some elements from its decsendants. In other words, deletion operation is directly proportional with h. T(n) α Θ(h) α Θ(log n).

1- Given an a complete binary tree implemented in an array, it requires a times proportional to n*log(n) to build a max heap using the first discussed algorithm and its costs a time proportional to n, if ‘heapify’ method is used.
2- Moreover, given a max heap implemented in an array. It is required to delete all the ‘n’ elements to obtain a sorted array. On the other hand, deletion of one single root costs O(log n).

From 1 and 2, we can conclude that heap sort costs a time proportional to n*log(n) to execute.

4.9.Binary Search Tree (BST)

A Binary search Tree(BST) is a binary tree, where the value of each node is greater than all nodes in its Left Sub-tree(LST) and less or equal to values of nodes in its Right Sub-tree(RST).

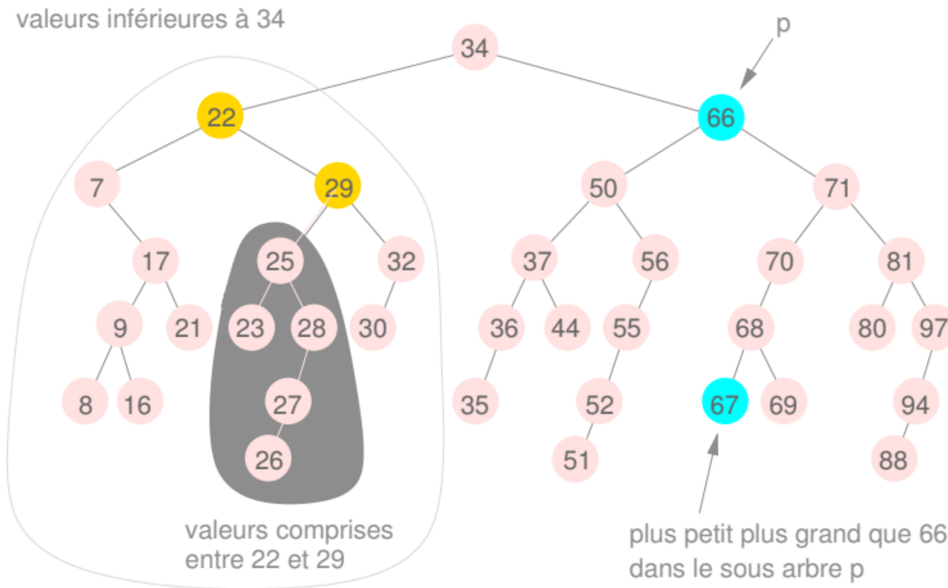


Figure 26 Example of a binary search tree

4.9.1. Advantages of BST

1- Speed the running time of frequent operations such as insertion, deletion and search.
2- The inorder traversal display elements in ascending sorted order.

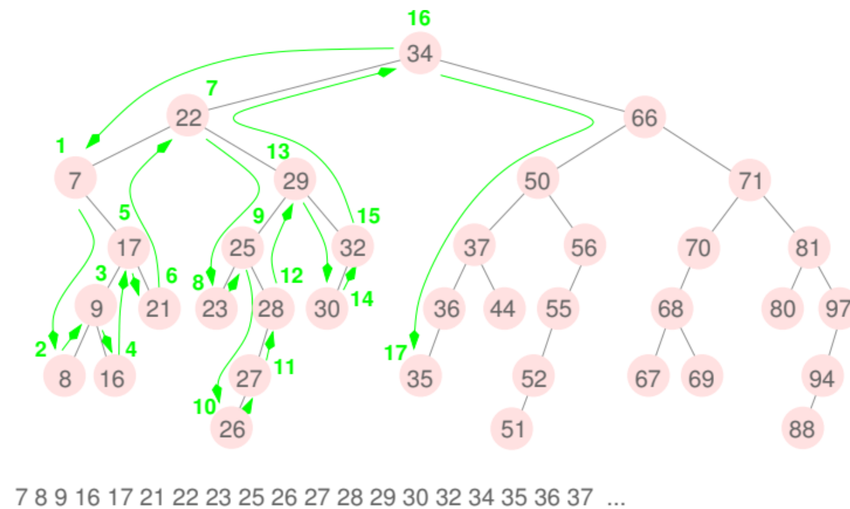
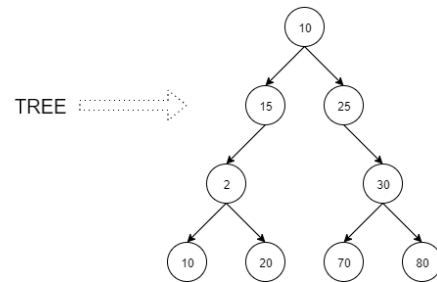


Figure 27 Sorting using binary sort tree

4.9.2. Implementation of binary trees

If we want to implement an arbitrary tree using arrays, We must store values of nodes in the array in a way that help us deduce the parent and child nodes of each node from the indices of the array. As we have seen in the previous chapter related to max heap. If we follow the breadth-first traversal to store nodes of the tree starting from cell index 1, the node with index ‘i’ will have parent at index ‘ $\lfloor i/2 \rfloor$ ’ and left child at index ‘2*i’ and right child at index ‘2*i+1’(provided that the indices does not exceed the size of the array).

Following this rule, the only type of trees that can be implemented in a form of array –without leaving empty cells is the binary complet tree. If a tree is not a complete binary tree and it is implemented in an array, an important numbers of cells has to be left empty leading to uncessary memory consumption, figure below illustrates such case.



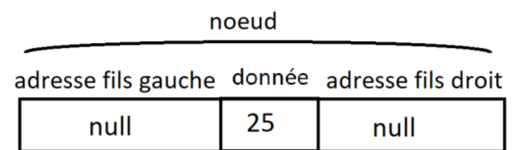
SEQUENTIAL REPRESENTATION

10	15	25	2	-	-	30	10	20	-	-	-	-	70	80

Figure 28 issue when storing any incomplete BST in an array

To avoid uncessary memory consumption when implementing a incomplete binary tree. We define each node as structure containing three field. A field specified for storing data, a second field for storing the address of the left child of that node and the third field for storing the address of the right right of the node. Graphical representation along with the corresponding pseudocode is shown below.

```
node : structure
        data        : integer
        LeftChild   : pointer on node
        RightChild  : pointer on node
endOfStructure
```



To have access and identify a binary tree, we store the address of the root node in a separate variable.

