Democratic and Popular Republic of Algeria
Ministry of Higher Education and Scientific Research

University of Mustapha Stambouli, Mascara

Faculty of Exact Sciences

Algorithmics and Data Structures 1

Lecture notes and solved exercises

Dr Mohammed ZAGANE

Preamble: These lecture notes are intended for first-year students in Computer Science as well as first-year students in Mathematics. The objective is to introduce students to algorithmics and data structures. The content of this course is organized into six chapters:

- Chapter 1 begins the course with a brief history of computer science and an introduction to the main concepts of algorithmics,
- Chapter 2 details simple sequential algorithms, covering the parts of an algorithm, data types (variables and constants), and more,
- Chapter 3 focuses on conditional structures, both in algorithmic language and in the C programming language,
- Chapter 4 explains loops and their applications,
- Chapter 5 presents arrays and strings,
- Chapter 6, the final chapter, introduces user-defined types.

At the end of each chapter, exercises are provided to help students practice and deepen their understanding. The solutions to these exercises are included in Appendix 1.

Contents

Preamble i

Contents ii

Chapter 1 ....

Introduction........

1.1 Brief historic of computer science 7

1.1.1 Definitions........ 7

A) Computer science 7

B) Automatic information processing 7

C) Computer........ 8

1.1.2 Brief historic of computer science 8

1.2 Introduction to algorithmics........ 13

1.2.1 Algorithmics........ 13

1.2.2 Algorithm 13

1.2.3 Concept of language and algorithmic language 14

A) Formal language vs Natural language........ 14

B) Programming language........ 14

C) Machine language........ 15

D) Algorithmic language........ 15

Chapter 2 ....

Simple sequential algorithm ....

2.1 Introduction........ 17

2.2 Algorithm parts........ 17

2.2.1 Input 17

2.2.2 Output........ 17

2.2.3 Processing 17

2.3 Data : variables and constants........ 19

2.3.1 Variables 19

2.3.1 Constants 19

2.4 Data types 20

2.5 Basic operations 21

2.5.1 Operators and operands 21

2.5.2 Arithmetic operators 21

2.5.3 Operator precedence 22

2.6 Basic instructions 23

2.6.1 Assignment 23

2.6.2 I/O instructions 23

A) Read statement 24

B) Write statement 24

2.7 Construction of a simple algorithm 25

2.7.1 Algorithm header 26

2.7.2 Algorithm body 27

2.8 Graphic representation of an algorithm (organigram) 28

2.9 Translation to C program 29

2.9.1 The C Programming language 30

2.9.3 Why C ? 30

2.9.2 Structure of a C Program 30

2.9.4 Translation from Algorithm to C Program 31

2.10 Exercises 33

Chapter 3 ....

Conditional structures ....

3.1 Introduction 37

3.2 Boolean expressions 37

- Logical Operators: 38

Example 3.2 : 38
3.3 Simple conditional structure 39
3.4 Nested conditional structures 40
3.5 Multiple choice conditional structure 42
3.5.1 if-elseif statement 42
3.5.2 switch-case statement 44
3.6 Exercises 47
Chapter 4 ....
Loops ....
4.1 Introduction 50
4.2 While loop 50
4.3 Repeat loop 51
4.4 For loop 53
4.5 Nested loops 54
4.6 Exercises 56
Chapter 5 ....
Arrays and strings ....
5.1 Introduction 59
5.2 Arrays 59
5.2.1 Definition 59
5.2.2 Unidimensional arrays 59
A) Declaration 60
B) Access for read/write 60
5.2.3 Multi-dimensional arrays 61
A) Declaration 62
B) Access for read/write 63
5.3 Strings 64

5.3.1 Definition 64
5.3.2 Declaration 64
5.3.3 Manipulation 65
A) I/O operations 65
B) Assignment 66
C) Concatenation 67
5.4 Exercises 70
Chapter 6 ....
User-defined types...
6.1 Introduction... 75
6.2 Enumerations 75
6.2.1 Enum definition (creation) 75
6.2.2 Enum usage 76
6.3 Records 78
6.3.1 Record Definition (Creation) 78
6.3.2 Record usage 79
6.4 Exercises 81
References... 83
Appendix 1: Solutions to Exercises 85

Chapter 1

Introduction

1.1 Brief historic of computer science
1.2 Introduction to algorithmics

1.1 Brief historic of computer science

1.1.1 Definitions

A) Computer science

Computer science is the field of science that focuses on the **automatic information processing** using **computers**.

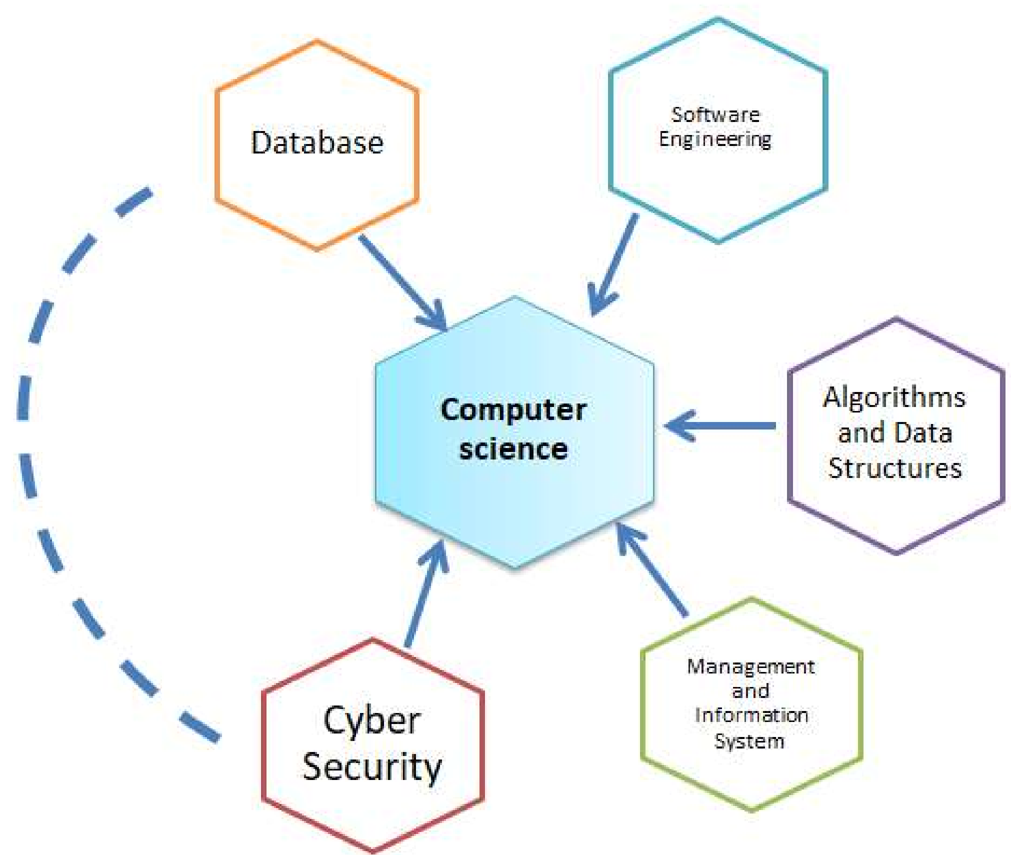


Figure 1: Examples of computer sciences fields.

B) Automatic information processing

Automatic information processing is the operations performed by a computer on the information introduced to it by the user.

- **Automatic:** Performed by a machine (in our case, a computer).

- **Information:** Facts, data, or knowledge that the user can input into the computer. This can include various forms such as numerical data, text, multimedia (image, sound) and represented and manipulated by the computer in binary (a sequence of 0 and 1).
- **Processing:** This involves a series of actions performed on the information, beginning with its acquisition (through input devices such as keyboards, cameras, microphones, etc.), followed by storage in computer memory units. It includes also tasks like calculations, handled by the processor, and preparing the information for output (utilizing components like graphic cards and sound cards). Finally, the processed information is delivered to the user through output devices such as printer, monitor, and speaker.

C) Computer

A computer is an electronic and programmable machine that performs automatic information processing.

- **Electronic:** it's built using electronic elementary components such as transistors and integrated circuits just like other electronic machines: calculator, TV,…
- **Programmable:** It performs the information processing by following a set of instructions (a program) stored in memory units.

1.1.2 Brief historic of computer science

The modern field of computer science emerged as a distinct academic discipline in the 1960's. The earliest mechanical devices (composed of gears, belt and pulley) used to aid calculation, and the subsequent calculation machines and the mathematical theories behind them, were initially developed primarily within the fields of physics and mathematics. In the following, the most important dates are highlighted :

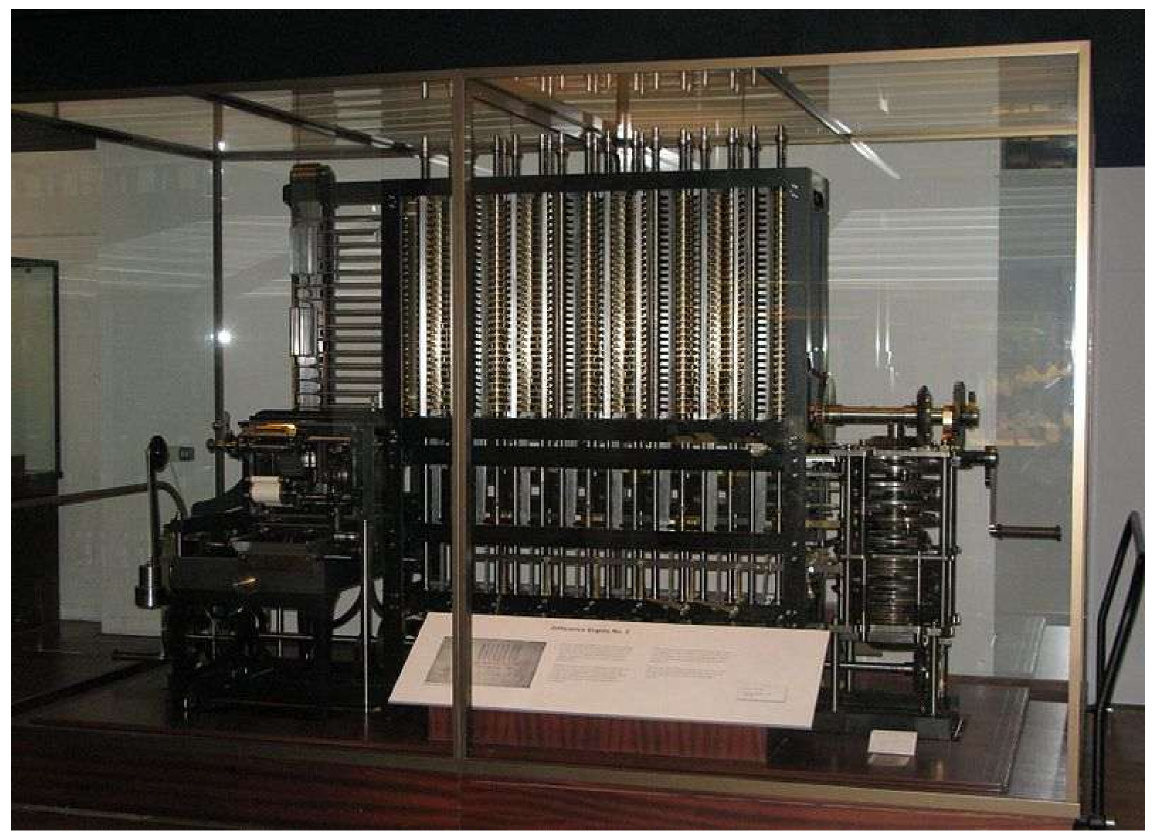
Early BCE period: The ancient Babylonians and Greeks developed mechanical devices to aid calculation such as the Abacus and the Antikythera mechanism,

Figure 2: Derek J. de Solla Price (1922–1983) with a model of the Antikythera mechanism [Web1].

1610: John Napier (1550-1617), the Scottish inventor of logarithms, invented Napier's rods to simplify the task of multiplication,

1641: the French mathematician Blaise Pascal (1623-1662) built a mechanical adding machine,

1822: Charles Babbage (1791-1871) designed the Difference Engine. The name difference engine is derived from the method of divided differences, a way to interpolate or tabulate functions by using a small set of polynomial coefficients.



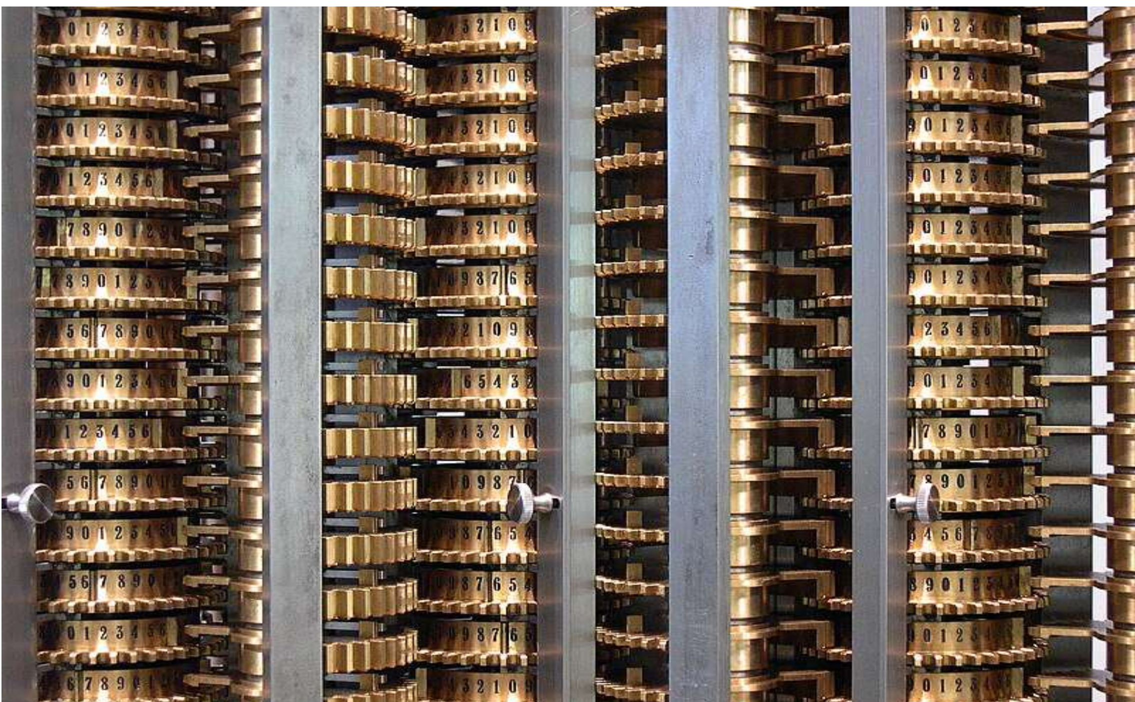


Figure 3: The London Science Museum's difference engine [Web2] [Web3].

1919: E. O. Carissan (1880-1925), designed and had built a mechanical device for factoring integers and testing them for primality.

Figure 4: the Carissan machine [Web4].

1936: Alan Turing (1912-1954) proposed the Turning machine which is a mathematical model of computation describing an abstract machine (it is similar to a mathematical function in that it receives inputs and produces outputs based on predefined rules).

1944: Howard H. Aiken (1900-1973) built the first electromechanical computation machine (the Mark I).

1945: John von Neumann (1903-1957) proposed the von Neumann architecture, the first model for the electronic digital computer that is still used in recent computers.

1950's: Rise of electronic computers

In this decade, computers have known several developments such as :

- the invention of the Transistor and integrated circuits which helped in minimizing the size of computers,

- the invention of the notion of a compiler and the Fortran and Lisp programming languages,
- Alan Turing introduced the Turing Test, one of the first efforts in the field of artificial intelligence.

1960's : The Rise of Computer Science

Computer science emerged as a distinct field of study in the beginning of this decade. Purdue University established the first computer science department in 1962. Additionally, towards the end of this decade, ARPANET, a precursor to the modern internet, began development.

1970's: Borne of Unix and C

In this decade, significant advancements were made in database theory, particularly on relational databases. Unix, a highly influential operating system, was developed at Bell Laboratories. Additionally, Brian Kernighan and Deniss Ritchie collaborated to create the C programming language. This decade also witnessed major breakthroughs in algorithms and computational complexity.

1980s: The Personal Computer Revolution

- The 1980s saw a surge in the popularity and affordability of personal computers, driven by companies like Apple and IBM,
- Computer Viruses: The first computer viruses emerged during this time, highlighting the growing security concerns,
- Operating Systems: New operating systems like MS-DOS and the Macintosh operating system were introduced.
- Networking: Local area networks (LANs) became more common, connecting computers within organizations.

1990s: The Internet Era

- Internet Growth: The internet experienced explosive growth, connecting more and more people and computers worldwide,

- World Wide Web: The World Wide Web, developed by Tim Berners-Lee, made it easy to access and share information online,
- E-commerce: Online shopping and e-commerce began to take off, revolutionizing retail.

Beyond 2000s: The Digital Age

- Mobile Computing: The rise of smartphones and tablets transformed how people interact with technology,
- Cloud Computing: Cloud computing services made computing resources accessible over the internet,
- Social Media: Platforms like Facebook, Twitter, and Instagram became integral parts of social life,
- Artificial Intelligence: Advances in AI led to breakthroughs in machine learning, natural language processing, and robotics,
- Internet of Things (IoT): Connected devices and sensors began to proliferate, creating the IoT.