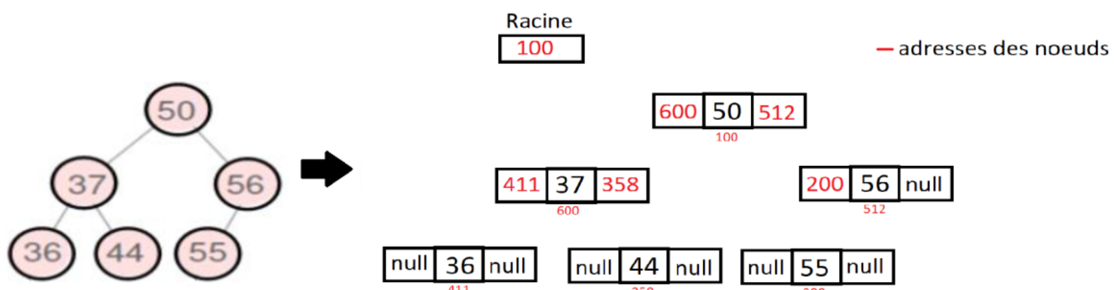


Figure 29 Implementation of BST using structures

4.9.3. Frequent operations on BSTs

4.9.3.1. Insertion operation

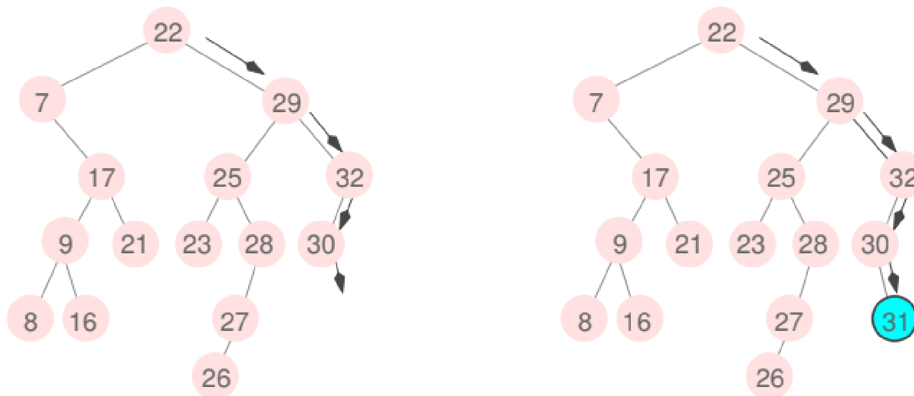


Figure 30 Insertion operation on a BST

We want to insert a new node having the value 'v' in a BST :

- We create a new node in memory and fill the data field with the value 'v'. Given that every inserted node in a BST will be leaf node, we fill the field values of left and right children with Null value.
- If the BST is empty (the address of the root node in this case is null, we assign the address of this node to the root variable.
- Otherwise if not empty (the address of the root node is not null) :
  - If the value 'v' is less than the value stored in the root node, we insert, recursively, the new node in the left sub-tree.
  - Otherwise, we insert, recursively, the new node in the right sub-tree.
- The insertion operation may affect the value of the root node, the new value of the root node is returned at the end of insertion of each node.

```
Fct Insert(root: ptr on node, value : integer) : ptr on node
Var temp : ptr on node
Begin
        if (root = null) then
          temp <- Allocate(sizeOf(node))
          temp.data <- value
          temp.LeftC  <- null
          temp.RightC  <- null
          root <- temp
        else if (value <= root.data ) then
          root.LeftC  <- Insert(root.LeftC, value)
        else
          root.RightC <- Insert(root.RightC, value)
        EndIf
        Return root
      End
```

The 'allocate(x)' function (malloc function in c language), will allocate a space in memory of size x and it will return the address of the allocate space.

Insert function is called multiple times to create a complete binary tree.

```
Function main
  Var root: ptr on node
   Begin
        root <- null
        root <- Insert(root,50)
        root <- Insert(root,37)
        root <- Insert(root,36)
        root <- Insert(root,56)
        root <- Insert(root,44)
        root <- Insert(root,55)
   End
```

The resulting BST after execution of the above code :

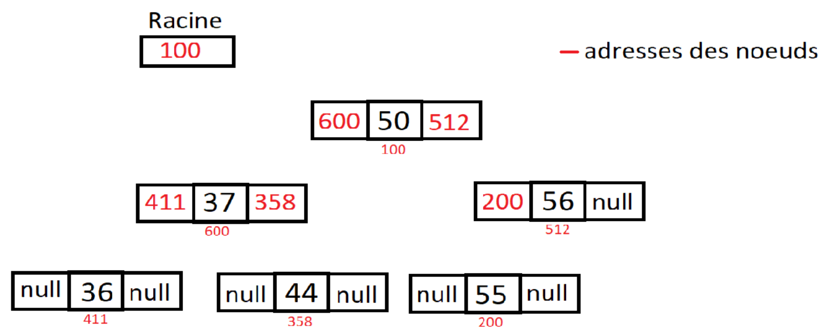


Figure 31 BST implemented in memory

In the worst case, the new node will be inserted as a leaf node in the longest branch. The value of this new node is compared 'h' times with the value of nodes in that branch. Where 'h' represent the height of the tree since it is equal to number of links in the longest branch.

Then, the insertion operation is directly proportional to the height of the tree. If the BST is balanced, the insertion operation costs a time proportional to 'log n' otherwise, it is proportional to 'n'.

Time complexity of insertion operation is O(h). In other words, time complexity is O(log n) if this BST is balanced , O(n) otherwise.

4.9.4. Searching for a value in a BST

Searching for a value in a BST consist of comparing it first with the value stored in root node, if the searched value is less or equal to the value in the root node, we search again in the left sub-tree starting from its root node. Otherwise if the searched value is greater than the value of the root node, we search in right sub-tree. The process is repeated until we find the searched value or we reach a leaf node whose value is not equal to the search value.

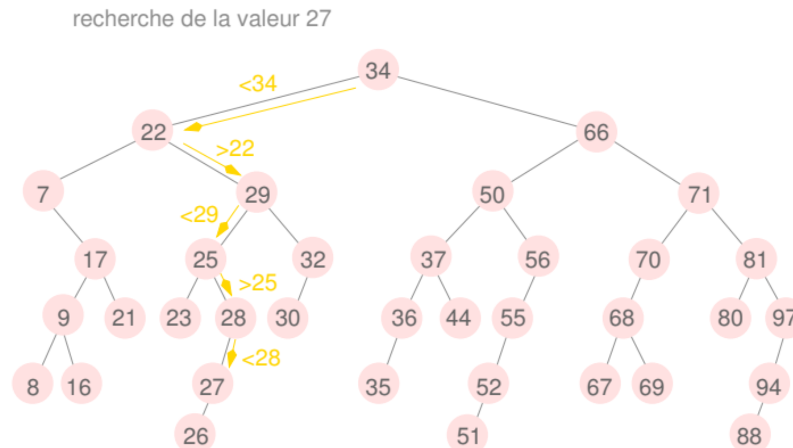


Figure 32 searching for a value in a BST

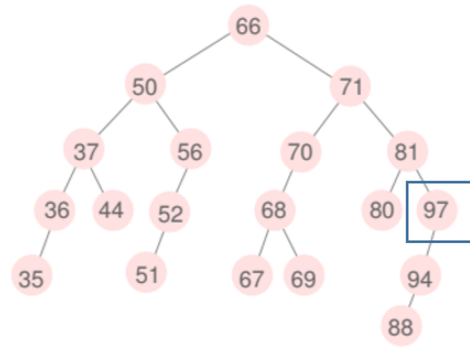
```
Fct search(root: ptr on node, value : integer) : boolean
Begin
  if (root = null) then return False  Endif
  if (value = root.data) then return true Endif
  else if (value <= root.data) then
     return search(root.LeftC, value)
  else
     return search(root.RightC, value)
  Endif
End
```

In the worst case, the time complexity of search operation is O(h), in terms of n, it is O(log n) is the tree is balanced, O(n) if it is not balanced.

4.9.5. Operation of finding the node with maximum value

To find the node with the highest value in the BST. We start from the root node and descend in the right sub-tree until we find a node that does not have a right child. This latter will be the node with the highest value.

```
fct max(root: ptr on node) : integer
Begin
    if (root = null)
            print('root is empty')
            return
    else if (root.rightC = null)
            return root.data
    endif
        return max(root.RightC)
End
```



4.10. Implementation of Traversals

4.10.1. Depth-first traversals

4.10.1.1. Preorder traversal

we recall that the rule for this traversal is

Root -> Left sub-tree -> Right sub-tree

```
Procedure Preorder(root : pointer on node )
Begin
   if(racine == nul) then return  EndIf
   Print(root.data)
   Preorder(root.LeftC)
   Preorder(root.rightC)
End
```

Exercise : Give the algorithm for inorder and postorder traversals.

4.10.2. Breadth-first traversal implementation

In this traversal, we need to visit nodes levels by level starting from the root node. if we assign a pointer having the address of the root node, we can exploit this pointer to display the data saved in the root node. From the root node, we can move the pointer to the left child node to display its value. The problem is that each node has only the address of its children. therefore, we cannot move from right child to the left child to display its value and we cannot move back from a given child to the parent. As a solution, we can save the address of each visited node along with the address of its child node in a queue data structure. In that way, we can easily display the data following the algorithm explained below :

1- Having already a queue and the BST which we want to display its values following breadth-first traversal, We enqueue the address of the root node.
2- While the queue is not empty :

- We dequeue and save the dequeued address in temporary pointer variable.
- Using that pointer, we display the data and enqueue children of the node whose address is hold in that pointer.

Exercise : Translate this pseudocode to code in C language.

4.10.3. Time complexity of traversals

For each traversal, it is required to display the values of all nodes, therefore, the running time of each traversal is directly proportional to n.

4.11. Deletion operation

The operation of deleting a node from a BST depends on the number of child nodes it has.

Case 1 : The node to delete **has no child nodes(it is a child node)**. In this case, this node can be deleted directly by breaking the link to its parent. If the BST from which we want to delete the node is composed only of that node.it is deleted directly without the need of breaking any link. Root is assigned to null in the latter case.

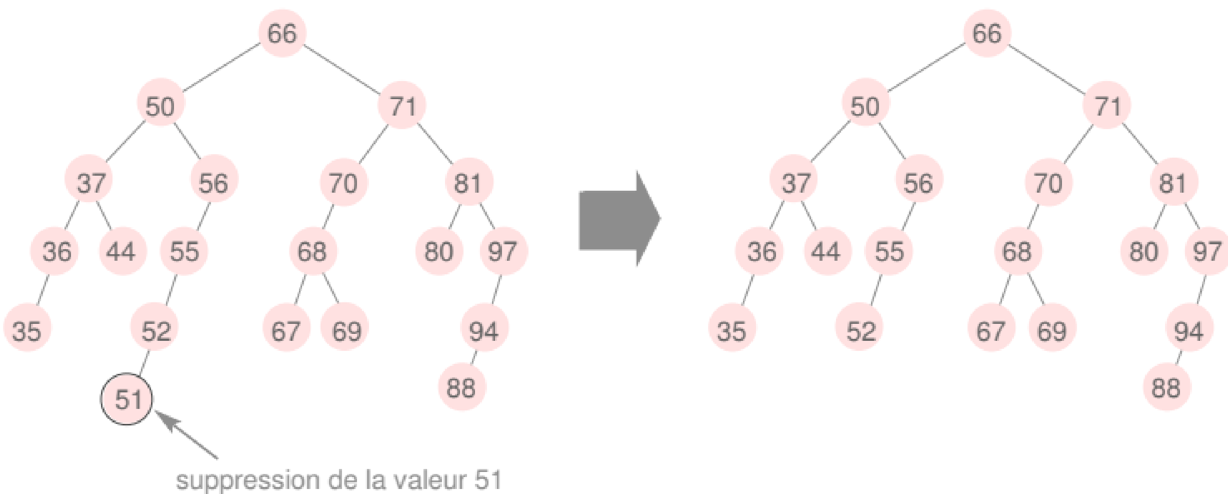


Figure 33 Deletion of a leaf node from BST

Case 2 : The node to delete **has only one child node**. That child node will be replacing the node in question. The parent of the node in question will be linked to its child node. Otherwise if the BST is composed of only this node and its child node. The child node becomes a root node for this BST, in other words, there is no need to link this child node to any other node.

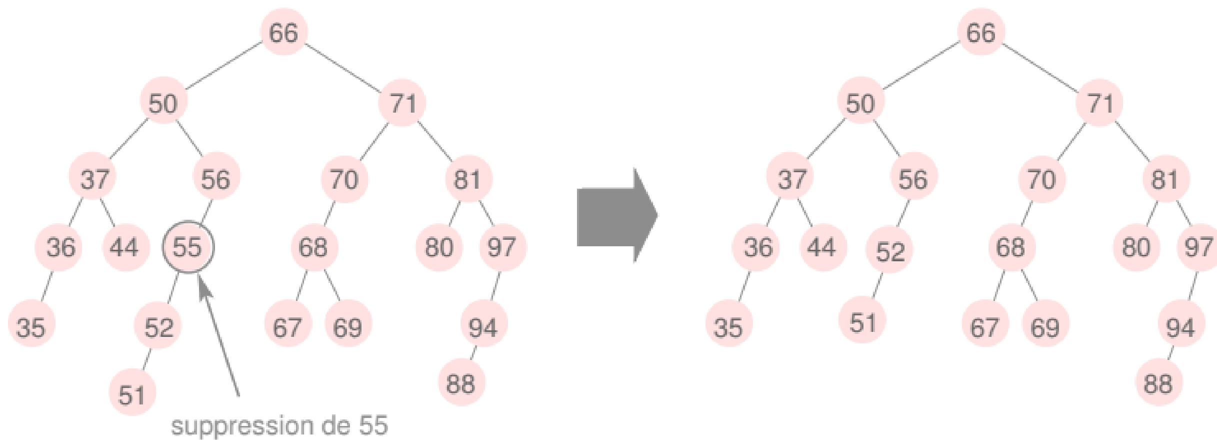


Figure 34 Deletion of an internal node having one direct child

Case 3 : If the node to delete **has two child nodes**. Either we replace this node with the node having the highest value in its left sub-tree or we replace it with the node having the lowest value in its right sub-tree.