The computer science continues to evolve rapidly, with new innovations and developments emerging constantly.

1.2 Introduction to algorithmics

1.2.1 Algorithmics

Algorithmics is the branch of computer science that deals with methods and concepts of studding algorithms.

1.2.2 Algorithm

An algorithm is a step-by-step procedure for solving a problem. It is a finite sequence of well-defined instructions that, when followed, will produce a solution to the problem. Algorithms are typically expressed in an informal language: naturel language (Example: Arabic description of the steps to follow to solve a problem), pseudocode (Set of

instructions in a simplified version of a programming language) or a flowchart diagrams (a graphic representation that show the different steps of solving a problem).

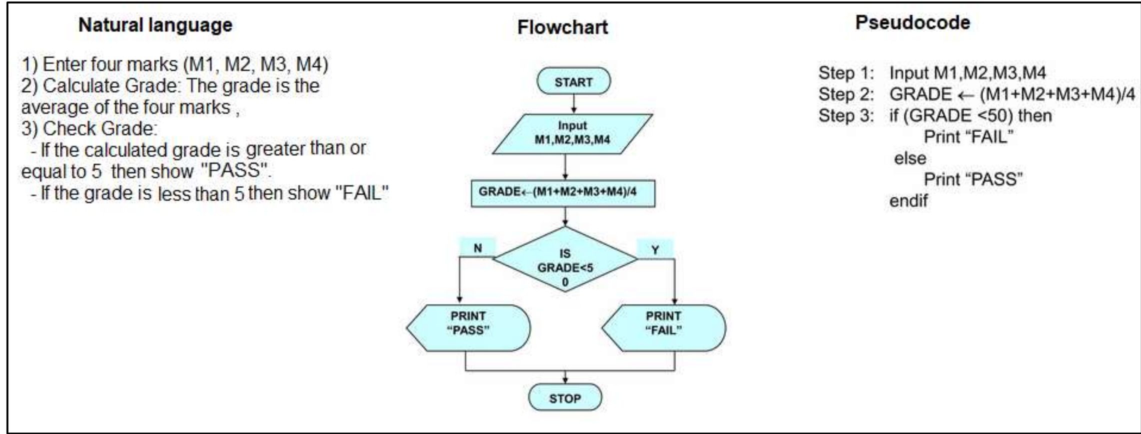


Figure 5: Example of algorithm representations (modified from [Web5]).

1.2.3 Concept of language and algorithmic language

A) Formal language vs Natural language

Formal languages are precisely defined systems of symbols and rules used to communicate specific information or instructions. They are often used in mathematics, computer science, and logic to express complex ideas and relationships in a clear and unambiguous manner. Natural languages, on the other hand, are spoken or written languages used for human communication. They are more flexible and context-dependent, allowing for a wider range of expressions and interpretations. While both formal and natural languages serve as communication tools, formal languages are more structured and precise, while natural languages are more nuanced and expressive.

B) Programming language

A programming language is a formal language used to communicate instructions to a computer. Programming languages are designed to be more human-readable than machine language, which is a sequence of binary digits that directly instruct the computer's hardware.

Historically, the first programs were written in machine language. However, as programs became more complex, machine language proved to be cumbersome and difficult to write and understand. This led to the development of higher-level programming languages that are easier for humans to read, write, and maintain. These higher-level languages abstract away the complexities of machine language, allowing programmers to focus on the logic and problem-solving aspects of their code. As a result, programming languages have evolved significantly over time, with new languages emerging to address specific needs and challenges.

C) Machine language

Machine language is the most basic level of programming language. It consists of a sequence of binary digits (0s and 1s) that directly instruct the computer's hardware. Each instruction corresponds to a specific operation that the computer can perform.

D) Algorithmic language

An algorithmic language is a language that can be used to express algorithms in a clear and concise manner. Unlike formal languages like mathematics and programming languages, algorithmic languages are often more informal and less rigid. They are designed to be easily understood by humans, without the strict syntax and rules of formal languages. Algorithmic languages are typically based on mathematical notation, but they may also incorporate elements of natural language. This informality makes them well-suited for describing algorithms in a way that is easy to understand and analyze, even for those who may not have a strong background in computer science or mathematics.

Chapter 2

Simple sequential algorithm

2.1 Introduction

2.2 Algorithm parts

2.3 Data: variables and constants

2.4 Data types

2.5 Basic operations

2.6 Basic instructions

2.6.1 Affectation

2.6.2 I/O instructions

2.7 Construction of a simple algorithm

2.8 Graphic representation of an algorithm (organigram)

2.9 Translation into C programing language

2.1 Introduction

Computers process information (process input data to get output data) following a set of instructions called a program.

Figure 6: Computer process input via program to get output.

Before writing the program to solve a problem (for example find the roots of a quadratic equation automatically by a computer), the problem must be analyzed to:

- Identify the data: input data, constants and output data (results)
- Identify the processing (the operations) to be done on the input to get the output

Then after identifying the data and the processing, the algorithm that solve the problem can be written and translated using a programming language to a program executable by the computer.

2.2 Algorithm parts

2.2.1 Input

This represents any **received** data from the **outside** to be processed by the algorithm.

2.2.2 Output

This represents the results that the algorithm must **send or provide to** the **outside** after processing the input.

2.2.3 Processing

This represents any instructions that the algorithm must execute:

- Acquire inputs from the outside (input instructions),
- Process the inputs (calculation (or arithmetic) instructions),
- Sending the outputs back to the outside (output instructions)

It is important to note that the order in which instructions are executed can be :

- Sequential: Instructions are executed one after another,
- Conditional: Instructions are executed based on a condition (conditional structures).
- Iterative: A block of instructions is repeated until a condition is met (loops).

We will discuss conditional structures and loops in details in chapter 3 and chapter 4 respectively.

Example 2.1:

As an example, suppose that we need to calculate the perimeter (circumference) of a circle. We know that to calculate the perimeter of a circle we have first to know the value of its radius; then we can use the formula: **perimeter = 2 x π x Radius** where π (Pi) is a mathematical constant that approximately equal to 13.4.

In this example:

- The input is : the radius of the circle,
- The output is : the perimeter,
- The processing is :
    - read(radius); an input operation to acquire the radius of the circle from the outside,
    - perimeter ← 2 * Pi * radius; this instruction includes two operations : calculation operation (*) and assignment operation (←),
    - write(perimeter); an output operation : showing in the monitor, printing by the printer, etc.) to send the result to the outside.

2.3 Data : variables and constants

In the context of algorithms, variables and constants are essential components for representing and manipulating data.

2.3.1 Variables

Variables are placeholders for values that can change during the execution of an algorithm or between different instances of it. They are used to store inputs, intermediate results and output. Every variable must be given a name that allows it to be identified uniquely in the algorithm. A variable name is generally a series of alphanumeric characters, the first of which is alphabetical.

Example 2.2: in the previous example of calculating the perimeter of a circle, the values that can change are the radius of the circle which can be represented by the variable: **radius** and also the perimeter of the circle which can be represented by the variable: **perimeter**.

Example 2.3: in an algorithm of finding the roots of a quadratic equation ($a.X^2 + b.X + c = 0$), the variables to be used are:

- **a**, **b** and **c** that represent the equation coefficients (input variables),

- **delta** that represent the discriminant delta **(**intermediate variables**)**

- **x1**, **x2** that represent the roots of the equation (output variables).

2.3.1 Constants

Constants are values that remain unchanged throughout the execution of an algorithm. Just like variables, constants also must have names to identify them in the algorithm.

Example 2.4: in the example of calculating the perimeter of a circle, the value that remain unchanged is the parameter π which can be represented by the constant **Pi**.

IMPORTANT: While constants are often useful in algorithms, they are not strictly necessary, and we can replace them by immediate values. As example, we can use the value of Pi directly as follow : Perimeter ← 2 x 3.14 x radius;

2.4 Data types

The manipulated data in an algorithm can be of one of the following primitive types:

- **integer**: this type is used to represent integer numbers (examples : -104 , 0, 5 etc),
- **real**: this type is used to represent real number with decimal point (-203.5, 5.8, etc). A **real** variable can take an **integer** value because the set of integer number is included in the set of real number.
- **char** : this type is used to represent single character data (examples : ‘t’, ‘d’, ‘#’, ‘@’ , etc).
- **boolean**: this type represents logic values (**true** or **false**).

These primitive data types form the foundation of more complex data structures like arrays, strings, records and lists which can represent various kinds of information: numerical, text or multimedia (image, sound and videos).

The type of data to use in variables is defined in the problem analysis phase.

Example 2.5:

Identify data (variables and constants) and their types for the following problems : calculation of the perimeter of a circle and calculation of the roots of a quadratic equation.

1) Calculation of the perimeter of a circle :

Variables :

Input : radius (real) ,

Output: perimeter (real)

in this problem there is no intermediate variables

Constants :

Pi = 3.14

2) Calculation of the roots of a quadratic equation (a.X2 + b.X + c = 0) :

Variables :

Input : a,b,c (real),

Intermediate variable: delta (real)

Output: x1,x2 (real)

Constants :

In this problem there is no need to use constants

2.5 Basic operations

Each instruction in the algorithm can perform one or more operations. Arithmetic operations that perform mathematical calculations on numerical data such as addition, subtraction, etc. represent the basic operations in algorithms.

2.5.1 Operators and operands

An operation is represented by an operator and one or more operands.

- The operator is a symbol that represents an action to be performed on the data,
- The operand is the data on which the operator acts and can be a variable, a constant or a call of a function.

For example, in the operation: **var1 + 5**, '+' is the operator, and the variable '**var1**' and the constant 5 are the operands.

In this section we will discuss only arithmetic operators, comparison (=, <, > , etc) and Boolean (AND, OR, etc) operators used in conditional instructions are discussed in the next chapter.

2.5.2 Arithmetic operators

Arithmetic operations can be formed by **arithmetic operators**. Common operators include: + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus) which gives the remainder of a division and ^ (Exponentiation).

A calculation instruction can involve one or more operations, forming an **arithmetic expression**. For example, in the expression **var1 + var2 * 5**, there are two operators: + and *. Each arithmetic expression can be evaluated to a final value. The data type of the final value (integer or real) depends on the data types of the operands involved. If all operands are integers, the result will be an integer. However, if at least one operand is a real number, the final result will be a real number.

2.5.3 Operator precedence

When evaluating arithmetic expressions, it's crucial to pay attention to the order in which operations are performed. Operators with higher precedence are evaluated before those with lower precedence. The standard order of precedence is as follows:

1) Parentheses: Expressions within parentheses are always evaluated first,
2) Exponentiation: Operations involving exponents (e.g., 2^3) are performed next,
3) Multiplication and Division: These operations have equal precedence and are evaluated from left to right,
4) Addition and Subtraction: These operations also have equal precedence and are evaluated from left to right.

Example 2.6:

- 7 + 5 * 6 is evaluated as 7 + 30 = 37 because multiplication has higher precedence than addition.
- (7 + 5) * 6 is evaluated as 12 * 6 = 72 because the expression within parentheses is evaluated first.
- 5^2 * 4 is evaluated as 25 * 4 = 100 because exponentiation has higher precedence than multiplication.

2.6 Basic instructions

In addition to instructions that involve mathematical calculation, algorithms often involve two fundamental types of instructions: assignment and input/output (I/O) instructions.

2.6.1 Assignment

An assignment instruction involves the operation of assigning a value to a variable. This operation is represented by the assignment operator (⟵**)**.

The syntax of an assignment instruction is :

```
variable_name ← value;
```

And here we say that : "the **variable** receives the **value"** and not "the variable equal to the value".

The value to be assigned to a variable can be:

- **An immediate value (a constant):** var1 ⟵ 25;
- **Another variable:** var1 ⟵ var2;
- **An expression:** var1 ⟵ (var2 + var3) * 2;
- **A call of a function or procedure:** var1 ⟵ some_function();

It is crucial to not confuse between the assignment operator (⟵) and the equality operator (=) which is a comparison operator used to compare two value and when evaluated it return a Boolean value (true or false). We will discuss the equality operator in the next chapter (Conditional structures).

2.6.2 I/O instructions

Input/Output instructions allow to perform I/O operations (exchanging information between the algorithm and the outside (mainly the user)). The most part of the I/O operations are performed by the statements: **read** and **write.**

A) Read statement

The read statement allows to acquire inputs from the input device of the computer (mainly the keyboard). This is done by assigning the entered value to a variable. The syntax of the read statement is :

```
read(variable_name);
```

Example 2.7:

In the previous problem of calculating the perimeter of a circle, we can read the input (the radius of the circle) as follow :

```
read(radius);
```

To read the values of multiple variables, we can either read them one by one using separate read statements or read all values simultaneously using a single read statement :

```
1. read(var1);
2. read(var2);
3. read(var3);
4. //or
5. read(var1, var2, var3);
```

Example 2.8:

In the previous problem of calculating the roots of a quadratic equations, we can read the inputs (the equation coefficients) as follow :

```
read(a, b, c);
```

B) Write statement

The write statement allows to send the outputs (variable' values and messages) to the outside via the output device of the computer (mainly the monitor). The message to be shown by the write statement must be delimited by double cotes (quotation marks) (“).

The syntax of the write statement is :

- Show the value of a variable: `write(variable_name);`
- Show the values of multiple variables: `write(var1, var2, var3);`
- Show a message: `write(“the message”);`

- Show a message and the value of a variable: `write ("the message", var1);`

Example 2.9:

In the previous problem of calculating the roots of a quadratic equations, we can show the results (the output) as follow:

In the case of delta < 0 :

```
1. Write("the discriminate delta <0, there are no roots for this equation in R ");
```

In the case of delta = 0 :

```
1. // using multiple write statements
2. write ("there is a single root: x1 = x2 =");
3. write (x1);
4.
5. // Using a single write statement
6. write ("there is a single root: x1 = x2 =", x1);
```

In the case of delta > 0 :

```
1. // using multiple write statements
2. write ("there is two roots: x1 and x2");
3. write ("x1 =", x1);
4. write ("x2 =", x2);
5.
6. // Using a single write statement
7. write ("there is two roots: x1 and x2 : ", "x1 =", x1, "x2 =", x2);
```

2.7 Construction of a simple algorithm

After discussing all the basic aspect of a simple algorithm: algorithm parts, variables, basic operations and basic instructions, now we can construct our first simple algorithm.

In information processing, the most used representation of algorithms is as pseudocode. In this representation, an algorithm consists of two main elements: the header and the body.

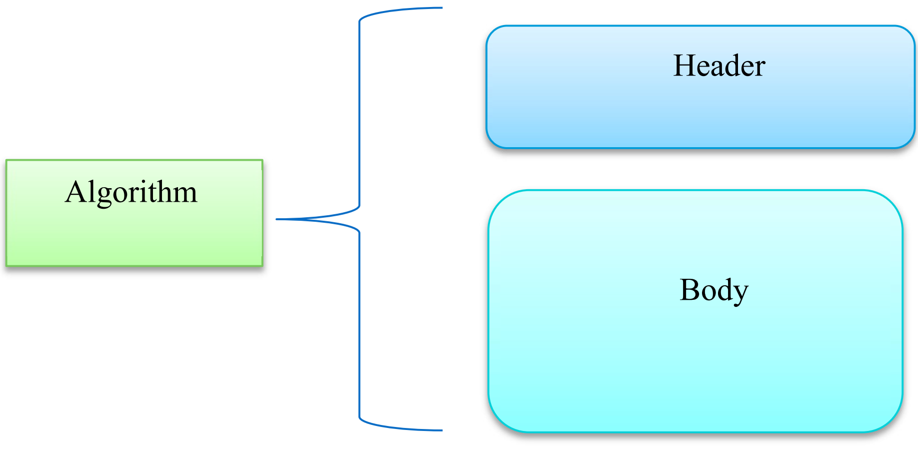


Figure 7: Structure of an Algorithm.

2.7.1 Algorithm header

The algorithm header typically includes a line to specify the algorithm's name and declarations for any constants and variables used within the algorithm. The typical syntax of the algorithm header is :

```
1. Algorithm name_of_the_algorithm
2. Const
3.   // Declare constants here
4. Var
5.   // Declare variables (input, output, and intermediate variables) here
```

The terms **Algorithm**, **Var**, and **Const** are keywords that are required in the header of an algorithm. They indicate the start of the algorithm and delimit the declaration of variables and the declaration of constants, respectively.

Constants declaration instruction has the following syntax:

```
constant_name = constant_value ;
```

Variable declaration instruction has the following syntax :

```
variable_name : variable_type ;
```

Like most of the instructions within an algorithm, declaration instructions always end with a semicolon character (;) .

Example 2.10:

```
1. Algorithm calculate_circle_perimeter
2. Const
3.      PI = 3.14;
4. Var
5.      radius, perimeter: real; // input and output
```

Example 2.11:

```
1. Algorithm calculate_roots_of_quad_equ
2. Const
3.      // in this algo. there are no constants
4. Var
5.      a,b,c,delta,x1,x2: real; // input and output
```

2.7.2 Algorithm body

The body of the algorithm contains all the instructions for acquiring inputs (using read statements), processing data (using arithmetic operation), and generating outputs (using write statement). Its typical syntax is as follow:

```
1. Begin
2.     Instructins1;
3.     Instruction2;
4.     ….
5. End.
```

The terms **Begin**, and **End.** (with the ‘ . ’) are keywords that are required in the body of an algorithm. They delimit the algorithm body.