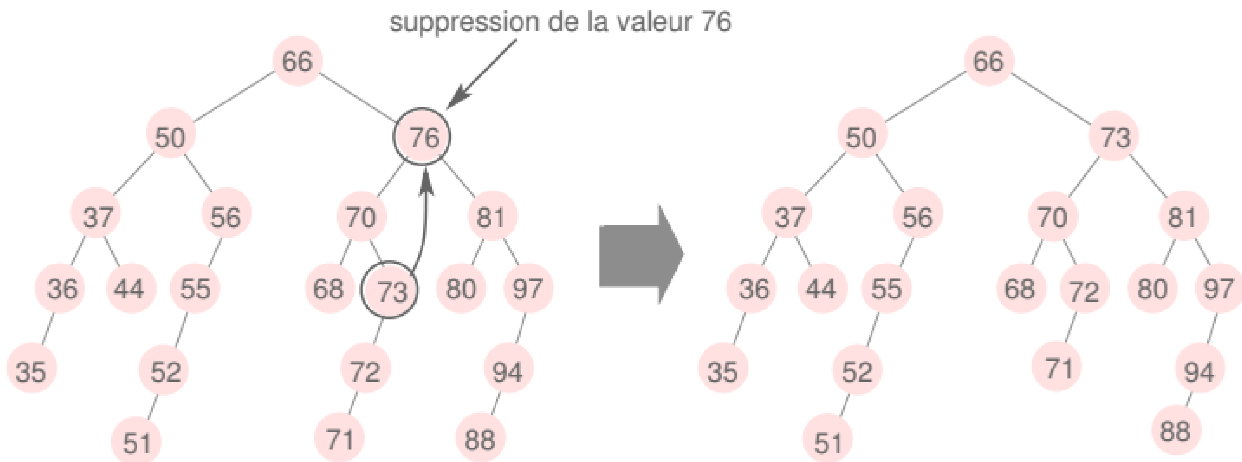


Figure 35 Deletion of a node having two child nodes

Exercise : considering all the three cases, suggest an algorithm for delete operation.

4.12. Operation of counting nodes

Given that any tree has a recursive nature, in other words, a tree is composed of a left sub-tree and a right sub-tree and a root node. The sum of nodes of a given BST can obtained by the addition of the sum of nodes in its left sub-tree with the sum of nodes in its left sub-tree plus one for accounting for the root node.

```
function count(root: ptr on node ) : integer
Begin
  if (root = null ) then return 0 ;
  return 1 + count(root.LeftC)+count(root.rightC)
End
```

4.13. Operation of search for successor of a given node

a successor of a given node ‘p’ is the node ‘q’ having the nearest highest value to node ‘p’. if node ‘p’ has a right child, its successor is the minimum value in the sub-tree whose root node is that right child. Otherwise if the node ‘p’ does not have a right child, its successor is its first ancestor located in the right of that node.

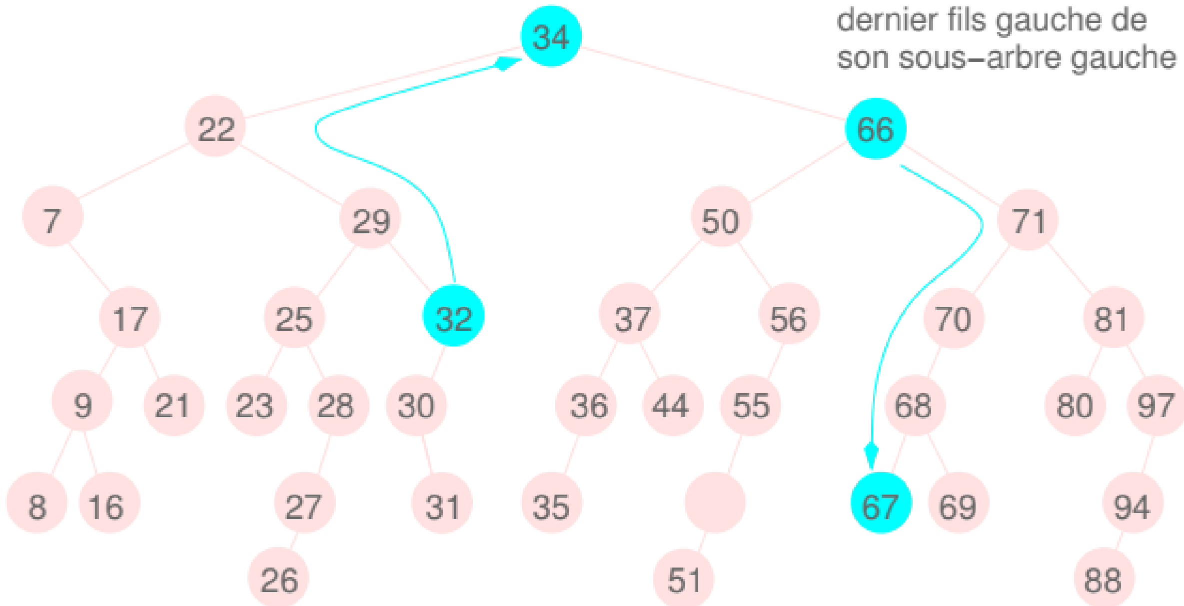


Figure 36 Inorder Successor cases

4.14. Conclusion

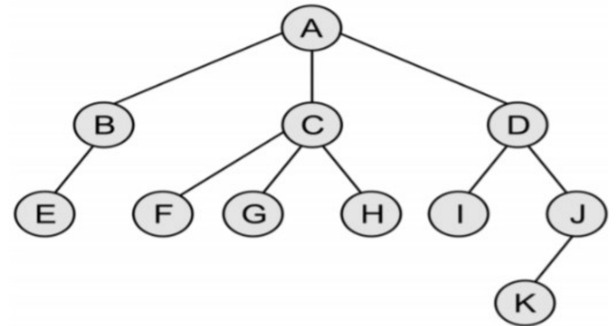
There are several types of trees that can be used as data structure, each having its own advantages and drawbacks. At the exception of heap trees, execution time of operations on other type tree is strongly related to the degree of unbalance of that tree. Insertion and deletion from a tree is the main cause of unbalance. Balancing a tree after each of these operations is an advanced topic which will be studied in an advanced course on Algorithms and data structures.