Example 2.12: Let’s retake the problem of calculating the perimeter of a circle and write the complete algorithm.

```
1. Algorithm calculate_circle_perimeter
2. Const
3.   PI = 3.14;
4. Var
5.   radius, perimeter: real;
6. Begin
7.   // Input
8.   write("Enter the radius of the circle: ");
9.   read(radius);
10.
11.  // Calculation
```

```
12.   perimeter ← 2 * PI * radius;
13.
14.   // Output
15.   write("The perimeter of the circle is: ", perimeter);
16. End.
```

IMPORTANT : it's essential to display a clear message before calling the read statement. This message should inform the user of the expected input, guiding them in providing the correct data.

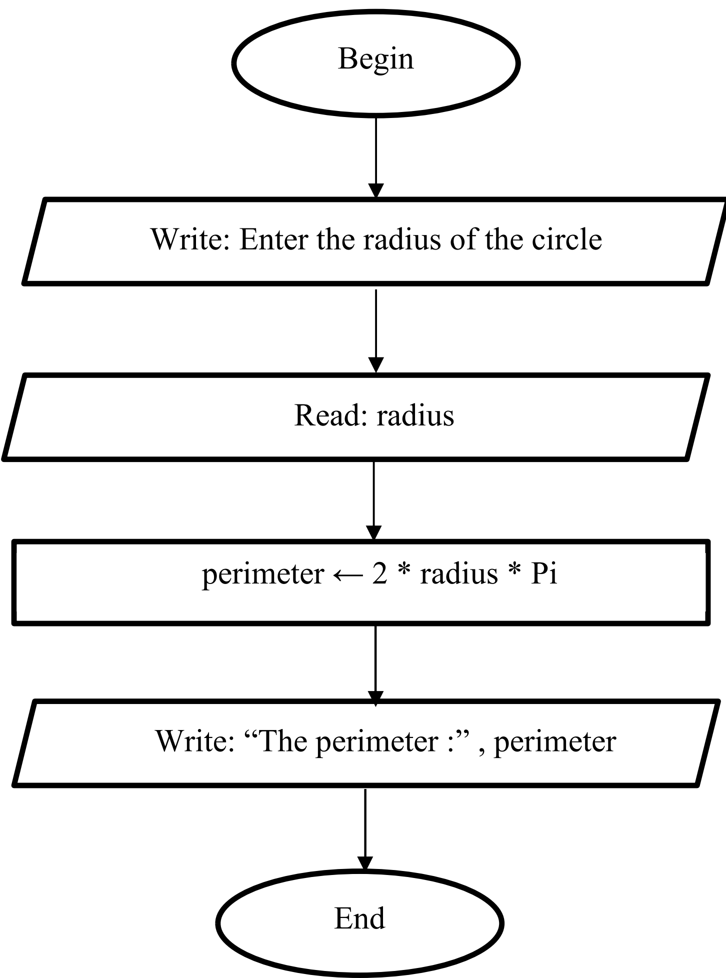
2.8 Graphic representation of an algorithm (organigram)

Algorithms can be represented graphically using Flowcharts (or organigrams). A Flowchart consists of components (shapes) where each shape represents an operation within the algorithms (see Table 1).

Symbol	Name	Purpose
	Oval	Begin , End
	Parallelogram	Input/Output Operation
	Arow	Show relationship between components
	Rectangle	Arithmetic and general processing
	Diamond	Decision (Condition)
	Predefined Process shape	Procedure or Function

Table 1: Building elements of a flowchart.

Example 2.13: Draw a flowchart representing the algorithm to calculate the perimeter of a circle (as discussed previously).



2.9 Translation to C program

Computers cannot directly execute algorithms written in **pseudocode**. Therefore, it is necessary to translate or implement the algorithm into a **source code** written in a programming language that the computer can understand and compile. A source code can be considered a representation of the algorithm using a specific programming language such as : C , C++ , Python, Java, etc.

2.9.1 The C Programming language

C is a general-purpose, high-level programming language developed in the early 1970s. It's known for its efficiency, portability, and flexibility, making it widely used for system programming, operating systems, and embedded systems.

2.9.3 Why C ?

In addition to its efficiency, portability, and flexibility, C is a foundational programming language that has influenced many other languages. Understanding C provides a strong understanding of programming concepts and principles. Mastering C can make it easier to learn more modern languages like C++, Java, and Python, as many of these languages share similarities with C in terms of syntax, control structures, and data types.

2.9.2 Structure of a C Program

A typical C program consists of the following components:

1. **Preprocessor Directives:** These lines, starting with #, instruct the preprocessor to perform tasks before compilation, such as including header files or defining macros.

2. **Global Declarations:** Variables and functions declared outside of any function are considered global and can be accessed from anywhere in the program.

3. **main Function:** The main function is the entry point of a C program. It is where the program's execution begins.

4. **Function Definitions:** Other functions that the program might use can be defined within the program.

5. **Comments:** Comments are used to explain the code and improve readability. They are optional and ignored by the compiler.

Example 2.14 : (a simple C program)

```
1. #include <stdio.h> // Include the standard input/output library
2.
3.
4. int main()
5. {
6.     // variables declarations and statements
7.     int var1;
8.     float var2;
9.     char var3;
10.
11.     printf("This is a simple C program!");
12.
13.     return 0;
14. }
```

2.9.4 Translation from Algorithm to C Program

The following table shows common algorithmic constructs and their corresponding C language equivalents:

<table>
<tr><th>Algorithm element</th><th>C Equivalent</th></tr>
<tr><td>Variables declaration:
<pre>1. Var
2.   Var1, Var2 : integer;
3.   Var3 : real;
4.   Var4 : char;</pre></td><td><pre>1. int Var1, Var2;
2. float Var3;
3. char Var4;</pre></td></tr>
<tr><td>Constants declaration:
<pre>1. Const
2.   Const1 = value;</pre></td><td><pre>1. const type Const1 = value;</pre>In C, we must specify the type of the constant, example :
<pre>1. const float Pi = 3.14;</pre></td></tr>
</table>

<table>
<tr><td>

Assignment:

```
1. Var1 ←(var2 + 3*var3) / var4;
```

</td><td>

```
1. Var1 = (var2 +3*var3) / var4;
```

</td></tr>
<tr><td>

Input statements:

```
1. read(var1);
2. read(var1, var2);
```

</td><td>

```
1. scanf("%d", & var1);
2. scanf("%d %d", & var1,&var2);
```

"%d" indicates that the variable **var1** is of **integer** type. If the variable is of **float** type we use "%f" and if it's of **char** type we use "%c".

Do not forget the **&** character before variable, the following statement will not work :

```
1. scanf("%d", var1);
```

</td></tr>
<tr><td>

Output statements:

```
1. write(var1);
2. write(var1, var2);
3. write("message");
4. write("message", var3, "anothermessage");
5.
```

</td><td>

```
1. printf("%d", var1);
2. printf("%d %d", var1,var2);
3. printf("message");
4. printf("message %d another message", var3);
5.
```

"%d" indicates that the variable **var1** is of **integer** type. If the variable is of **float** type we use "%f" and if it's of **char** type we use "%c".

In printf , we do not use the **&** character before variable name.

</td></tr>
<tr><td>

Comments:

</td><td></td></tr>
</table>

```1. // one line comment 2. /* multiple 3. Lines 4. Comment 5. */ 6.```	```1. // one line comment 2. /* multiple 3. Lines 4. Comment 5. */ 6.```

Table2 : Algorithmic elements and their corresponding C equivalents.

2.10 Exercises

Exercise 2.1:

Suppose that we want to write a simple algorithm that can do the addition, the substruction, the multiplication and the division of two integer numbers.

Analyze this problem to identify:

- inputs/outputs,
- names and types of used variables,
- constants if any,
- processing (arithmetic operations, I/O operations).

Exercise 2.2 :

Apply the same problem-analyzing approach as in Exercise 1 to the following problems:

- Calculation of the age of a person given their birth year,
- Converting an amount from DZ Dinars to US Dollars (exchange rate: 150 DZD = 1 USD),

Exercise 2.3 :

1) Correct the following expression that is supposed to calculate the average of 03 numbers (N1, N2 and N3) : Average ← N1 + N2 + N3 / 3

1) Write the equivalent expressions using parentheses to make the order of operations explicit for the following expressions:

- a * b /c + d
- a / b – c + d/e
- a + b * c – d/e

Examples : a + b * c becomes a+(b*c), because * has higher precedence than +

a/b*c becomes (a/b)*c, because when operators have the same precedence we evaluate the expression from left to right

Exercise 2.4:

Evaluate the following expressions (show your work step by step to demonstrate your understanding of operator precedence):

1) (5 + 2) * (7 - 3) / 4
2) 2^3 + 4 * 5 - 10 / 2
3) (15 - 3) / (2 * 4) + 1
4) ((5 + 2) * 3) - (4 * 2)^2
5) (10 / 2) * (3 + 4) - 5^2

Exercise 2.5:

Consider the following write statements:

1) write(“X”, “Y”);
2) write(X, Y);
3) write(“X=”, X);
4) write(“X=”, X, “Y=”,Y);

What will be the output in each case knowing that the values of X and Y are 7 , 6 respectively ?

Exercise 2.6:

Write the complete algorithms that solve problems seen in exercise 1 and exercise 2.

Exercise 2.7:

Draw the flowchart of the algorithms written in exercise 6.

Exercise 2.8:

Consider the following algorithm:

```
1: Algorithm Algo2
2: Var
3:     X, Y: integer;
4: Begin
5:     write("Enter the first number: ");
6:     read(X);
7:     write("Enter the second number: ");
8:     read(Y);
9:     X← X + Y;
10:    Y← X - Y;
11:    X← X - Y;
12:    write("X=", X);
13:    write("Y=", Y);
14: End.
```

1) Execute (in your mind) the following algorithm and provide the values of the variables X and Y, as well as the output for each line. Assume that the user enters 7 for the first number and 9 for the second number
2) What is the purpose of this algorithm?
3) Modify the algorithm to achieve the same purpose without using arithmetic operations.

Exercise 2.9:

Translate all the seen algorithms to C programs.

Chapter 3

Conditional structures

3.1 Introduction

3.2 Boolean expressions

3.3 Simple conditional structure

3.4 Nested conditional structures

3.5 Multiple choice conditional structure

3.1 Introduction

In simple sequential algorithms (seen in chapter2), instructions are executed one after the other in a sequential order. In some problems, the algorithm may contain instructions that are executed based on a condition, which means that theses instructions are executed only if a condition is verified.

Take the example of calculating the division of two numbers. The correct algorithm that resolves the problem is as follow:

```
1. read number1 and number2
2. if the number2 equals 0 then
3.     write: “division impossible”
4. if the number2 not equal to 0 then
5.     result = number1 / number2
6.     write: “result =”, result
```

As it is clear, instruction 3 (write: “division impossible”), is executed only if number2 is equal to 0 otherwise the algorithm jumps from instruction 2 to instruction 4 (not sequential).

3.2 Boolean expressions

In the conditional execution, the condition based on it the execution is performed or not is represented as a **Boolean expression**. Similar to arithmetic expressions, Boolean expressions can be formed using specific operators (comparison operators and logical operators) :

- **Comparison Operators:** Theses operators are used to compare two values. The result of a comparison is always a Boolean value (true or false). Common operators include: Less than (<), Greater than (>), Less than or equal to (<=), Greater than or equal to (>=), Equal to (=), Not equal to (≠). The values to be compared can be : variables, immediate values or arithmetic expression.

  Example 3.1:

  Var1 > Var2 , Var1 <= 0 , Var1 ≠ (Var2 + 5)

- **Logical Operators:** Theses operators are used to combine Boolean expressions to form complex expressions. Common operators include: NOT (¬ or !) - negates a Boolean value, AND (∧ or &&), OR (∨ or ||).

  Example 3.2 : suppose that we want to represent a condition that verify if a variable **Var1** is in the range [0 , 10]. In this case, the expression is composed of two sub-expressions, the first verify that **Var1** is greater or equal to 0 (Var1 >= 0) and the second verify that **Var1** is less or equal to 10 (Var1 <= 0). So to form the final expression, we simply combine the first sub-expression with the second sub-expression using the AND operator as follow: (Var1 >= 0) ∧ (Var1 <= 10)

The final value of a Boolean expression is Boolean (true or false). When evaluating combined Boolean expression we take in account the following:

A	B	NOT (A)	A ∨ B	A ∧ B
False	False	True	False	False
False	True	True	True	False
True	False	False	True	False
True	True	False	True	True

Example 3.4:

Consider the following Boolean expressions:

```
1. Var1 > Var2
2. Var1 <= (Var2 * 2)
3. (Var1 < 0 ) ∨ (Var2 <0)
```

Calculate the value of each expression when Var1 = 5 and Var2 = -8 .

Solution:

We replace variables by their values:

1) $5 > -8 = true$
2) $5 <= -16 = false$
3) $(5 < 0) \lor (-8 < 0) = false \lor true = true$

3.3 Simple conditional structure

The simple conditional structure, often referred to as the **if-else** statement, allows an algorithm to make a decision (execute one or multiple instructions) based on a specific condition. Its syntax is as follow:

Pseudo code:

```
1. if ( condition ) then
2.     // instructions to be executed if the condition is true
3.     instruction1;
4.     instruction2;
5.     ..;
6. else
7.     // instructions to be executed if the condition is false
8.     Instruction3;
9.     Instruction4;
10.    ..;
11. endif
```

C language :

```
1. if ( condition )
2. {
3.     // instructions to be executed if the condition is true
4.     instruction1;
5.     instruction2;
6.     ..;
7. }
8. else
9. {
10.    // instructions to be executed if the condition is false
11.    Instruction3;
12.    Instruction4;
13.    ..;
14. }
```

Example 3.5:

Pseudocode :

```
1. Algorithm division
2. Var
3.   num1, num2, result: real;
4. Begin
5.   // Input
6.   write("Enter the first number: ");
7.   read(num1);
8.   write("Enter the second number: ");
9.   read(num2);
10.   if (num2 ≠ 0) then
11.      result ← num1 / num2;
12.      write("result :", result);
13.   else
14.      write("Impossible to perform the division");
15.   endif
16. End.
```

C language :

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     float num1, num2, result;
6.     printf("Enter the first number: ");
7.     scanf("%f", &num1);
8.     printf("Enter the second number: ");
9.     scanf("%f", &num2);
10.     if (num2 != 0)
11.     {
12.         result = num1 / num2;
13.         printf("Result: %f", result);
14.     }
15.     else
16.     {
17.         printf("Impossible to perform the division ");
18.     }
19.     return 0;
20. }
```

3.4 Nested conditional structures

Nested conditional structures involve one or more conditional statements within another. This allows for more complex decision-making processes. The syntax of nested conditional structures can be as follow :

Pseudocode:

```
1. if ( condition1 ) then
2.     //instructions to be executed if condition1 is true
3.     if ( condition2 ) then
4.         //instructions to be executed if condition2 is true
5.         //this can include other if-statements
6.     else
7.         //instructions to be executed if condition2 is false
8.
9.     endif
10. else
11.     // instructions to be executed if condition1 is false
12. endif
```

C language:

```
1. if ( condition1 )
2. {
3.     //instructions to be executed if condition1 is true
4.     if ( condition2 )
5.     {
6.          //instructions to be executed if condition2 is true
7.          //this can include others if-statements
8.     }
9.     else
10.     {
11.          //instructions to be executed if condition2 is false
12.     }
13. }
14. else
15. {
16.     // instructions to be executed if condition1 is false
17. }
```

Example 3.6:

Determine if a number is positive, negative, or zero, and if it's positive, determine if it's even or odd.

Pseudocode :

```
1. Algorithm CheckNumber
2. Var
3.     number : integer;
4.
5. Begin
6.     write("Enter a number: ");
7.     read(number);
8.
9.     if ( number > 0) then
10.         if (number % 2 = 0) then
11.             write("The number is positive and even";
```

```
12.         else
13.             write("The number is positive and odd";
14.         endif
15.      else
16.         if ( number < 0 ) then
17.             write("The number is negative")
18.         else
19.             write("The number is zero")
20.         endif
21.      endif
22. End.
```

C language:

```
 1. #include <stdio.h>
 2.
 3. int main()
 4. {
 5.     int number;
 6.     printf("Enter a number: ");
 7.     scanf("%d", &number);
 8.
 9.     if (number > 0)
10.     {
11.         if (number % 2 == 0)
12.         {
13.             printf("The number is positive and even\n");
14.         }
15.         else
16.         {
17.             printf("The number is positive and odd\n");
18.         }
19.     }
20.     else
21.     {
22.         if (number < 0)
23.         {
24.             printf("The number is negative");
25.         }
26.         else
27.         {
28.             printf("The number is zero");
29.         }
30.     }
31.     return 0;
32. }
```

3.5 Multiple choice conditional structure

3.5.1 if-elseif statement

if-elseif statement allows an algorithm to follow different execution paths based on multiple conditions. Its syntax is as follow:

Pseudocode:

```
1. if (condition1) then
2.        inst1;
3.       …
4. elseif (condition2) then
5.        inst3;
6.       …
7. elseif (condition3)
8.       inst4;
9.        …
10. ….
11. ….
12. else
13.       inst1;
14.        …
15.        …
16.
17. endif
18.
```

C language:

```
1. if (condition1)
2. {
3.        inst1;
4.       …
5. }
6. else if (condition2)
7. {
8.        inst3;
9.       …
10. }
11. else if (condition3)
12. {
13.        inst4;
14.        …
15. }
16. ….
17. ….
18. }
19. else
20. {
21.       inst1;
22.        …
23.        …
24. }
```

Example 3.7: A common example is displaying a month's name based on its number.