4.15. Exercises

Exercise 01

1) Convert the tree shown in figure into a binary tree

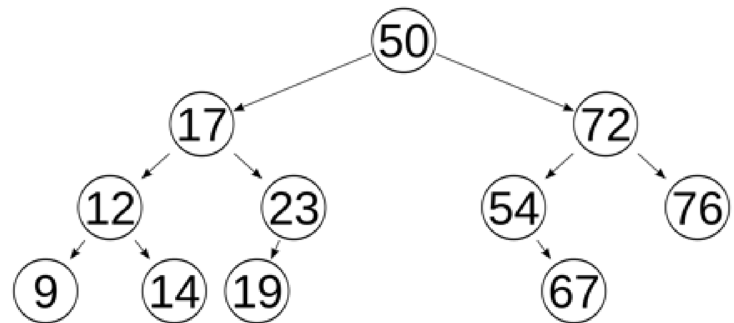
2) Give the result of the 4 traversals



3) Draw the max heap obtained from insertion of these values from this sequence : 25 60 35 10 5 20 65 45 70 40 50 55 30 15
4) Draw the max-heap obtained after deleting five times from the max-heap obtained in question 1.

Exercise 02

1- Redraw the obtained BST after insertion of two nodes with following values : 60, 63.
2- What is the height of the new tree ?
3- Is it a balanced tree ?
4- What is the result of the inorder traversal ?
5- Redraw the BST after deleting nodes whose values are : 76, 23 and 17.
6- Suggest an iterative form of the insertion operation.
7- Write an algorithm that count the number of nodes of a binary tree.
8- Write an algorithm that calculate the height of a binary tree.



Chapter 04 : Graphs

5. Chapter 04 : Graphs

5.1. Introduction

Graphs are natural models that are used to represent arbitrary relationships among data objects. We often need to represent such arbitrary relationships among the data objects while dealing with problems in computer science, engineering, and many other disciplines. Therefore, the study of graphs as one of the basic data structures is important.

5.2. Basic Definitions and Terminology

A graph is a structure made of two components: a set of vertices V, and a set of edges E. In other words, a graph is an ordered pair G = (V, E). The graph may be directed or undirected. In a *directed graph*, every edge of the graph is an ordered pair of vertices connected by the edge, whereas in an *undirected graph*, every edge is an unordered pair of vertices connected by the edge. Figure1 shows an undirected and a directed graph.

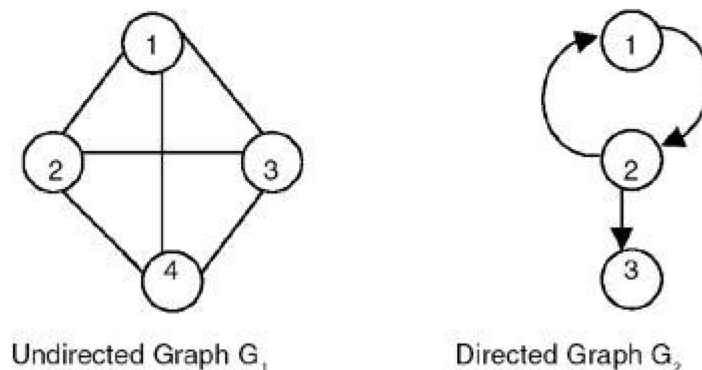


Figure 1 Undirected VS directed graphs

$G_1 = (V_1, E_1)$; $V_1 = \{1, 2, 3, 4\}$; $E_1 = \{ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) \}$, since G_1 is a directed graph , (x,y) is an unordered pair meaning (x,y) = (y,x).

$G_2 = (V_2, E_2)$; $V_2 = \{1, 2, 3 \}$; $E_2 = \{ (1,2), (2,1), (2,3)\}$, since G_2 is a directed graph , (x,y) is an ordered pair meaning $(x,y) \neq (y,x)$ if $x \neq y$.

Incident edge: (v_i,v_j) is an edge, then edge(v_i,v_j) is said to be incident to vertices v_i and v_j. For example, in graph G_1 shown in Figure 1, the edges incident on vertex 1 are (1,2), (1,4), and (1,3), whereas in G_2, the edges incident on vertex 1 are (1,2).

Degree of vertex: The number of edges incident onto the vertex. For example, in graph G_1, the degree of vertex 1 is 3, because 3 edges are incident onto it. For a directed graph, we need to define indegree and outdegree. *Indegree* of a vertex v_i is the number of edges incident onto v_i, with v_i as the head. *Outdegree* of vertex v_i is the number of edges incident onto v_i, with v_i as the tail. For graph G_2, the indegree of vertex 2 is 1, whereas the outdegree of vertex 2 is 2.

Chapter 04 : Graphs

Directed edge: A directed edge between the vertices v_i and v_j is an ordered pair. It is denoted by $<v_i,v_j>$.

Undirected edge: An undirected edge between the vertices v_i and v_j is an unordered pair. It is denoted by (v_i,v_j).

Path: A path between vertices v_p and v_q is a sequence of vertices v_p, v_{i1}, v_{i2},…, v_{in},v_q such that there exists a sequence of edges (v_p, v_{i1}), (v_{i1}, v_{i2}), …, (v_{in}, v_q). In case of a directed graph, a path between the vertices v_p and v_q is a sequence of vertices v_p, v_{i1}, v_{i2},…, v_{in}, v_q such that there exists a sequence of edges $<v_p, v_{i1}>$, $< v_{i1}, v_{i2}>$, …, $<v_{in}, v_q>$. If there exists a path from vertex v_p to v_q in an undirected graph, then there always exists a path from v_q to v_p also. But, in the case of a directed graph, if there exists a path from vertex v_p to v_q, then it does not necessarily imply that there exists a path from v_q to v_p also.

Simple path: A simple path is a path given by a sequence of vertices in which all vertices are distinct except the first and the last vertices. If the first and the last vertices are same, the path will be a cycle.

Maximum number of edges: The maximum number of edges in an undirected graph with n vertices is $n*(n-1)/2$. In a directed graph, it is $n*(n-1)$.

Subgraph: A *subgraph* of a graph G = (V, E) is a graph G' where V(G') is a subset of V(G). E(G') consists of edges (v_x,v_y) in E(G), such that both v_x and v_y are in V(G'). [Note: If G = (V,E) is a graph, then V(G) is a set of vertices of G and E(G) is a set of edges of G.]

If E(G') consists of all edges (v_x,v_y) in E(G), such that both v_x and v_y are in V(G), then G is called an induced subgraph of G. For example, the graph shown in Figure 2 is a subgraph of the graph G_2.

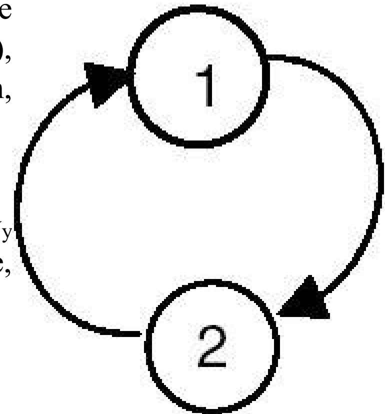


Figure 2 Subraph of graph 2

For the graph shown in Figure 3, one of the induced subgraphs is shown in Figure 4.