Pseudocode:

```
1. read(month_num);
2. if ( month_num = 1) then
3.        write (“January”);
4. elseif ( month_num = 2 ) then
5.        write (“February”);
6. elseif ( month_num = 3 ) then
```

```
7.       write (“march”);
8. …
9. …
10. elseif ( month_num = 12 ) then
11.       write (“December”);
12. else
13.      write (“ invalid month number “);
14. endif
```

C language:

```
1. scanf(“%d”, &month_num);
2. if ( month_num == 1)
3. {
4.       printf (“January”);
5. }
6. else if ( month_num == 2 )
7. {
8.       printf (“February”);
9. }
10. else if ( month_num == 3 )
11. {
12.       printf (“march”);
13. …
14. …
15. }
16. else if ( month_num == 12 )
17. {
18.       printf (“December”);
19. }
20. else
21. {
22.      printf (“ invalid month number “);
23. }
```

3.5.2 switch-case statement

When an algorithm need to follow different execution paths based on the value of a variable, a sequence of if-elseif statements can be used. However, Algorithmic provide a more elegant solution which is the **switch-case** statement. Its syntax is as follow :

Pseudocode:

```
1. switch ( variable_name)
2.     case value1:
3.         // code to be executed if variable equals value1
4.         Break;
5.     case value2:
6.         // code to be executed if variable equals value2
7.         Break;
8.     ...
9.     else
10.       //code to be executed if variable doesn't match any case
```

```
11. endswitch
```

C language :

```
1. switch ( variable_name  )
2. {
3.     case value1:
4.         // code to be executed if variable equals value1
5.         break;
6.     case value2:
7.         // code to be executed if variable equals value2
8.         break;
9.     ...
10.    default:
11.      //code to be executed if variable doesn't match any case
12. }
```

Example 3.8: let's retake the example of showing the month name based on its number problem and rewrite the algorithm using the switch-case structure :

Pseudocode :

```
1. Algorithm month_names
2. Var
3. month_num : integer;
4. Begin
5. write ("enter the month number:");
6. read (month_num);
7. switch ( month_num)
8.                case 1:
9.                          write ("january");
10.                         break;
11.               case 2:
12.                         write ("february");
13.                         break;
14.                case 3:
15.                         write ("march");
16.                         break;
17.               case 4:
18.                         write ("april");
19.                         break;
20.               case 5:
21.                         write ("may");
22.                         break;
23.               case 6:
24.                         write ("june");
25.                         break;
26.                case 7:
27.                         write ("july");
28.                         break;
29.               case 8:
30.                         write ("august");
31.                         break;
```

```
32.                 case 9:
33.                           write ("september");
34.                           break;
35.                 case 10:
36.                           write ("october");
37.                           break;
38.                  case 11:
39.                           write ("november");
40.                           break;
41.                 case 12:
42.                           write ("december");
43.                           break;
44.
45.                 else
46.                    write ("please enter a number between 1 and 12");
47.       endswitch
48. End.
```

C language:

```
 1. #include <stdio.h>
 2.
 3. int main()
 4. {
 5. int month_num;
 6. printf ("enter the month number:");
 7. scanf ("%d", &month_num);
 8. switch ( month_num )
 9. {
10.                 case 1:
11.                           printf ("january");
12.                           break;
13.                 case 2:
14.                           printf ("february");
15.                           break;
16.                  case 3:
17.                           printf ("march");
18.                           break;
19.                 case 4:
20.                           printf ("april");
21.                           break;
22.                 case 5:
23.                           printf ("may");
24.                           break;
25.                 case 6:
26.                           printf ("june");
27.                           break;
28.                  case 7:
29.                           printf ("july");
30.                           break;
31.                 case 8:
32.                           printf ("august");
33.                           break;
34.                 case 9:
35.                           printf ("september");
36.                           break;
37.                 case 10:
38.                           printf ("october");
39.                           break;
40.                  case 11:
```

```
41.                         printf ("november");
42.                         break;
43.                 case 12:
44.                         printf ("december");
45.                 break;
46.
47.                 default:
48.                         printf("please enter a number between 1 and 12");
49.     }
50.     return 0;
51. }
```

3.6 Exercises

Exercise 3.1: (Boolean expressions)

Consider the following expressions, where A, B and C are integer variables and their values are 10, 6 and 3 respectively:

1) **A< (B +C)** 2) **A % B** 3) **A >= (B % C)** 4) **NOT (A >= b)** 5) **10 > A > 0** 6) **A <10 AND A>0**
7) A > **NOT (B)** 8) **A OR B OR C** 9) **((A <= 20) AND (B < = 6)) OR (C = 2)**

Identify valid Boolean expressions and calculate their final values.

Exercise 3.2 : (Simple conditional structure, simple condition)

Write an algorithm that asks the user to enter an integer number and shows if the number is positive or negative.

Exercise 3.3 : (Combined condition, Nested conditional structure)

1) Modify the previous algorithm to let it shows also if the number is odd or even when it is positive, using :
   - Simple conditional structure,
   - Nested conditional structures
2) Write an algorithm that takes as input a year and verifies if it's a leap or non-leap year, (To be a leap year, a year must be divisible by 4, but not by 100, unless it's also divisible by 400).

Exercise 3.4: (Multiple choices conditional structures)

1) Write an algorithm that takes as input a month number and shows the name and the number of days of the month, using :
   - **if-else-if**
   - **switch-case**

2) Write an algorithm that resolves a quadratic equation.
3) Write an algorithm that takes as input a birth date as : day, month and year, then it verifies if the date is valid by ensure the following :
    - the year must be in the range [1910-2024],
    - the month should be in the range [1-12],
    - the day must be in one of the following ranges: [1-28], [1-29], [1-30] or [1-31] depending on the entered month and year (leap or non-leap).

Chapter 4

Loops

4.1 Introduction

4.2 While loop,

4.3 Repeat loop,

4.4 For loop,

4.5 Nested loops

4.1 Introduction

In algorithmics, repeating one or more instructions is very common. Take the simple example of showing all the odd numbers less than 100, in this case the repeated instruction are: 1) testing if the number is odd or not, 2) if the number is odd writing it. Writing such code without specific algorithmic structures is impossible.

The algorithmic structure that allow code repeating are loops. There are three types of loops : while loop, repeat loop and for loop.

4.2 While loop

The **while** loop repeats a bloc of instructions while a **condition** is true and stop repeating when the condition became false.

Pseudocode Syntax:

```
1. while (condition) do
2.     // instructions to be repeated
3. endwhile
```

C Language Syntax:

```
1. while (condition)
2. {
3.     // instructions to be repeated
4. }
```

Example 4.1: Printing odd numbers less than or equal a given number.

Pseudocode:

```
 1. Algorithm Print_Odd_Numbers_while
 2. Var
 3.     number, i: integer;
 4.
 5. Begin
 6.     write("Enter a number: ");
 7.     read(number);
 8.
 9.     i ← 0;
10.     while (i <= number) do
11.         if ( i % 2 ≠ 0) then
```

```
12.             write(i);
13.         endif
14.         i ← i + 1;
15.     endwhile
16. End.
```

C Language:

```
 1. #include <stdio.h>
 2.
 3. int main()
 4. {
 5.     int number, i;
 6.
 7.     printf("Enter a number: ");
 8.     scanf("%d", &number);
 9.
10.     i = 0;
11.     while (i <= number)
12.     {
13.         if (i % 2 != 0)
14.         {
15.             printf("%d ", i);
16.         }
17.         i++; // i = i+1;
18.     }
19.
20.     return 0;
21. }
```

4.3 Repeat loop

The repeat loop repeats a bloc of instructions until a condition become true.

Pseudocode Syntax:

```
1. repeat
2.     // instructions to be repeated
3. until ( condition)
```

C Language Syntax:

C doesn't have a direct equivalent to a repeat-until loop. However, we can simulate it using a **do-while** loop. The loop **do-while** in C is similar to the loop **while**, the difference is that **do-while** starts with executing instruction then it checks the condition:

```
1. do {
2.     // code to be executed
3. } while (condition);
```

IMPORTANT: The repeat loop is guaranteed to do at least one iteration, as the condition is checked after the loop body. This differs from the while loop, which may do any iteration if the condition is false from the beginning as it check the condition before the loop body.

Example 4.2: we retake the example of showing odd number.

Pseudocode:

```
1.  Algorithm Print_Odd_Numbers_Repeat
2.  Var
3.      number, i: integer;
4.
5.  Begin
6.      write("Enter a number: ");
7.      read(number);
8.
9.      i ← 0;
10.     repeat
11.         if ( i MOD 2 ≠ 0) then
12.             write(i);
13.         endif
14.         i := i + 1;
15.     until ( i > number)
16. End.
```

C Language:

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.      int number, i;
6.
7.      printf("Enter a number: ");
8.      scanf("%d", &number);
9.
10.     i = 1;
11.     do
12.     {
13.         if (i % 2 != 0) {
14.             printf("%d ", i);
15.         }
16.         i++;
17.     } while (i < number);
18.
19.     return 0;
20. }
```

4.4 For loop

Pseudocode Syntax:

```
1. for i ← initial_Value to final_Value step = step_value do
2.     // code to be executed
3. endfor
```

The variable '**i**' is used as a counter to count the number of iterations (repetitions) that the for loop must repeat the code inside it. The counter begins by the value **initial_value** and ends by the value **final_value**. At the end of each iteration, the counter '**i**' is incremented or decremented by a value equal to **step_value**.

When the step =1, it can be omitted and the syntax become :

```
1. for i ← initial_Value to final_Value do
2.     // code to be executed
3. endfor
```

C Language Syntax:

```
1. for (initialization; condition; increment/decrement)
2. {
3.     // code to be executed
4. }
```

Example 4.3: we retake the example of showing odd number.

Pseudocode:

```
1. Algorithm Print_Odd_Numbers_for1
2. Var
3.     number, i: integer;
4.
5. Begin
6.     write("Enter a number: ");
7.     read(number);
8.
9.     for i←0 to number do
10.        if ( i MOD 2 ≠ 0) then
11.            write(i);
12.        endif
13.    endfor
14. End.
```

Pseudocode, second method:

```
1. Algorithm Print_Odd_Numbers_for2
2. Var
3.     number, i: integer;
4.
5. Begin
6.     write("Enter a number: ");
7.     read(number);
8.
9.     for i←1 to number step =2 do
10.        write(i);
11.    endfor
12. End.
```

In this second method, we do not need a test at all, because the counter i will take only the odd values : 1 , 3 , 5 ,…

C Language:

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     int number, i;
6.
7.     printf("Enter a number: ");
8.     scanf("%d", &number);
9.
10.    for (i = 1; i<= number; i=i+2)
11.    {
12.        printf("%d ", i);
13.    }
14.
15.    return 0;
16. }
```

4.5 Nested loops

Nested loops involve one or more loop within another loop. This is needed to repeat code inside another code that needs to be repeated such as manipulating multi-dimensional arrays.

Pseudocode syntax (for loop):

```
1. for i←initial_value to final_value do
2.       …;
3.      …;
4.       for j←initial_value2 to final_value2 do
```

```
5.            …;
6.           …;
7.           /* this code may contains loops as well*/
8.       endfor
9.       …;
10. endfor
```

C language syntax (for loop):

```
1. for (initialization; condition; incrementation)
2. {
3.       …;
4.       …;
5.       for (initialization2; condition2; incrementation2)
6.       {
7.           …;
8.           …;
9.          /* this code may contains loops as well*/
10.      }
11.      …;
12. }
```

Example 4.4: write an algorithm that display the multiplication table.

Pseudocode :

```
1. Algorithm MultiplicationTable
2. Var
3.     i,j : integer;
4. Begin
5.    for i←1 to 9 do
6.        write ("Multiplication table of ", i);
7.        for j←1 to 9 do
8.            write (i , "x", j , " = " , i*j);
9.        endfor
10.   endfor
11. End.
```

C language :

```
1. int main()
2. {
3.    int i,j;
4.    for (i=1; i<=9; i++)
5.    {
6.        printf ("Multiplication table of %d :\n", i);
7.        for (j =1; j<=9; j++)
8.        {
9.            printf("%d  x %d = %d \n", i , j , i*j);
10.       }
11.   }
12.   return 0;
13. }
```

4.6 Exercises

Exercise 4.1:

Consider the following algorithms:

Algorithm Alg1
Var
i: integer;
Begin
for i←3 **to** 30 step=3 **do**
write (i);
endfor
End.
	Algorithm Alg2
	Var
	i: integer;
	Begin
	i ← 0;
	repeat
	write (i);
	i = i+2;
	until (i >= 50)
	End.

1) What is the purpose of each algorithm?
2) How many iterations does each loop perform?
3) Rewrite the algorithm **Alg1** to display numbers in a descending way using the **while** loop,
4) Rewrite the algorithm **Alg2** using the **while** loop.

Exercise 4.2: (Factorial using while loop)

Write an algorithm that takes in input a positive integer from the user and displays its factorial.

Exercise 4.3: (x power y using repeat-until loop)

Write an algorithm that takes as inputs two positive integer numbers and computes x power y (x^y).

Exercise 4.4: (Prime number check using for loop / use of break inside loops)

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. Write an algorithm that asks the user to enter a number and check if the entered number is prime or not.

Exercise 4.5: (perfect number check)

A perfect number is a positive integer that is equal to the sum of its proper positive divisors, excluding the number itself. For example, the divisors of 28 are 1, 2, 4, 7, 14 and 28. Since 1 + 2 + 4 + 7 + 14 = 28, 28 is a perfect number.

Write an algorithm that takes as input a number and check if it's perfect or not.

Exercise 4.6:

Write an algorithm that takes as input a char (example : '*') and shows the following pattern using nested loops:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

Exercise 4.7:

Write an algorithm that asks the user to enter a positive integer number and computes the sum of its digits.

Exercise 4.8:

Write an algorithm that asks the user to enter a positive integer number N and shows the first N numbers of the Fibonacci series.

Fibonacci series is defined as : $F_n = F_{n-1} + F_{n-2}$ and $F_0 = 0$ and $F_1 = 1$.

Chapter 5

Arrays and strings

5.1 Introduction,

5.2 Array type,

5.3 Multidimensional arrays,

5.4 Strings

5.1 Introduction

Until now, we manipulated data (variables) of single value in different primitive types (integer, real, Boolean and char). When we need to manipulate data of multiple values, for example: marks of tens of students, prices of hundreds of products, etc. the primitive variables cannot be used; because it's impossible to declare and works with tens or hundreds of variables. To deal with such problems, algorithmics provide the concept of **data structures**. A data structure is a variable that can hold multiple values and can be manipulated easily using control structures like loops. There are several data structure that can we use to store and manipulate complex data (Arrays, String, Lists, etc). In this chapter we will study: **Arrays** and **Strings**.

5.2 Arrays

5.2.1 Definition

An array is a data structure that can store multiple elements of the same type. Arrays are characterized by the following:

- Sequential data structure: elements inside arrays are stored in memory one after the other,
- Fixed-size data structure: the number of array elements must be specified in the declaration and remains unchanged during the algorithm execution,
- Unidimensional/multidimensional: elements inside the array can be organized as unidimensional (array of primitive values) or as multidimensional (array of arrays of primitive values) (see section 5.2.2 and 5.2.3).

When declaring an array, we have to specify three essential things: the array **name** (variable name), the array **type** (integer, real, etc) and the array **size** (number of the values that the array can hold).

5.2.2 Unidimensional arrays

A unidimensional array often referred to as a **vector** is an array with primitive (integer, real, char, or Boolean) values as elements. It can be likened to a row of cells; each cell represents an element and has a number (index) to address it.

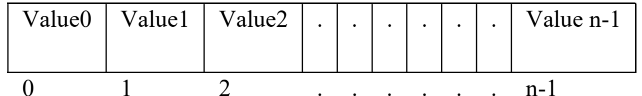


Figure 8: Representation of a vector with a size of n elements; index of cells always star with 0 and ends with n-1.

A) Declaration

Vectors are manipulated as variables; and like other variables they must be declared in the algorithm header. The syntax is as follow:

Pseudocode:

```
name_of_vector : array [0..n-1] of primitive_type ;
```

Example 5.1: declare a vector to store marks of the ADS1 unit knowing that the class contains 150 students.

```
ADS1_Marks : array [0..149] of  real;
```

C language:

```
primitive_type name_of_vector[n] ;
```

Example 5.2: declare a vector to store marks of the ASD1 unit knowing that the class contains 150 students.

```
float   ADS1_Marks[150];
```

B) Access for read/write

To access an array element for read/write operations, we use an index that acts as an address. The index can be :

- An immediate value, example: `ASD1_Marks[5]`, this represents the sixth element of the vector,
- A variable such as a loop counter, example:

```
1. for i←0 to 9 do
2.     write (ADS1_Marks[i]);
3. endfor
```

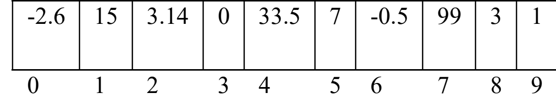
this will write all the first 10 elements of the vector (from `ADS1_Marks[0]` to `ADS1_Marks[9]`)

Example 5.3:

Consider the following algorithm,

```
1.  Algorithm Example
2.  Var
3.        V1 : array[0..9] of real;
4.        i: integer;
5.  Begin
6.        write (V1[5]);
7.        V1[6] ← 9.5;
8.        i ← 3;
9.        write(V1[i]);
10.       V1[i+1] = V1[i+2] + 5;
11.       for (i←2 to 7) do
12.            write(V1[i]);
13.       endfor
14. End.
15.
```

what will be the output if the vector V1 is :



-2.6	15	3.14	0	33.5	7	-0.5	99	3	1
0	1	2	3	4	5	6	7	8	9

Solution :

The output is:

7
0
3.14
0
12
7
-0.5
99

5.2.3 Multi-dimensional arrays

A multidimensional array is an array with arrays as elements (array of arrays). To make things clear let's take the particular case of the two-dimensional array which often

referred to as a **matrix.** A matrix is a vector of vectors (each element in the vector is a vector) and it can be likened to a table which consists of a row of columns.

	0	1	2	.	.	.	.	.	.	n-1
0	Value0,0	Value0,1	Value0,2	.	.	.	.	.	.	Value 0,n-1
1	Value1,0	Value1,1	Value1,2	.	.	.	.	.	.	Value 1,n-1
2	Value2,0	Value2,1	Value2,2	.	.	.	.	.	.	Value 2,n-1
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
m-1	Value m-1,0	Value m-1,1	Value m-1,2	.	.	.	.	.	.	Value 0,n-1

Figure 9: Representation of a matrix with a size of **m** x **n** elements (m rows and n columns).

A) Declaration

A matrix can be declared as follow :

Pseudocode :

```
name_of_matrix : array [0..m-1, 0..n-1] of primitive_type ;
```

Where **m** is the number of rows and **n** is the number of columns

Example 5.4: declare a matrix with 150 rows and 7 columns.

```
my_matrix : array [0..149, 0..6] of  real;
```

C language :

```
primitive_type name_of_matrix[m][n] ;
```

Example 5.5:

```
float   my_matrix[150][7];
```

B) Access for read/write

To access a matrix element for read/write operations, we use two indexes; the first two indicate the row and the second to indicate the column. The indexes can be :

- Immediate values, example: `my_matrix[5,3]`, this represents the element that exists in the row number 5 and the column number 3,
- Variables such as loop counters, example :

```
1. for i←0 to 149 do
2.    for j←0 to 6 do
3.        write (my_matrix[i, j]);
4.    endfor
5. endfor
```

this will write all the elements of the matrix.

Example 5.6:

Consider the following algorithm:

```
 1. Algorithm Example
 2. Var
 3.       mat1 : array[0..3, 0..2] of real;
 4.       i,j: integer;
 5. Begin
 6.       write (mat1[2,1]);
 7.       mat1[0,0] ← 9.5;
 8.       i ← 2;
 9.
10.       for (j←0 to 2) do
11.            write(mat1[i,j]);
12.       endfor
13.
14.       j ← 0;
15.
10.       for (i←0 to 3) do
17.            write(mat1[i,j]);
18.       endfor
19. End.
```

What will be the output if the matrix **mat1** is:

	0	1	2
0	10.5	11	9
1	8	7	6.5
2	13	14	15.5
3	17	16.5	15

Solution:

The output is:

14
13
14
15.5
9.5
8
13
17

5.3 Strings

5.3.1 Definition

A string is a data structure that can hold a sequence of characters delimited by double cotes (" "), often used to represent textual information. It can be considered a special type of array, specifically a vector of characters. However, strings often have additional specific operations to manipulate them that we cannot apply on arrays.

5.3.2 Declaration

To declare a variable as a string we use the following syntax:

Pseudocode :

```
name_of_string : string[size];
```

Size represents the max number of characters that the string can hold. If it's not specified, the default value 255 is used as size for the string.

Example 5.7:

```
1. Var
2. first_name : string[25]; /* first_name is a string, it can hold a max of 25
                               characters*/
3. address : string; /* address is a string, since the size not specified, it can hold a
                      max of 255 characters*/
```

C language :

In the C programing language, a string is declared as a vector of char:

```
char name_of_string[size];
```

Example 5.8:

```
1. char first_name[25];
2. char address[100];
```

5.3.3 Manipulation

Algorithmics and programming languages offers several operations to work with and manipulate strings.

A) I/O operations

Just like variables declared as primitive types, a string can be used with I/O statements (**read** and **write)**.

Example 8.9:

Pseudocode :

```
1. Algorithm example
2. Var
3.        first_name : string[30];
4. Begin
5.        write (“enter your name”);
6.        read (first_name);
7.        write (“Hello ”, first_name);
8. End.
```

C language :

```
1. #include <stdio.h>
2. int main()
3. {
4.        char first_name[30];
5.        printf("enter your name");
6.        scanf("%s", first_name); // with strings, we do not use '&' in scanf
7.        printf ("Hello %s ", first_name); // with strings, we use %s in printf and scanf
8.        return 0;
9. }
```

B) Assignment

We can use the assignment operator ← to assign a value to a variable declared as string as follow :

- Immediate value: `my_string ← "this is a string 123 @# ";` the string value to assigned must be delimited by double cotes " … ",
- another string : `my_string1 ← my_string2 ;`

In the C programing language, the assignment operator (=) can be used only in the declaration to assign an initial value to a string variable. To assign a value to a string already declared, we use a specific function called **strcpy**. The function **strcpy** is defined in the header file <string.h>, so we need to include it when using **strcpy** and other string manipulation functions. Here is an example:

```
1. #include <stdio.h>
2. #include <string.h>
2. int main()
3. {
4.        char first_name[30];
5.        char my_message[10];
5.        printf("enter your name");
6.        scanf("%s", first_name);
7.        strcpy (my_message, "Hello "); /*copy the string "Hello " to the variable
8.                                          my_message*/
9.        printf ("%s %s", my_message, first_name);
10.       return 0;
11. }
```

We can use **strcpy** function to copy a string variable to another string variable :

```
strcpy (destination_string, source_string);
```

C) Concatenation

To concatenate two strings, in pseudocode we can use the + operator and in C language code we can use the **strcat** function. The **strcat** function takes two parameters: source_str and destination_str and concatenates the content of the source_str to the content of the destination_str.

Example 5.10:

Pseudocode:

```
1. Algorithm Example
2. Var
3.       my_string1, my_string2, my_string3 : string [50];
4. Begin
5.       my_string1 = "Hello ";
6.       my_string2 = "World !";
7.       my_string3 = my_string1 + my_string2;
8.       write (my_string3); // this will show : Hello World !
9.       write (my_string1 + my_string2);// we can also do this to show : Hello World !
10. End.
```

C language :

```
1. #include <string.h>
2. int main()
3. {
4.       char my_string1[50], my_string2[50];
5.       strcpy(my_string1, "Hello ");
6.       strcpy(my_string2, "World !");
7.       strcat(my_string1, my_string2);
8.       printf("%s", my_string1); // this will show : Hello World !
9.       return 0;
10. }
```

D) String Length

The size indicated in strings declaration define the max of characters that the string can hold. To know the actual number of characters in a string (the string length), we use the function **strlen** (in pseudocode and C language). The strlen function takes as parameter a string and returns an integer value that represents the number of characters in the string.

Example 5.11:

Pseudocode:

```
1. Algorithm Example
2. Var
3.       my_string1, my_string2, my_string3 : string [50];
4.       len1, len2, len3 : integer;
4. Begin
5.       my_string1 = “Hello ”;
6.       my_string2 = “World !”;
7.       my_string3 = my_string1 + my_string2;
8.       len1 = strlen(my_string1);
9.       len2 = strlen(my_string2);
10.      len3 = strlen(my_string3);
11.      write (len1); // this will show : 6  (space included)
12.      write (len2); // this will show : 7
13.      write (len3); // this will show : 13
14.      write (strlen (my_string1 + my_string2));// this also will show : 13
15. End.
```

C language:

```
1. #include <string.h>
2. int main()
3. {
4.       char my_string1[50], my_string2[50];
5.       int len1, len2, len3;
6.       strcpy(my_string1, “Hello ”);
7.       len1 = strlen(my_string1);
8.       strcpy(my_string2, “World !”);
9.       len2 = strlen(my_string2);
10.      strcat(my_string1, my_string2);
11.      len3 = strlen(my_string1);
12.      printf(“%d %d %d”, len1,  len2, len3); // this will show : 6 7 13
13.      return 0;
14. }
```

E) String comparison

To compare two strings, in pseudocode we can use equality = and inequality ≠ operators. In C language, we use the **strcmp** function. The strcmp function takes as parameter two strings and return an integer value, if the returned value is 0, then the strings are equal otherwise the string are inequal.

Example 5.12:

Pseudocode :

```
1. Algorithm example
2. Var
3.       my_string1, my_string2, my_string3 : string[20];
4. Begin
5.       my_string1 = “abcdefg”;
6.       my_string2 = “abcd”,
7.       my_string3 = my_string2 + “efg”;
8.       if ( my_string1 = my_string2) then    // false
9.                //do some processing;
10.      endif;
11.      if ( my_string1 ≠ my_string2) then    // true
12.               //do some processing;
13.      endif;
14.      if ( my_string1 = my_string3) then    // true
15.               //do some processing;
16.      endif;
17.      if ( my_string1 ≠ my_string3) then    // false
18.               //do some processing;
19.      endif;
20. End.
21.
```

C Language :

```
1. #include <string.h>
2. int main()
3. {
4.       char my_string1[20], my_string2[20];
5.       strcpy(my_string1, “abcdefg”);
6.       strcpy(my_string2, “abcd”);
7.       if ( strcmp(my_string1, my_string2) == 0 ) // false
8.       {
9.                //do some processing
10.      }
11.      if ( strcmp(my_string1, my_string2) != 0 ) // true
12.      {
13.               //do some processing
14.      }
15.      strcat(my_string2, “efg” );
16.      if ( strcmp(my_string1, my_string2) == 0 ) // true
17.      {
18.               //do some processing
19.      }
20.      if ( strcmp(my_string1, my_string2) != 0 ) // false
21.      {
22.               //do some processing
23.      }
24.
25. }
```

E) Manipulation of characters inside a string

A string can be manipulated as a vector of char, consequently, we can access to read/write a character within a string. To do so, it is sufficient to indicate the index of the char as we do with element inside a vector.

Example 5.12:

Pseudocode :

```
1. my_string = "abcdefg";
2. write( my_string[3]); // this will show : d
3. my_string[3] = 'k'; // this will replace d by k, and the string become : abckefg
```

C language:

```
1. strcpy (my_string, "abcdefg");
2. printf("%c", my_string[3]); // this will show : d, we must use "%c" instead of "%s"
3. my_string[3] = 'k'; // this will replace d by k, and the string become : abckefg
```

5.4 Exercises

Exercise 5.1: (Sequential search in a vector)

1)Write an algorithm that takes as input a real value X and searches for it sequentially in a vector V containing 100 unsorted real values. If the value is found, the algorithm should display its index in the vector, otherwise, it should display the message: "Value not found in the vector."

2) Do the same for the case where the vector contains sorted values. In this case, the algorithm should stop searching and show the message "Value not found, Search Stopped." if it encounters a value greater than the searched value.

Exercise 5.2: (Array Rotation)

1) Write an algorithm that read 10 integer values, store them in a vector, rotate the values inside the vector by 1 step to the left and write values of the rotated vector.

2) Modify the algorithm to let it rotate the values by k steps to the right, where k is a positive integer read from input.

Exercise 5.3: (Sequential search in a matrix)

Let's retake the problem in question 1 of the exercise 1 and replace "in a vector V containing 100 unsorted real values" by "in a matrix 50x10 containing unsorted real values".

1) Rewrite the algorithm.

Exercise 5.4: (A practical use of matrixes)

Consider a scenario where we have 30 students, each with marks in 7 units. To store and manipulate this information, we can use 7 vectors, each with 30 elements, where each vector stores the marks for a specific unit.

1) Propose an optimal data structure to efficiently store and manipulate this information than the proposed solution.

2) Write an algorithm to:

- Read and store in the proposed data structure marks of the 7 units for each student,
- Calculate and output the average grade for each student (∑ marks / 7),
- Calculate and output the overall class grade (average of all student' grades).
- Calculate and output the average mark for each unit.

Exercise 5.5: (Strings: Basic operation (I/O, concatenation, string length,…))

1) Write an algorithm that asks the user to enter his name and shows the following message : "Hello [name]" where [name] is the name entered by the user.

2) Modify the algorithm to store the hole message "Hello [name]" in a single variable before showing it,

2) Modify the algorithm to let it show also the number of letters that the user name contains:
"hello [name]"
"your name contains xx letters"

Exercise 5.6: (Strings: Manipulating chars inside strings)

1) Write an algorithm that takes in input a sentence (words separated by spaces) and a letter, then it shows in output the number of occurrences of the letter in the sentence:

Example:

INPUT< Take care of your studies

INPUT< r

OUTPU> 'r' appeared 02 times in the sentence

2) Modify the algorithm to let it counts and shows the number of word in the sentences and then converts lowercase letter at the beginning of each word to uppercase letter.

Example:

INPUT< Take care of your studies

OUTPU> The sentence contains 05 words

OUTPU> The sentence after processing is :

OUTPU> Take Care Of Your Studies

Hints :

Use the following predefined functions:

- chartoascii (char) : takes in input a char and return its ascii code
- asciitochar (integer) : takes in input an integer and return the corresponding char

ASCII codes: lowercase letters [97 -122], uppercase letter [65 - 90]

Exercise 5.7: (binary "dichotomic" search in a sorted vector)

Write an algorithm to implement binary search on a sorted vector. The algorithm must output the index of the target value if found or the message "Value not found!" if the target value is not present in the vector.

Binary search method starts by comparing the target value to the middle element of the vector. If the value at the middle is:

Equal to the target value, the target is found and stop the search,

Less than the middle element, the right half of the vector is eliminated,

Greater than the middle element, the left half of the array is eliminated.

The process of comparing and eliminating continues until the target value is found or the search space is exhausted. If the search ends without finding the target value, it means the target is not present in the array.

Exercise 5.8: (sorting a vector)

Write an algorithm that takes as input 20 real values, store them in a vector V, sort the vector and show its values again. To sort the vector, use the bubble sort method which works as follow:

Compare adjacent elements (the first element with the second element, then the second element with the third element and so on…). In each pair, if the first element is greater than the second, swap them. These steps are repeated n-1 times (n is the number of the elements in the vector).

Example:

Consider the array [5, 2, 8, 1, 9].

Pass 1:

Compare 5 and 2: Swap them. Array becomes: [2, 5, 8, 1, 9]

Compare 5 and 8: No swap.

Compare 8 and 1: Swap them. Array becomes: [2, 5, 1, 8, 9]

Compare 8 and 9: No swap.

Pass 2:

Compare 2 and 5: No swap.

Compare 5 and 1: Swap them. Array becomes: [2, 1, 5, 8, 9]

Compare 5 and 8: No swap.

Compare 8 and 9: No swap.

Pass 3:

Compare 2 and 1: Swap them. Array becomes: [1, 2, 5, 8, 9]

Compare 2 and 5: No swap.

Compare 5 and 8: No swap.

Compare 8 and 9: No swap.

Chapter 6

User-defined types

6.1 Introduction,

6.2 Enumerations,

6.3 Records,

6.1 Introduction

Primitive data types such as Real, Integer, Char, etc. are built-in or predefined data types that the user (the programmer) can use directly in algorithms or programs. User-defined data types, allow to create or define new data types tailored to specific needs where predefined data types cannot adequately represent or handle specific problem domains or complex data structures. This chapter introduces two main custom data types: enumerations and records.

6.2 Enumerations

An enumeration (enum) defines a set of named constants to adequately represent specific data. Enums are useful when dealing with data that have a limited set of values, for example day names, month names, etc.

6.2.1 Enum definition (creation)

Before using the Enum to declare and use variables, we should first define our Enum type.

Enum types and all the other user-defined types are defined in the algorithm header, before the variable declaration section (before **Var**) using the (**Type**) keyword:

```
1. Algorithm Algorithm_Name
2. Type
3.        // user-defined types definitions
4. Var
5.        // variables declaration
6. Begin
7.        //Algorithm body
8. End.
```

When defining a new enum type, we specify the enum name and the constant value (named values) of the enum as follow:

```
1. Type
2.         enum enum_name
3.        {
4.                 constant_value1,
5.                 constant_value2,
6.                 ...
7.        };
8.
```

Example 6.1:

let's defined an Enum type called **day** that has the following constant values : (Friday, Saturday, Sunday, Monday, Tuesday, Wednesday and Thursday)

Pseudocode :

```
 1. Type
 2.         enum day
 3.         {
 4.                  Friday,
 5.                  Saturday,
 6.                  Monday,
 7.                  Tuesday,
 8.                  Wednesday,
 9.                  Thursday
10.         };
```

C language :

```
 1.         enum day
 2.         {
 3.                  Friday,
 4.                  Saturday,
 5.                  Monday,
 6.                  Tuesday,
 7.                  Wednesday,
 8.                  Thursday
 9.         };
10.
```

6.2.2 Enum usage

After defining the Enum new type, we can use it to declare variables and use them in our algorithm as we do with the predefined types (integer, real,..).