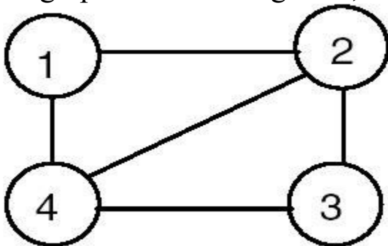


Figure 3 Graph G

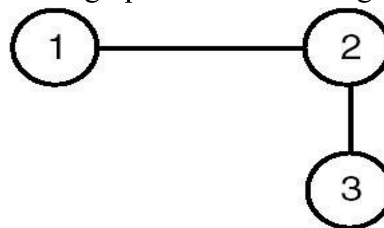


Figure 4 Induced subgraph of Graph G.

Chapter 04 : Graphs

In the undirected graph G, the two vertices v_1 and v_2 are said to be connected if there exists a path in G from v_1 to v_2 (being an undirected graph, there exists a path from v_2 to v_1 also).

Connected graph: A graph G is said to be connected if for every pair of distinct vertices (v_i,v_j), there is a path from v_i to v_j. A connected graph is shown in figure 5.

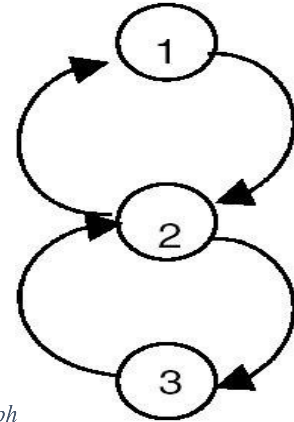


Figure 5 Connect Graph

Completely connected graph: A graph G is completely connected if, for every pair of distinct vertices (v_i,v_j), there exists an edge. A completely connected graph is shown in Figure 6.

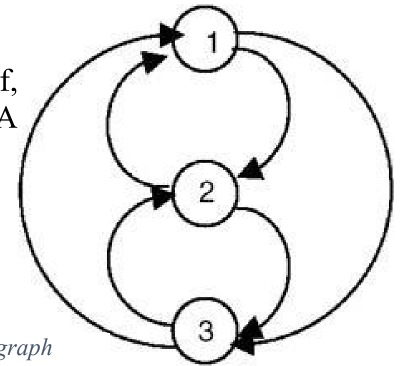


Figure 6 Completly connected graph

A self-loop is an edge starting and ending at the same vertex, in graph on the right side it has a self-loop for vertex 4.

Vertex 5 is a **multi-edge** vertex linking it with vertex 4.

A simple graph is a graph containing neither a self-loop nor a multi-edge.

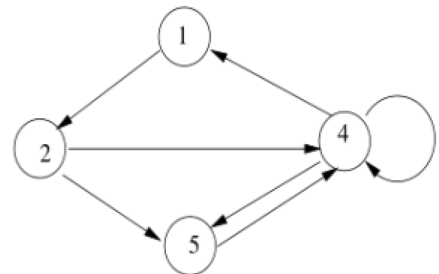


Figure 7 A graph containing self-loop and multi-edge

In a **Weighted graph**, each edge is assigned a weight value. As an example, inter-wilaya roads can be modelled as a weighted undirected graph, where the weight denotes the distance between each two cities (vertices).

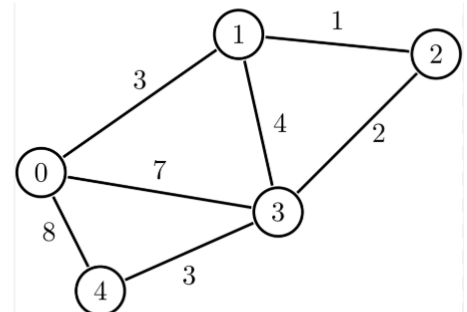


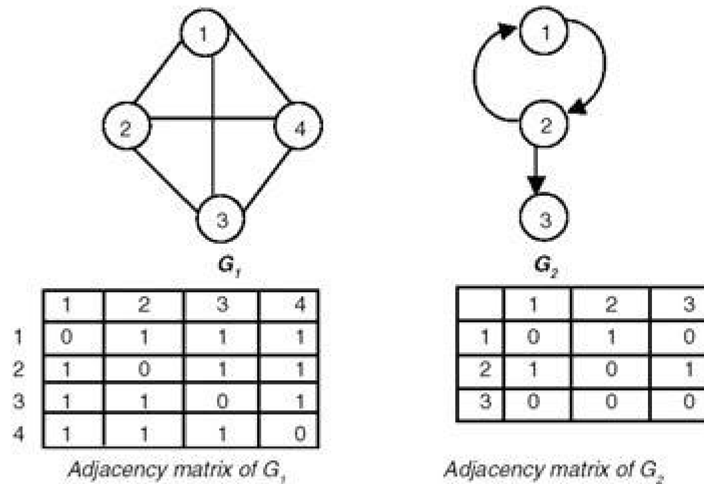
Figure 8 Example of a weighted graph

Chapter 04 : Graphs

5.3.REPRESENTATIONS OF A GRAPH

5.3.1. Array Representation

One way of representing a graph with n vertices is to use an n^2 matrix (that is, a matrix with *n* rows and *n* columns—that means there is a row as well as a column corresponding to every vertex of the graph). If there is an edge from v_i to v_j then the entry in the matrix with row index as v_i and column index as v_j is set to 1 (adj[v_i, v_j] = 1, if (v_i, v_j) is an edge of graph G). If e is the total number of edges in the graph, then there will 2*e entries which will be set to 1, as long as G is an undirected graph. Whereas if G were a directed graph, only e entries would have been set to 1 in the adjacency matrix. The adjacency matrix representation of an undirected as well as a directed graph is show in Figure 9.



	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Adjacency matrix of G_1

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

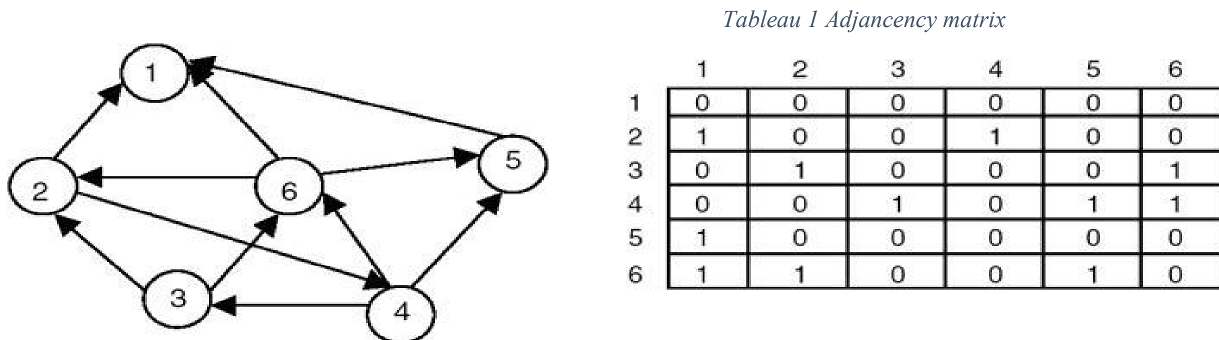
Adjacency matrix of G_2

Figure 9 Adjacency List

Note: if vertex keys are not natural numbers, an additional array of size n is created to store these keys.