Example 6.2:

Pseudocode:

```
1. Algorithm enum_example1
2. Type
3.         enum day
4.         {
5.                 Friday,
6.                 Saturday,
7.                 Monday,
8.                 Tuesday,
9.                 Wednesday,
10.                Thursday
11.         };
12. Var
13.          today, yesterday, tomorrow: day;
14. Begin
15.       today ← Monday;
16.       write(“Today is ”, today); // this will show: Today is Monday
17. End.
18.
```

C Language :

C language cannot read/write enum constant values directly. However, we can still use them with conditional structures (if, switch,..) :

```
1. int main ()
2. {
3.         enum day
4.         {
5.                 Friday,
6.                 Saturday,
7.                 Monday,
8.                 Tuesday,
9.                 Wednesday,
10.                Thursday
11.         };
12.        day today;
13.
14.        today = Monday;
15.        switch (today )
16.        {
17.                case Friday :
18.                {
19.                            printf(“Today is Friday”);
20.                            break;
21.                 }
22.                case Saturday :
23.                {
24.                            printf(“Today is Saturday”);
25.                            break;
26.                 }
27.                case Monday :
28.                {
29.                            printf(“Today is Monday”);
```

```
30.                             break;
31.                 }
32.                 case Tuesday :
33.                 {
34.                             printf(“Today is Tuesday”);
35.                             break;
36.                 }
37.                 case Wednesday :
38.                 {
39.                             printf(“Today is Wednesday”);
40.                             break;
41.                 }
42.                 case Thursday:
43.                 {
44.                             printf(“Today is Thursday”);
45.                             break;
46.                 }
47.                 default :
48.                 {
49.                           printf(“Error! Invalid Values”);
50.                 }
51.
52.       }
53.       return 0;
54.   }
```

6.3 Records

A record is a composite data type that groups data of different types under a single type. It is useful for representing real-world entities with multiple attributes (information).

6.3.1 Record Definition (Creation)

To define a record type, we specify its name and the data types and names of its fields as follow:

Pseudocode:

```
1. Type
2.     record record_name
3.    {
4.         field1: type1;
5.         field2: type2;
6.         ...
7.    };
```

C Language:

```
1. struct record_name
2. {
3.     type1 field1;
```

```
4.     type2 field2;
5.     ...
6. };
```

Example 6.3:

Let's define a record type called **product** that represent a product in a store; the product has the following information (**fields**) : **designation** (string), **quantity** (integer), and **price** (real).

Pseudocode:

```
1. Type
2.     record product
3.     {
4.         designation: string[50];
5.         quantity: integer;
6.         price: real;
7.     };
```

C Language:

```
1. struct product
2. {
3.     char designation [50];
4.     int quantity;
5.     float price;
6. };
```

6.3.2 Record usage

Once a record type is defined, we can declare variables of that type and access its fields using the dor (.) operator.

Example 6.4:

In this example, we declare two variables (prod1 and prod2) of type **product** seen in Example 6.3. then we assign values to the fields of the first variable (prod1), read values from the input for the second variable (prod2) and display on the output the information of the first variable (prod1).

Pseudocode :

```
1. Algorithm example_record
2. Type
3.        record product
4.         {
5.                  designation: string[50];
6.                  quantity : integer;
7.                  price: real;
8.         };
9.  Var
10.         prod1, prod2 : product;
11. Begin
12.          // assigning values to a record
13.          prod1.designation ← “Sugar”;
14.          prod1.quantity← 20;
15.          prod1.price← 90.5;
16.          // read from input
17.          write (“Enter information of the product 2 :”);
18.          write (“Designation :”);
19.           read(prod2.designation);
20.          write (“Quantity :”);
21.          read(prod2.quantity);
22.          write (“Price :”);
23.          read(prod2.price);
24.         // write to output
25.          write (“Information of the product 1:”);
26.          write (“Designation :”);
27.           write(prod1.designation);
28.          write (“Quantity :”);
29.          write(prod1.quantity);
30.          write (“Price :”);
31.          write(prod1.price);
32. End.
33.
```

C Language :

```
1. #include <stdio.h>
2. #include <string.h>
3. // defining the new type (product) outside main function
4. struct product {
5.     char designation[50];
6.     int quantity;
7.     float price;
8. };
9.
10. int main() {
11.     struct product prod1, prod2; // declaring two variables of product type
12.
13.     // Assigning values to a record
14.     strcpy(prod1.designation, "Sugar");
15.     prod1.quantity = 20;
16.     prod1.price = 90.5;
17.
18.     // Read from input
19.     printf("Enter information of the product 2:\n");
20.     printf("Designation: ");
21.     scanf("%s", prod2.designation);
22.     printf("Quantity: ");
23.     scanf("%d", &prod2.quantity);
24.     printf("Price: ");
```

```
25.     scanf("%f", &prod2.price);
26.
27.     // Write to output
28.     printf("\nInformation of the product 1:\n");
29.     printf("Designation: %s\n", prod1.designation);
30.     printf("Quantity: %d\n", prod1.quantity);
31.     printf("Price: %.2f\n", prod1.price);
32.
33.     return 0;
34. }
35.
```

6.4 Exercises

Exercise 6.1:

Using the Enum **day** from the Example 6.2, write an algorithm that display whether a day read from input is a weekday or weekend. Use the following categories:

Weekdays: Sunday to Thursday

Weekend: Friday and Saturday

Exercise 6.2:

1) Define a record type **book** that represent books in a book store. Each book has the following information: title, author, number of page and price.

2) Write an algorithm that:

- Declares a data structure that can store information of 1000 books,
- Reads the details of N book from the user. N is an integer value read from input,
- Finds and prints the title of the most expensive book.

Exercise 6.3:

1) Define a record type **student** that represent students in a school. Each student has the following information: first name, last name and birth date. The birth date must be represented by a record type **date** that consists of three fields : day, month and year.

Write an algorithm that:

- Declares a data structure that can store information of 100 students.

- Reads the details of 20 student from the user,

- Finds and writes the first and the last names of students born before 2005.

References

Books:

Cormen, Thomas H. Introduction à l'algorithmique: cours et exercices. Dunod, 2002. ISBN 9782100039227. Series: Sciences SUP.: Informatique.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms: Fourth Edition*. The MIT Press, 2022. Cambridge, Massachusetts.

Damien Berthet, Vincent Labatut. Algorithmique & programmation en langage C - vol.1 : Supports de cours Volume 1 Période 2005-2014. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.232. cel-01176119v2

Damien Berthet, Vincent Labatut. Algorithmique & programmation en langage C - vol.2 : Sujets de travaux pratiques. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.258. cel-01176120

Damien Berthet, Vincent Labatut. Algorithmique & programmation en langage C - vol.3 : Corrigés de travaux pratiques. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.217. cel-01176121

Damphousse, Pierre. *Petite introduction à l'algorithmique*. Ellipses, 2005.

Zegour, Djamel Eddine. *Apprendre et enseigner l'algorithmique: Tome 1 : Cours et annexes*. Institut National d'Informatique, 2013.

Papers and Chapters:

Di Carlo S. & Prinetto P. Models in Memory Testing, From functional testing to defect-based testing. In: Hans-Joachim Wunderlich (Ed.), Models in Hardware Testing, (pp. 157–185). Springer DEU, 2010.

Garner, R.B., & Dill, R.. The Legendary IBM 1401 Data Processing System. *IEEE Solid-State Circuits Magazine, 2*, 28-39. 2010.

Academic Courses:

BACHIR Malika, Algorithms and Data Structure1, University Batna 1, 2023.

BOUCHERIT Ammar, Algorithmique et Structures de Données 1, Université de Hama Lakhdar, El Oued, 2021.

KARA Messaoud, Algorithmique et Structures de Données 1, Université Mohamed Seddik BENYAHIA, JIJEL, 2021.

Web:

[Web1]:https://en.wikipedia.org/wiki/Antikythera_mechanism#/media/File:DerekdeSollaPrice.jpg (visited in 19-09-2024)

[Web2]:https://en.wikipedia.org/wiki/Difference_engine#/media/File:Babbage_Difference_Engine.jpg (visited in 19-09-2024)

[Web3]:https://en.wikipedia.org/wiki/Difference_engine#/media/File:LondonScienceMuseumsReplicaDifferenceEngine.jpg (visited in 19-09-2024)

[Web4]: https://cs.uwaterloo.ca/~shallit/Papers/cari2.html (visited in 21-09-2024)

[Web5]: https://www.slideshare.net/slideshow/algorithms-and-flowcharts/19018771#1 (visited in 23-09-2024)

[Web6]: https://www.javatpoint.com/c-programming-language-tutorial (visited in 10-11-2024)

Appendix 1: Solutions to Exercises

In this appendix, you will find the solutions to all the exercises provided in the course.

Chapter 2: Simple Sequential Algorithms

Exercise 01:

Problem Analysis:
Inputs:

- num1: integer ; (the first number),
- num2: integer; (the second number).

Outputs:

- sum: integer; (the sum of num1 and num2),
- difference: integer; (the difference between num1 and num2),
- product: integer; (the product of num1 and num2),
- quotient: real; (the quotient of num1 and num2) this must be real .

Constants:

- None

Processing:

1. **Input operations:**
    - Read the values of the inputs (num1 and num2) from the user.
2. **Calculation (arithmetic operations):**
    - sum = num1 + num2 ;
    - difference = num1 - num2;
    - product = num1 * num2 ;
    - quotient = num1 / num2 ;
3. **Output operations:**
    - Display the values of the outputs (sum, difference, product, and quotient).

Exercise 02 :

1) Calculating Age
Inputs:

- birth_year: Integer (the person's birth year)

Outputs:

- age: Integer (the person's age)

Constant :

- current_year = 2024 (to demonstrate the use of constant we used it as a constant, the correct way - especially when translating it to a program- is to use a function that read the system date and extract the current year from it)

Processing:

1. **Input operations:**
    - Read the values of the inputs (birth_year) from the user.
2. **Calculation:**
    - age = current_year - birth_year
3. **Output operations:**
    - Display the values of the outputs (age).

2) Currency Conversion
Inputs:

- amount_dzd: Real (the amount in Algerian Dinars)

Outputs:

- amount_usd: Real (the converted amount in US Dollars)

Constant :

- exchange_rate = 150 (the exchange rate between DZD and USD)

Processing:

1. **Input operations:**
   - Read the values of the inputs (amount_dzd) from the user.
2. **Calculation:**
   - amount_usd = amount_dzd / exchange_rate
3. **Output operations:**
   - Display the values of the outputs (amount_usd).

Exercise 03 :

1) Corrected Expression:

The correct expression to calculate the average of three numbers (N1, N2, and N3) is:

average ← (N1 + N2 + N3) / 3

Explanation:

The parentheses are essential to ensure that the sum of N1, N2, and N3 is calculated before being divided by 3. Without parentheses, the division would be performed first, leading to an incorrect result.

2) Equivalent Expressions with Parentheses:

We follow the order of operations (exponents, multiplication/division, addition/subtraction) to put parentheses.

- a * b / c + d   becomes   ((a * b) / c)+ d , *we cannot put parentheses as : (a * (b / c))+ d , because when operator have the same precedence we evaluate from left to right*
- a / b – c + d/e   becomes   ((a / b) - c) + (d / e)
- a + b * c – d/e   becomes   a + ((b * c) - (d / e))

Exercise 04:

We follow the order of operations (parentheses, exponents, multiplication/division, addition/subtraction) to correctly evaluate these expressions and when operator have the same precedence we evaluate from left to right.

1. **(5 + 2) * (7 - 3) / 4**

   (7 * 4) / 4 = 28 / 4 = 7

2. **2^3 + 4 * 5 - 10 / 2**

   8 + 4 * 5 - 10 / 2 = 8 + 20 - 5 = 23

3. **(15 - 3) / (2 * 4) + 1**

   (12 / 8) + 1 = 1.5 + 1 = 2.5

4. **((5 + 2) * 3) - (4 * 2)^2**

   (7 * 3) - (8)^2 = 21 - 64 = -43

5. **(10 / 2) * (3 + 4) - 5^2**

(5 * 7) - 25 = 35 - 25 = 10

Exercise 05 :

1. **write("X", "Y");**
   - Output: XY
   - This statement shows the strings "X" and "Y" directly, without including the values of the variables.
2. **write(X, Y);**
   - Output: 7 6
   - This statement shows the values of the variables X and Y, separated by a space.
3. **write("X=", X);**
   - Output: X=7
   - This statement shows the string "X=" followed by the value of the variable X.
4. **write("X=", X, "Y=", Y);**
   - Output: X=7 Y=6
   - This statement shows the strings "X=" and "Y=" followed by the respective values of the variables X and Y, separated by spaces.

Explanation:

- When you enclose a value within quotation marks (""), it is treated as a string and showed directly.
- You can combine strings and variables within a write statement to create formatted output.

Exercise 06 :

Write the complete algorithms that solve problems seen in exercise 1 and exercise 2.

Algorithm1: Simple calculations

```
1. Algorithm Simple_Calculations
2. Var
3.    num1,num2,sum,difference,product:integer;
4.    quotient : real;
5. Begin
6.   // Input
7.   write("Enter the first number: ");
8.   read(num1);
9.   write("Enter the second number: ");
10.  read(num2);
11.  // Calculation
12.  sum ← num1 + num2;
13.  difference ← num1 - num2;
14.  product ← num1 * num2;
15.  quotient ← num1 / num2;
16.  // Output
17.  write("Sum:", sum);
18.  write("Difference:", difference);
19.  write("Product:", product);
20.  write("Quotient:", quotient);
21. End.
```

Algorithm 2: Calculating Age

```
1. Algorithm Calculate_Age
2. Const
3.       current_year = 2024;
4. Var
5.       birth_year : integer;
6. Begin
7.     //input
8.   write("Enter your birth year: ");
9.   read(birth_year);
10.  // calculation
11.  age ← current_year - birth_year;
12.  //Output
13.  write("Your age is: ", age);
14. End.
```

Algorithm 3: Currency Conversion

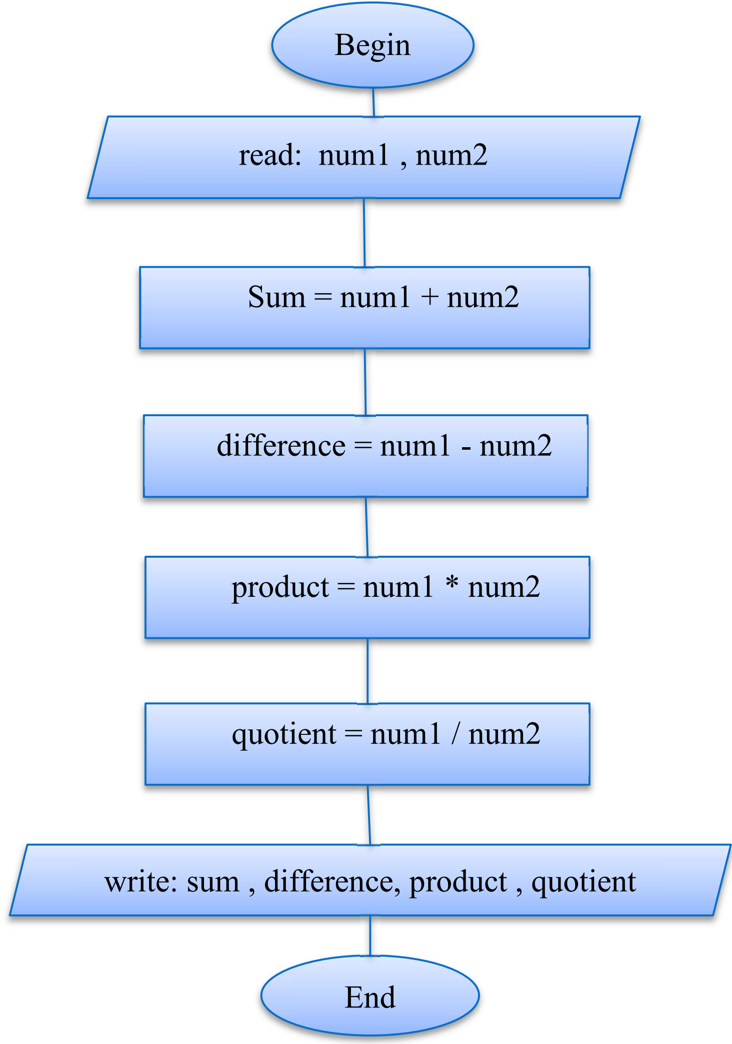
```
1. Algorithm Currency_Conversion
2.
3. Const
4. exchange_rate = 150;
5. Var
6.       amount_usd, amount_dzd: real;
7.
8. Begin
9.   // Input
10.  write("Enter the amount in Algerian Dinars: ");
11.  read(amount_dzd);
12.  // Calculation
13.  amount_usd ← amount_dzd / exchange_rate;
14.  // Output
15.  write("The equivalent amount in US Dollars is: ", amount_usd);
16. End.
```

Notes :

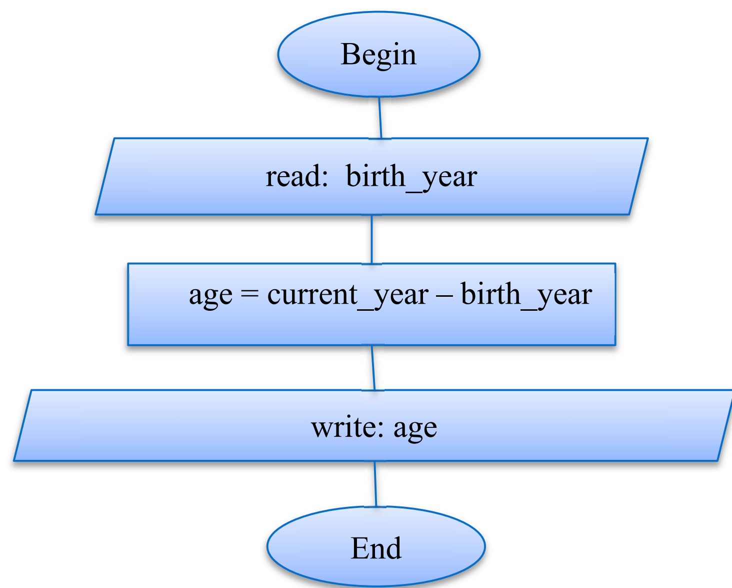
1. Comments (text written between /* */ or after //) are optional but essential to explain any thing that need explanation in your pseudocode,
2. It is crucial to explain to the user what you need as input, which mean that before any **read** statement you have to add a **write** statement to show a message that indicate the required input,
3. It's important to indent the pseudocode (adding indentation) to maintain the clarity of your algorithms and improve its readability.

Exercise 07 :

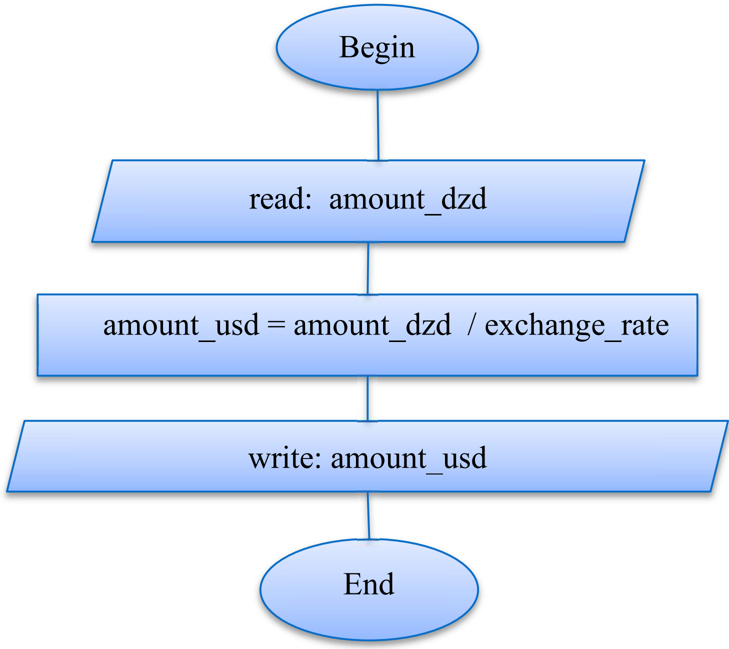
Flowchart 1 : Simple calculations



Flowchart 2 : Calculating age



Flowchart 3: Currency conversion



Exercise 08:

Consider the following algorithm :

```
1: Algorithm Algo2
2: Var
3:     X, Y: integer;
4: Begin
5:     write("Enter the first number: ");
6:     read(X);
7:     write("Enter the second number: ");
8:     read(Y);
9:     X← X + Y;
10:    Y← X - Y;
11:    X← X - Y;
12:    write("X=", X);
13:    write("Y=", Y);
14: End.
```

4) Execute the following algorithm and provide the values of the variables X and Y, as well as the output for each line. Assume that the user enters 7 for the first number and 9 for the second number

Line number	Value of X	Value of Y	Output
1:	/	/	/
2:	/	/	/
3:	/	/	/
4:	/	/	/
5:	/	/	Enter the first number:
6;	7	/	Enter the first number: 7
7:	7	/	Enter the first number: 7 Enter the second number:
8:	7	9	Enter the first number: 7 Enter the second number: 9
9:	7+9 = 16	9	Enter the first number: 7 Enter the second number: 9
10:	16	16-9=7	Enter the first number:

			7 Enter the second number: 9
11:	16-7= 9	7	Enter the first number: 7 Enter the second number: 9
12:	9	7	Enter the first number: 7 Enter the second number: 9 X=9
13:	9	7	Enter the first number: 7 Enter the second number: 9 X=9 Y=7
14:	9	7	Enter the first number: 7 Enter the second number: 9 X=9 Y=7

5) What is the purpose of this algorithm?
   This algorithm swap the values of the input numbers
6) Modify the algorithm to achieve the same purpose without using arithmetic operations.

We can use a temporary variable Z and use it to swap the values of X and Y as follow :

```
1: Algorithm Algo1
2: Var
3:     X, Y, Z: integer;
4: Begin
5:  write("Enter the first number:");
6:  read(X);
7:  write("Enter the second number:");
8:  read(Y);
9:  Z←X;
10: X←Y;
11: Y←Z;
12: write("X=", X);
13: write("Y=", Y);
14:End.
```

Exercise 09:

C Program 1 : Simple calculations

```
1. #include <stdio.h>
2.
3. int main()
4. {
5.     int num1, num2, sum, difference, product;
6.       float quotient;
7.     printf("Enter the first number: ");
8.     scanf("%d", &num1);
9.
10.    printf("Enter the second number: ");
11.    scanf("%d", &num2);
12.
13.    sum = num1 + num2;
14.    difference = num1 - num2;
15.    product = num1 * num2;
16.    quotient = (float)num1 / (float)num2;
17.    printf("Sum: %d", sum);
18.    printf("Difference: %d", difference);
19.    printf("Product: %d", product);
20.    printf("Quotient: %.2f", quotient);
21.
22.    return 0;
23. }
```

C Program 2 : Calculating age

```
1. #include <stdio.h>
2. int main()
3. {
4.     Const float current_year = 2024 ;
```

```
 5. int birth_year, age;
 6.     printf("Enter your birth year: ");
 7.     scanf("%d", &birth_year);
 8.     age = current_year - birth_year;
 9.     printf("Your age is: %d", age);
10.
11.     return 0;
12. }
```

C Program 3 : Currency conversion

```
 1. #include <stdio.h>
 2.
 3. int main()
 4. {
 5.   float amount_dzd, amount_usd;
 6.   const float exchange_rate = 150;
 7.
 8.   printf("Enter the amount in Algerian Dinars: ");
 9.   scanf("%f", &amount_dzd);
10.
11.   amount_usd = amount_dzd / exchange_rate;
12.
13.     printf("The equivalent amount in US Dollars is: %.2f", amount_usd);
14.
15.     return 0;
16. }
```

Chapter 3: Conditional Structures

Exercise 01:

Given:

- A = 10
- B = 6
- C = 3

Evaluating the Expressions:

1. **A < (B + C)**
   - 10 < (6 + 3)
   - 10 < 9
   - **False**
2. **A % B**
   - This is not a boolean expression. It's a modulus operation (arithmetic expression).
   - 10 % 6 = 4
3. **A >= (B % C)**
   - 10 >= (6 % 3)
   - 10 >= 0
   - **True**
4. **NOT (A >= B)**
   - NOT (10 >= 6)
   - NOT (True)
   - **False**
5. **10 > A > 0**
   - This expression is not well-formed. It should be two separate comparisons:
     - 10 > A: 10 > 10 is **False**

- A > 0: 10 > 0 is **True**

6. **A < 10 AND A > 0**
   - (10 < 10) AND (10 > 0)
   - False AND True
   - **False**
7. **A > NOT (B)**
   - This expression is not valid. NOT operator (a logical operator) should be applied to a boolean variable, not a number.
8. **A OR B OR C**
   - This expression is not valid. OR operator (a logical operator) should be applied to boolean variables
9. **((A <= 20) AND (B <= 6)) OR (C = 2)**
   - ((10 <= 20) AND (6 <= 6)) OR (3 = 2)
   - (True AND True) OR False
   - True OR False
   - **True**

Exercise 02 :

Inputs : integer number : num

Outputs : a message to indicate that the number is positive or negative

```
1. Algorithm Positiveornegative
2. Var
3.     Num : integer;
4. Begin
5.     write("Enter a number: ");
6.     read(num);
7.
8.     if (num > = 0) then
9.         write("The number is positive: ");
10.    else
11.        write("The number is negative ");
12.    endif
13.
14. End.
```

Exercise 03 :

1) Indicating if a number is positive or negative and if it is positive if it is odd or even

A) Using nested conditional structures

Inputs : integer number : num

Outputs : a message to indicate if the number is positive or negative and if it is positive if it is odd or even

```
1. Algorithm positiveornegative2
2. Var
3.     num: integer;
4.
5. Begin
6.     write("Enter a number: ");
7.     read(num);
8.
9.     if ( num >= 0) then
```

```
10.         if ( num % 2 = 0) then
11.           write("The number is positive and even");
12.         else
13.           write("The number is positive and odd");
14.         endif
15.     else
16.         write("The number is negative");
17.     endif
18. End.
```

B) Simple conditional structures

Inputs : integer number : num

Outputs : a message to indicate if the number is positive or negative and if it is positive if it is odd or even

```
1. Algorithm positiveornegative2
2. Var
3.     num: integer;
4.
5. Begin
6.     write("Enter a number: ");
7.     read(num);
8.
9.     if ( ( num >= 0) AND ( num % 2 = 0)) then
10.       write("The number is positive and even");
11.    endif
12.    if ( ( num >= 0) AND ( num % 2 ≠ 0)) then
13.       write("The number is positive and odd");
14.    endif
15.    if (  num < 0) then
16.       write("The number is negative");
17.    endif
18. End.
```

2) Leap or non-leap year

```
1. Algorithm Leap_Year_Check
2. Var
3.     year: integer;
4.
5. Begin
6.     write("Enter a year: ");
7.     read(year);
8.
9.     if ( (year % 4 = 0 AND year % 100 ≠ 0) OR (year % 400 = 0) ) then
10.        write(year, " is a leap year");
11.    else
12.        write(year, " is not a leap year");
13.    endif
14. End.
```

Exercise 04:

1) Month name and days :

A) Using if-elseif

```
1.  Algorithm Month_Name_And_Days
2.  Var
3.      monthNumber: integer;
4.
5.  Begin
6.      Write("Enter a month number (1-12): ");
7.      read(monthNumber);
8.
9.      if ( monthNumber = 1) then
10.         Write("January, 31 days")
11.     elseif ( monthNumber = 2 ) then
12.         write("February, 28/29 days")
13.     elseif( monthNumber = 3) then
14.         Write("March, 31 days")
15.     /* (similar for other months) */
16.     else
17.         Write("Invalid month number")
18.     endif
19. End.
```

A) Using switch-cases

```
1.  Algorithm MonthNameAndDaysSwitchCase
2.  Var
3.      monthNumber: integer;
4.
5.  Begin
6.      write("Enter a month number (1-12): ");
7.      read(monthNumber);
8.
9.      switch ( monthNumber)
10.         case 1:
11.             write("January, 31 days");
12.             break;
13.         case 2:
14.             write("February, 28/29 days");
15.             break;
16.         /* ... (similar for other months) */
17.         else
18.             write("Invalid month number");
19.     endswitch
20. End.
```

2) Quadratic equation resolution

```
1.  Algorithm QuadraticEquation
2.  Var
3.      a, b, c, discriminant, root1, root2: Real;
4.
5.  Begin
6.      write("Enter coefficients a, b, c: ");
7.      read(a, b, c);
8.
9.      if ( a = 0) then
10.         Write("Not a quadratic equation");
11.     else
12.         discriminant ← b^2 - 4*a*c;
```

```
13.         switch discriminant
14.           case discriminant > 0:
15.              root1 ← (-b + sqrt(discriminant)) / (2*a);
16.              root2 ← (-b - sqrt(discriminant)) / (2*a);
17.              write("Two roots: ", root1, " and ", root2);
18.              break;
19.           case discriminant = 0:
20.              root1 ← -b / (2*a);
21.              write("One real root: ", root1);
22.           break;
23.           else
24.              write("No real roots");
25.         endswitch
26.     endif
27. End.
```

3) check date

```
1. Algorithm ValidateDate
2. Var
3.     day, month, year: integer;
4.
5. Begin
6.     write("Enter day, month, year: ");
7.     read(day, month, year);
8.
9.     if ( year < 1910 OR year > 2024) then
10.        write("Invalid year");
11.    elseif ( month < 1 OR month > 12 ) then
12.        write("Invalid month");
13.    elseif (month = 4 OR month = 6 OR month = 9 OR month = 11) AND ( day > 30) then
14.        write("Invalid day for this month");
15.    elseif ( month = 2 ) then
16.        if ( (year MOD 4 = 0 AND year MOD 100 ≠ 0) OR (year MOD 400 = 0)) then
17.            if ( day > 29)  then
18.                Write("Invalid day for February in a leap year")
19.            else
20.                Write("Valid date")
21.            endif
22.        else
23.            if ( day > 28 ) then
24.                write("Invalid day for February in a non-leap year");
25.            else
26.                write("Valid date");
27.            endif
28.    elseif ( day < 1 ) then
29.        write("Invalid day");
30.    else
31.        write("Valid date");
32.    endif
33. End.
```

Chapter 4: Loops

Exercise 01:

Consider the following algorithms:

Algorithm Alg1
Var
i: integer;
Begin
for i←3 **to** 30 step=3 **do**
write (i);
endfor
End.
	Algorithm Alg2
	Var
	i: integer;
	Begin
	i ← 0;
	repeat
	write (i);
	i = i+2;
	until (i >= 50)
	End.

1) **the purpose of :**
   a. Alg1 is displaying multiple of 3 numbers small than or equal to 30,
   b. Alg2 is displaying even numbers small than 50 (50 not included),
2) In Alg1 the loop performs **10 iterations** and in Alg2 the loop performs **25 iterations**
3) **Rewrite the algorithm Alg1 to display numbers in a descending way using the while loop**,

   to show numbers from bigger to smaller we change the initial value from 3 to 30, we also change the condition and also decrementing the counter i instead of incrementing it.
4) **Rewrite the algorithm Alg2 using the while.**

   when transforming a **repeat-until** loop to a **while** loop, we have to invert the condition.

Algorithm Alg1
Var
i: integer;
Begin
i←30;
while (i >= 3) **do**
write (i);
i← i - 3;
endwhile
End.
	Algorithm Alg2
	Var
	i: integer;
	Begin
	i←0;
	while (i < 50) **do**
	write (i);
	i← i + 2;
	endwhile
	End.

Exercise 02 : (Factorial using while loop)

```
1. Algorithm factorial
2. Var
3.     n, i, factorial: integer;
4. Begin
5.     Write("Enter a positive integer: ");
6.     read(n);
7.
8.     factorial ← 1;
9.     i ← 1;
```

```
10.     while (i <= n) do
11.         factorial ← factorial * i;
12.         i ← i + 1;
13.     endwhile
14.     write("Factorial of ", n, " is: ", factorial);
15. End.
```

Exercise 03 : (x power y using repeat-until loop)

```
 1. Algorithm Power
 2. Var
 3.     x, y, result, i: integer;
 4. Begin
 5.     write("Enter base and exponent: ");
 6.     read(x, y);
 7.
 8.     result ← 1;
 9.     i ← 1;
10.     Repeat
11.         result ← result * x;
12.         i ← i + 1;
13.     until (i > y) ; /* i >= y is wrong because in this case
14.                     the algorithm will compute x power y-1*/
15.
16.     write(x, " raised to the power of ", y, " is: ", result);
17. End.
```

Exercise 04 : (Prime number check using for loop / use of break inside loops)

```
 1. Algorithm PrimeNumberCheck
 2. Var
 3.     num, i: integer;
 4.     isPrime : Boolean;
 5. Begin
 6.     write("Enter a number: ");
 7.     read(num);
 8.
 9.     isPrime ← true;
10.     for (i ← 2 To num/2) do
11.         if (num % i = 0) Then
12.             isPrime ← false;
13.             break; /* this will stop the loop, because no    needs to continue the 
rest of iterations */
14.         endif
15.     endfor
16.
17.     if (isPrime = 1) then
18.         Write(num, " is a prime number");
19.     else
20.         Write(num, " is not a prime number");
21.     endif
22. End.
```

Exercise 05 : (perfect number check)

```
 1. Algorithm PerfectNumberCheck
 2. Var
```

```
 3.     num, i, sum: Integer
 4.
 5. Begin
 6.     write("Enter a number: ");
 7.     read(num);
 8.     sum ← 0;
 9.     for i ← 1 to num / 2 do
10.         if ( num % i = 0) then
11.             sum ← sum + i;
12.         endif
13.     endfor
14.
15.     if ( sum = num ) then
16.         write(num, " is a perfect number");
17.     else
18.         write(num, " is not a perfect number");
19.     endif
20. End.
```

Exercise 06: (Multiplication tables using nested loops)

```
 1. Algorithm Multiplication_Table
 2. Var
 3.     i,j : integer;
 4. Begin
 5.    for i←1 to 9 do
 6.        write (“Multiplication table of ”, i);
 7.        for j←1 to 9 do
 8.            write (i , “ x ”, j , “ = ” , i*j);
 9.        endfor
10.    endfor
11. End.
```

Exercise 07:

```
 1. Algorithm Pattern_Printing
 2. Var
 3.     i, j: integer;
 4.     char_pat : char;
 5. Begin
 6.      write(“Enter a character”);
 7. read(char_pat);
 8. for i ← 1 to 9 do
 9.         for j ← 1 to i do
10.             write(char_pat);
11.         endfor
12.     endfor
13. End.
```

Exercise 08 :

```
 1. Algorithm SumOfDigits
 2. Var
 3.     num, sum, digit: integer;
 4.
 5. Begin
 6.     write("Enter a positive integer: ");
```

```
7.      read(num);
8.      sum ← 0;
9.      while ( num > 0 ) do
10.         digit ← num % 10;
11.         sum ← sum + digit;
12.         num ← num / 10;
13.     endwhile
14.
15.     write("Sum of digits: ", sum);
16. End.
```

Exercise 09:

```
1. Algorithm FibonacciSequence
2. Var
3.      N, i, first, second, next: integer;
4. Begin
5.      write("Enter the number of terms: ");
6.      read(N);
7.
8.      first ← 0;
9.      second ← 1;
10.
11.     write(first, " ");
12.     write(second, " ");
13.
14.     for i ← 3 to N do
15.         next ← first + second;
16.         write(next, " ");
17.         first ← second;
18.         second ← next;
19.     endfor
20. End.
```

Chapter 5: Arrays and Strings

Exercise 01:

Algorithm 1: Sequential Search in an Unsorted Array

```
1. Algorithm SequentialSearch
2. Var
3.      X :real;
4.      V : array[0..99] of real;
5.      i : integer;
6.      x_found : boolean;
7.
8.  Begin
9.        x_found ← false;
10.       i ← 0;
11.       While (i <= 99 AND x_found = false) Do
12.            if (V[i] = X) Then
13.                 x_found ← true;
14.            else
15.                 i ← i + 1;
16.            endif
```

```
17.        endwhile
18.
19.        if ( x_found = true) then
20.             write("Value found at index ", i);
21.         else
22.              write("Value not found");
23.         endif
24.   End.
```

In the beginning the Boolean variable **x_found** is initialized to false to indicate that the target value hasn't been found yet and the variable **i** is initialized to 0, the index of the first element in the vector,

While the end of the vector is not yet reached (**i<=99**) and **X** not found (**x_found = false**), the algorithm iterates through each element of the array. In each iteration, the current element (**V[i]**) is compared with X. If a match is found (**V[i] = X**), the **x_found** is set to **true**, and the loop breaks. If the end of the array is reached without finding a match **x_found** remains equal to **false**, the loop terminates,

if **x_found** equals **true** this means that X was found and its index is the last value of **i,** otherwise the value is considered not found.