Example The adjacency matrix representation of the following diagraph(directed graph), along with the adjacency matrix representation of the above diagraph are shown below:

Tableau 1 Adjancency matrix



	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	1	0	0	0	1
4	0	0	1	0	1	1
5	1	0	0	0	0	0
6	1	1	0	0	1	0

Figure 10 A directed graph

Adjacency matrix representation is only preferred when dealing with a dense graph, a graph having many edges between vertices of the graph. Otherwise, if the graph is a sparse graph (i.e. containing few edges between vertices of the graph), storage of the graph in this way is a waste of memory as many entries from the array having size n^2 will be just zeros.

In case of a weighted graph, the weight of the edge is assigned in the matrix. A very large or a very small value is assigned to entries between each v_x and v_y vertices not having an edge linking them.

5.3.2. Linked List Representation

Another way of representing a graph G is to maintain a list for every vertex containing all vertices adjacent to that vertex, as shown in figure 11.

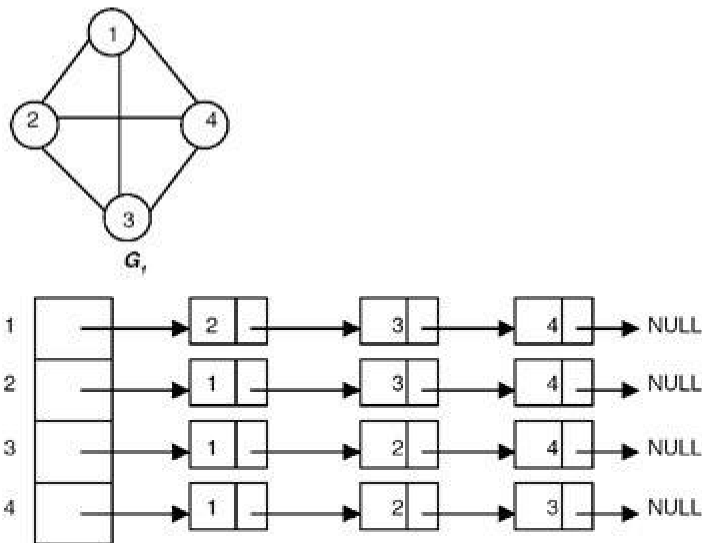


Figure 11 Graph G1 and its adjacency list

5.4.DEPTH-FIRST TRAVERSAL

5.4.1. Introduction

A graph can be traversed either by using the *depth-first traversal* or *breadth-first traversal*. When a graph is traversed by visiting the nodes in the forward (deeper) direction as long as possible, the traversal is called depth-first traversal. For example, for the graph shown in Figure 12, the depth-first traversal starting at the vertex 0 visits the node in the orders:

i. 0 1 2 6 7 8 5 3 4
ii. 0 4 3 5 8 6 7 2 1

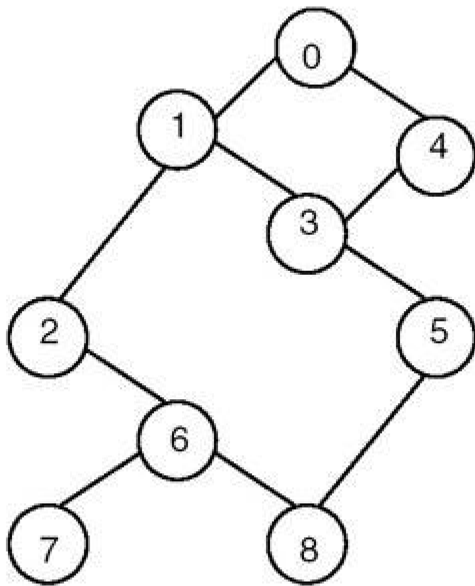


Figure 12 Graph G

A complete C program for depth-first traversal of a graph follows. It makes use of an array visited of *n* elements where *n* is the number of vertices of the graph, and the elements are Boolean. If visited[i] = 1 then it means that the ith vertex is visited. Initially we set visited[i] = 0.

5.4.2. Program

```
#include <stdio.h>
#define max 10

/* a function to build adjacency matrix of a graph */
void buildadjm(int adj[][max], int n){
        int          i,j;
        for(i=0;i<n;i++)
                  for(j=0;j<n;j++)
                   {
           printf("enter 1 if there is an edge from %d to %d, otherwise enter 0 \n",
            i,j);

           scanf("%d",&adj[i][j]);
                   }
      }

   /* a function to visit the nodes in a depth-first order */
   void dfs(int x,int visited[],int adj[][max],int n){
      int         j;
      visited[x] =1;
      printf("The   node   visited   id   %d\n",x);
      for(j=0;j<n;j++)
         if(adj[x][j]      ==1      &&      visited[j]      ==0)
              dfs(j,visited,adj,n);
```

```
    }

void main(){
      int adj[max][max],node,n;
      int i, visited[max];
      printf("enter the number of nodes in graph maximum = %d\n",max);
scanf("%d",&n);
      buildadjm(adj,n);
      for(i=0; i<n; i++)
           visited[i] =0;
         for(i=0; i<n;
           i++)
          if(visited[i]==0)
               dfs(i,visited,adj,n)
               ;
}
```

5.4.3. Explanation

1. Initially, all the elements of an array named `visited` are set to 0 to indicate that all the vertices are unvisited.
2. The traversal starts with the first vertex (that is, vertex 0), and marks it visited by setting `visited[0]` to 1. It then considers one of the unvisited vertices adjacent to it and marks it `visited`, then repeats the process by considering one of its unvisited adjacent vertices.
3. Therefore, if the following adjacency matrix that represents the graph of 12 is given as input, the order in which the nodes are visited is given here:
   - Input: 1. The number of nodes in a graph
     2. Information about edges, in the form of values to be stored in adjacency matrix 1 if there is an edge from node i to node j, 0 otherwise
   - Output: Depth-first ordering of the nodes of the graph starting from the initial vertex, which is vertex 0, in our case.

5.4.4. Analysis

1. If the graph G to which the Depth-First Search (DFS) is applied is represented using adjacency lists, then the vertices y adjacent to x can be determined by following the list of adjacent vertices for each vertex.
2. Therefore, the `for` loop searching for adjacent vertices has the total cost of $d_1 + d_2 + \ldots + d_n$, where d_i is the degree of vertex v_i, because the number of nodes in the adjacency list of vertex v_i are d_i.
3. If the graph G has n vertices and e edges, then the sum of the degree of each vertex ($d_1 + d_2 + \ldots + d_n$) is 2e. Therefore, there are total of 2e list nodes in the adjacency lists of G. If G is a directed graph, then there are a total of e list nodes only.
4. The algorithm examines each node in the adjacency lists once, at most. So the time required to complete the search is O(e), provided $n <= e$. Instead of using adjacency lists, if an adjacency matrix is used to represent a graph G, then the time required to determine all adjacent vertices of a vertex is $O(n)$, and since at most n vertices are visited, the total time required is $O(n^2)$.

5.5.BREADTH-FIRST TRAVERSAL

5.5.1. Introduction

When a graph is traversed by visiting all the adjacent nodes/vertices of a node/vertex first, the traversal is called breadth-first traversal. For example, for a graph in which the breadth-first traversal starts at vertex v_1, visits to the nodes take place in the order shown in Figure 22.10.

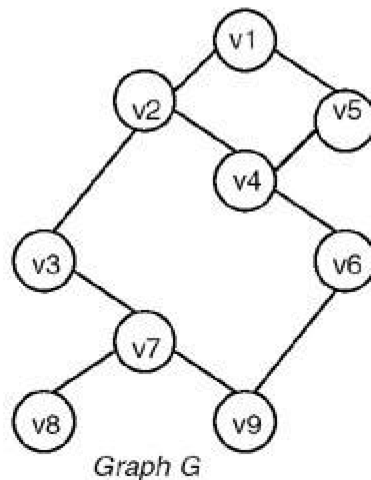


Figure 13 BFS result on the related graph

5.5.2. Program

A complete C program for breadth-first traversal of a graph appears next. The program makes use of an array of *n* visited elements where *n* is the number of vertices of the graph. If `visited[i] = 1`, it means that the ith vertex is visited. The program also makes use of a queue and the procedures `addqueue` and `deletequeue` for adding a vertex to the queue and for deleting the vertex from the queue, respectively. Initially, we set `visited[i] = 0`.

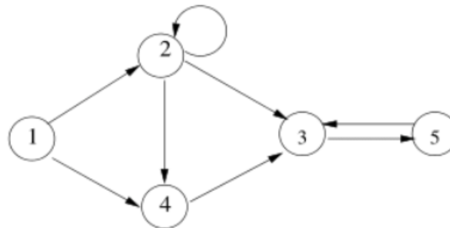
```
#include   <stdio.h>
#include  <stdlib.h>
#define    MAX    10
struct node{
      int data;
      struct node *link;
      };
void buildadjm(int adj[][MAX], int n){
      int i,j;
      printf("enter      adjacency      matrix      \n",i,j);
for(i=0;i<n;i++)
      for(j=0;j<n;j++)
```

```
      scanf("%d",&adj[i][j]);
}
/* A function to insert a new node in queue*/ struct
node *addqueue(struct node *p,int val){
            struct    node
      *temp; if(p == NULL)
      {
      p = (struct node *) malloc(sizeof(struct node));
/* insert the
new node first node*/
if(p == NULL)
{
      printf("Cannot allocate\n"); exit(0);
}
      p->data  =val;
p->link=NULL;
}
else{
          temp= p;
         while(temp->link != NULL)
         {
          temp = temp->link;
         }
      temp->link = (struct node*)malloc(sizeof(struct node));
      temp = temp->link;
            if(temp == NULL){
                 printf("Cannot      allocate\n");
                 exit(0);
                 }
      temp->data =val;
      temp->link=NULL;
         }
         return(p);
         }
struct node *deleteq(struct node *p,int *val){
      struct  node  *temp;
if(p == NULL){
           printf("queue is empty\n");
           return(NULL);
            }
*val = p->data;
temp = p;
             p  = p->link;
       free(temp);
       return(p);
           }
void bfs(int adj[][MAX], int x,int visited[], int n, struct node **p){
   int y,j,k;
      *p = addqueue(*p,x);
do{

               *p = deleteq(*p,&y);
               if(visited[y] == 0){
                 printf("\nnode    visited    =    %d\t",y);
                 visited[y] = 1;
      for(j=0;j<n;j++)
      if((adj[y][j] ==1) && (visited[j] == 0))
```

```
      *p = addqueue(*p,j);
      }

}while((*p) != NULL);}

void main(){
      int adj[MAX][MAX];

      int n;
        struct node *start=NULL;
        int i, visited[MAX];
        printf("enter the number of nodes in graph maximum = %d\n",MAX);
        scanf("%d",&n);
        buildadjm(adj,n);
        for(i=0; i<n; i++)
           visited[i] =0;
        for(i=0; i<n; i++)
          if(visited[i]==0)
               bfs(adj,i,visited,n,&start);
      }
```

5.6.Exercises

1) Draw the oriented graph given by G = (V,E) , V = {1,2,3,4,5} E={ (x,y), (x,y), (x,y), (x,y), (x,y), (x,y), (x,y), (x,y), }
2) Give the corresponding adjacency matrix and adjacency list representing graph in question 1
3) What is the result of DFS and BFS of graph in question 1.
4) Give the adjacency matrix representation of the graph below



5.7.Conclusion

Graphs as data structures has many real world application, knowing the classification of graphs, its way of storage in memory and some number of popular algorithms for operations on graph, is crucial to the deeper understanding of related courses especially the computer networking course. The concepts covered in the chapter will help the students to follow a more advanced course in algorithms and data structures.

General Conclusion

General Conclusion

This course has provided a comprehensive foundation in data structures and algorithms, equipping you with the tools needed to design efficient, reliable, and scalable solutions to computational problems. By studying **algorithmic complexity**, you learned how to evaluate performance and make informed choices based on time and space constraints rather than intuition alone.

Through hands-on exploration of **core data structures**, including **trees** and **graphs**, you developed an understanding of how data organization impacts problem-solving. These structures enabled you to model hierarchical systems, networks, and relationships that frequently arise in real-world applications. The study of **sorting algorithms** further reinforced the importance of efficiency and trade-offs, highlighting how different approaches can be optimal in different contexts.

Beyond individual topics, the course emphasized **algorithmic thinking**—breaking problems down, identifying patterns, and selecting the most appropriate structures and techniques. These skills extend far beyond this course and are fundamental to software engineering, competitive programming, and technical interviews.

By completing this course, you are now prepared to approach complex problems with confidence, analyze solutions critically, and continue learning more advanced topics in computer science. Data structures and algorithms are not just academic concepts—they are essential tools that will support your growth as a developer and problem solver throughout your career.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press.

[2] Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-wesley professional.

[3] Bhargava, A., & Bhargava, A. Y. (2024). Grokking algorithms. Simon and Schuster.

[4] Djelloul BOUCHIHA. Algorithmics and C Programming: Lectures, Solved Exercises, and Practical Work. Publisher: LAP LAMBERT Academic Publishing. Copyright © 2024 Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L Publishing Group. ISBN: 978-620-7-47639-8