Algorithm 2: Sequential Search in a sorted Array

```
1. Algorithm SequentialSearch
 2. Var
 3.     X :real;
 4.     V : array[0..99] of real;
 5.     i : integer;
 6.     x_found : boolean;
 7.
 8.   Begin
 9.        x_found ← false;
10.        i ← 0;
11.        While (i <= 99 AND x_found = false AND V[i] <= X) Do
12.             if (V[i] = X) Then
13.                  x_found ← true;
14.             else
15.                  i ← i + 1;
16.             endif
17.        endwhile
18.
19.        if ( x_found = true) then
20.             write("Value found at index ", i);
21.         else
22.              write("Value not found");
23.         endif
24.   End.
```

When elements inside the vector are sorted, the search should be stopped (the while loop is breaked) if the current element is greater than X.

Exercise 02 : (Array Rotation)

1) Write an algorithm that read 10 integer values, store them in a vector, rotate the values inside the vector by 1 step to the left and write values of the rotated vector.

```
1. Algorithm Rotate_Left
2. Var
3.    V : array[0..9] of integer;
4.    i, temp : integer;
5. Begin
6.   // Read 10 integer values
7.   write(“enter 10 integer values :”);
8.   for i ← 0 to 9 do
9.       write("Integer number ", i, ": ");
10.      read(V[i]);
11.  endfor
12.  // Rotate the vector by 1 step to the left
13.  temp ← V[1];
14.  for i ← 1 to 9 do
15.     V[i-1] ← V[i];
16.  endfor
17.  V[9] ← temp;
18.  // Write the rotated vector
19.  for i ← 0 to 9 do
20.      write(V[i]);
21.  endfor
22. End.
```

2) Modify the algorithm to let it rotate the values by k steps to the right, where k is a positive integer read from input.

```
1. Algorithm Rotate_Right_K_Steps
2. Var
3.   V : array[0..9] of integer;
4.   i, j, k, temp : integer;
5. Begin
6.   // Read 10 integer values
7.   for i ← 0 to 9 do
8.       write("Integer number ", i, ": ");
9.       read(V[i]);
10.  endfor
11.  // Read the number of steps to rotate
12.  write("Enter the number of steps to rotate: ");
13.  read(k);
14.
15.  // Rotate the vector by k steps to the right
16.  for i ← 1 to k do
17.      temp ← V[9];
18.      For j ← 9 to 1 step=-1 do
19.         V[j] ← V[j-1];
20.      endfor
21.      V[0] ← temp;
22.   endfor
23.
24.  // Write the rotated vector
```

```
25.   for i ← 0 to 9 do
26.       write(V[i]);
27.   endfor
28. End.
```

Exercise 03 : (Sequential search in a matrix)

Let's retake the problem in question 1 of the exercise 1 and replace "in a vector V containing 100 unsorted real values" by "in a matrix 50x10 containing unsorted real values".

```
 1. Algorithm Sequential_Search_In_Matrix
 2. Var
 3.   matrix : array[0..49, 0..9] of real;
 4.   i, j : integer;
 5.   x: real;
 6.   found : boolean;
 7. Begin
 8.   // Read the matrix elements
 9.   for i ← 0 to 49 do
10.       for j ← 0 to 9 do
11.           write("Enter element at row ", i, " column ",     j, ": ");
12.           read(matrix[i, j]);
13.       endfor
14.   endfor
15.
16.   // Read the value to search for
17.   write("Enter the value to search for: ");
18.   read(x);
19.
20.   // Initialize the found flag
21.   found ← false;
22.
23.   // Sequential search
24.   for i ← 0 to 49 do
25.       for j ← 0 to 9 do
26.                 if (matrix[i, j] = x) then
27.                           found ← true;
28.                           write("Value found at index (", i, ", ", j, ")");
29.                           break; /* Exit the loop as soon as the value is found*/
30.                 endif
31.       endfor
32.    endfor
33.
34.    // If the value is not found
35.    if (found = false) then
36.       write("Value not found in the matrix.");
37.    endif
38. End.
39.
```

Exercise 04 : (A practical use of matrixes)

Consider a scenario where we have 30 students, each with marks in 7 units. To store and manipulate this information, we can use 7 vectors, each with 30 elements, where each vector stores the marks for a specific unit.

1) Propose an optimal data structure to efficiently store and manipulate this information than the proposed solution.

Instead of the proposed solution, we use a matrix (array with 2 dimensions) with 30 rows and 7 columns, where each row represents a student, and each column represent a unit.

For example if marks are as follow :

student 1 : 10, 12.5 , 13, 8 , 14.5 , 7 , 10.25

student 2 : 13, 10.5 , 7, 9 , 12.5 , 12 , 13.5

….

student 30: 8, 7.5 , 6, 7 , 10.5 , 9.5 , 12.25

The matrix will be:

indexes	0	1	2	3	4	5	6
0	10	12.5	13	8	14.5	7	10.25
1	13	10.5	7	9	12.5	12	13.5
..	..	..	..	..	..	..	..
49	8	7.5	6	7	10.5	9.5	12.25

2) Write an algorithm to:

- Read and store in the proposed data structure marks of the 7 units for each student,
- Calculate and output the average grade for each student (∑ marks / 7),
- Calculate and output the overall class grade (average of all student' grades).
- Calculate and output the average mark for each unit.

```
1. Algorithm StudentMarks
2.
3. Var
4.   marks : array[0..29, 0..6] of real;
5.   i, j : integer;
6.   student_avg, class_avg, unit_avg : real;
7.
8. Begin
9.   // Read the marks for each student and unit
10.  for i ← 0 to 29 do
11.      for j ← 0 to 6 do
12.        write("Enter mark for student ", i, " in unit ", j, ": ");
13.        read(marks[i, j]);
14.      endfor
15.  endfor
16.
17.  /* Calculate and output the average grade for each student*/
18.  for i ← 0 to 29 do
19.      student_avg ← 0;
```

```
20.       for j ← 0 to 6 do
21.                 student_avg ← student_avg + marks[i, j];
22.       endfor
23.       student_avg ← student_avg / 7;
24.       write("Average grade for student ", i, ": ", student_avg);
25.    endfor
26.
27.   // Calculate and output the overall class average
28.   class_avg ← 0;
29.   for i ← 0 to 29 do
30.       for j ← 0 to 6 do
31.                 class_avg ← class_avg + marks[i, j];
32.       endfor
33.   endfor
34.   class_avg ← class_avg / (30 * 7);
35.   write("Overall class average: ", class_avg);
36.   /* Calculate and output the average mark for each unit*/
37.   for j ← 0 to 6 do
38.       unit_avg ← 0;
39.       for 0 ← 1 to 29 do
40.                 unit_avg ← unit_avg + marks[i, j];
41.       endfor
42.       unit_avg ← unit_avg / 30;
43.       write("Average mark for unit ", j, ": ", unit_avg);
44.    endfor
45. End.
```

Exercise 05: (Strings: Basic operation (I/O, concatenation, string length,…))

1) Write an algorithm that asks the user to enter his name and shows the following message : "Hello [name]" where [name] is the name entered by the user.

```
1. Algorithm hello
2. Var
3.       name : string[30];
4. Begin
5.       write("Enter your name:");
6.       read(name);
7.       write("Hello ", name);
8. End.
```

2) Modify the algorithm to store the hole message "Hello [name]" in a single variable before showing it,

```
1. Algorithm hello2
2. Var
3.       message : string[30];
4. Begin
5.       write("Enter your name:");
6.       read(message);
7.       message ← "Hello " + message;
8.       write(message);
9. End.
```

3) Modify the algorithm to let it show also the number of letters that the user name contains: “hello [name]”

“your name contains xx letters”

```
1. Algorithm hello
2. Var
3.       name : string[30];
4.       str_length : integer;
5. Begin
6.       write(“Enter your name:”);
7.       read(name);
8.       str_length = strlen(name);
9.       write(“Hello ”, name);
10.      write(“your name contains ”, str_length, “letters”);
11. End.
```

Exercise 06: (Strings: Manipulating chars inside strings)

```
1. Algorithm Count_Letter
2.
3. Var
4. sentence: string;
5. letter : character;
6. i, count : integer;
7. Begin
8.   // Read the sentence and letter
9.   write("Enter a sentence: ");
10.  read(sentence);
11.  write("Enter a letter: ");
12.  read(letter);
13.
14.  // Initialize the count
15.  count ← 0;
16.
17.  // Iterate through the sentence
18.  sentence_len ← strlen(sentence) -1;
19.  for i ← 0 to sentence_len do
20.      if (sentence[i] = letter) then
21.               count ← count + 1;
22.      endif
23.  endfor
24.
25.  // Print the result
26.  write("The letter '", letter, "' appeared ", count, " times in the sentence.");
27.
28. End.
```

2) Modify the algorithm to let it counts and shows the number of word in the sentences and then converts lowercase letter at the beginning of each word to uppercase letter.

Example:

INPUT< Take care of your studies

OUTPU> The sentence contains 05 words

OUTPU> The sentence after processing is :

OUTPU> Take Care Of Your Studies

Hints :

Use the following predefined functions:

- chartoascii (char) : takes in input a char and return its ascii code

- asciitochar (integer) : takes in input an integer and return the corresponding char

ASCII codes: lowercase letters [97 -122], uppercase letter [65 - 90]

```
1. Algorithm ProcessSentence
2. Var
3.   sentence : string;
4.   len_sentence, char_code ,i, word_count : integer;
5. Begin
6.   // Read the sentence write("Enter a sentence: "); read(sentence);
7.   // Initialize word count
8.   word_count ← 1;
9.
10.   // Iterate through the sentence
11.   len_sentence ←  strlen(sentence);
12.   for i ← 0 to len_sentence -1 do
13.       // Count words
14.       if (sentence[i] = ' ') then
15.               word_count ← word_count + 1;
16.       endif
17.
18.       // Convert lowercase letters at the beginning of words to uppercase
19.       if (i = 0 OR sentence[i-1] = ' ') then
20.               char_code ← charToAscii(sentence[i]);
21.               if (char_code >= 97 AND char_code <= 122) Then
22.                       char_code ← char_code - 32;
23.                       sentence[i] ← asciiToChar(char_code);
24.               endif
25.       endif
26.   endfor
27.
28.   // Print the word count and the processed sentence
29.   write("The sentence contains ", word_count, " words.");
30.   write("The sentence after processing is:");
31.   write(sentence);
32. End.
```

Exercise 07: (binary “dichotomic” search in a sorted vector)

```
1. Algorithm Binary_Search
2.
3. Var
4.   vector : array[0..99] of integer;
5.   target, mid, left, right : integer;
```

```
6.   found : boolean;
7.
8. Begin
9.   // Read the sorted vector and the target value
10.   // ... (Assume the vector is already read and sorted)
11.   write("Enter the target value: ");
12.   read(target);
13.
14.   // Initialize search boundaries
15.   left ← 0;
16.   right ← 99;
17.   found ← false;
18.
19.   // Binary search loop
20.   while (left <= right) do
21.       mid ← (left + right) div 2;
22.       if (vector[mid] = target) then
23.                found ← true;
24.                write("Target value found at index ", mid);
25.                break;
26.       else if (vector[mid] < target) then
27.                left ← mid + 1;
28.       else
29.                right ← mid - 1;
30.       endif
31.   endwhile
32.
33.   // If the target is not found
34.   if (not found) then
35.       write("Value not found!");
36.   endif
37. End.
38.
```

Exercise 8 :

Input: 20 unsorted real values

Output: 20 sorted real values

Processing:

1- write a message to ask the user to enter 20 unsorted real values,

2- read , from the keyboard, the 20 real values and store them in the vector V,

3- for each pair (V[j], V[j+1]) compare values and swap them if V[j] >V[j+1], where j = [0 -18],

3- repeat the step 3, 18 times to completely sort the vector .

```
1. Algorithm Bubble_Sort
```

```
2.    Var
3.       V : array[0..19] of real;
4.       i,j : integer;// used as loop counter
5.       t : real; // used in the swap operation
6.
7.    Begin
8.       write(" Enter 20 real values");
9.       // reading 20 real values
10.      for i ← 0 to 19 do
11.         read(V[i]);
12.      // sorting the vector
13.       for i ← 0 to 18 do  // 19 passes
14.          for j ← 0 to 18 do // comparing 19 pairs
15.             if V[j] > V[j+1] then
16.                  t ← V[j];
17.                  V[j] ← V[j+1];
18.                  V[j+1] ← t;
19.              endif
20.           endfor
21.        endfor
22.      // showing the vector after sorting it
23.      for i ← 0 to 19 do
24.         write(V[i]);
25.     End.
```

Chapter 6: User-Defined Types

Exercise 6.1:

```
1. Algorithm Day_Category
2. Type
3.     enum day {Friday, Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday};
4. Var
5.     input_day : day;
6. Begin
7.     write("Enter a day : ");
8.     read(input_day);
9.     if (input_day = Friday or input_day = Saturday) then
10.        write("The day is a weekend.");
11.    else
12.        write("The day is a weekday.");
13.    endif;
14. End.
15.
```

Exercise 6.2 :

```
1. Algorithm Book_Store
2. Type
3.     record book
4.     {
5.         title: string[100];
```

```
6.          author: string[50];
7.          num_pages: integer;
8.          price: real;
9.      };
10. Var
11.     books : array[0..999] of book;
12.     N, i, max_index : integer;
13.     max_price : real;
14. Begin
15.     write("Enter the number of books: ");
16.     read(N);
17.
18.     for i ← 0 to N-1 do
19.         write("Enter details for book ", i+1, ":");
20.         write("Title: ");
21.         read(books[i].title);
22.         write("Author: ");
23.         read(books[i].author);
24.         write("Number of pages: ");
25.         read(books[i].num_pages);
26.         write("Price: ");
27.         read(books[i].price);
28.     endfor
29.
30.     max_price ← books[0].price;
31.     max_index ← 0;
32.     for i ← 1 to N-1 do
33.         if (books[i].price > max_price) then
34.             max_price ← books[i].price;
35.             max_index ← i;
36.         endif;
37.     endfor
38.     write("The most expensive book is: ", books[max_index].title);
39. End.
```

Exercise 6.3 :

```
1. Algorithm Student_Records
2. Type
3.      record date
4.      {
5.          day: integer;
6.          month: integer;
7.          year: integer;
8.      };
9.
10.     record student
11.     {
12.         first_name: string[50];
13.         last_name: string[50];
14.         birth_date: date;
15.     };
16.
17. Var
18.     students : array[0..99] of student;
19.     i : integer;
20.
21. Begin
22.     // Read details of 20 students
23.     for i ← 0 to 19 do
```

```
24.         write("Enter details for student ", i+1, ":");
25.         write("First name: ");
26.         read(students[i].first_name);
27.         write("Last name: ");
28.         read(students[i].last_name);
29.         write("Birth date (day): ");
30.         read(students[i].birth_date.day);
31.         write("Birth date (month): ");
32.         read(students[i].birth_date.month);
33.         write("Birth date (year): ");
34.         read(students[i].birth_date.year);
35.     endfor
36.
37.     // Find and write names of students born before 2005
38.     write("Students born before 2005:");
39.     for i ← 0 to 19 do
40.         if ( students[i].birth_date.year < 2005 ) then
41.             write("First name: ", students[i].first_name);
42.             write("Last name: ", students[i].last_name);
43.         endif;
44.     endfor
45. End.
```