People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University Abdelhamid Ibn Badis – Mostaganem
Faculty of Sciences and Technology
Department of Science and Technology

Computer Architecture and Applications

Course Handout with Solved Exercises

1st year– Science and Technology

Expertisé par:
Dr BENIDRIS Fatima-Zohra
Dr ROUBA Baroudi

Dr. BOUATTOU Zina
January 2026

Preface

This document constitutes a teaching support intended for first-year undergraduate students (Science and Technology). It was used in the context of teaching between 2018 and 2026 as a component of the methodological teaching unit. The course handbook brings together a set of theoretical lectures and tutorial sessions (TD), enabling students to acquire a progressive and applied understanding of the concepts studied.

The content has been developed in accordance with the official curriculum defined by the Ministry of Higher Education and Scientific Research. Its aim is to provide students with solid foundations in computer science, which are essential both for their academic training and for their future professional integration.

Course Objectives

The main objective of this module is to enable students to:

- Understand the fundamental concepts of computer science and algorithmics;
- Develop the logic and methodology required for program design;
- Learn programming using a high-level programming language.

Within this framework, the C language has been chosen for its simplicity, efficiency, portability, and its closeness to computer architecture, allowing students to relate program logic to the operations performed by the machine.

For pedagogical reasons, the course is structured into three main chapters:

1. **Introduction to Computer Science:** This chapter provides the necessary foundations for understanding how a computer works and prepares students for the concepts of algorithmics and programming that will be developed in the following chapters.
2. **Notions of Algorithms:** This chapter focuses on the development of logic and program design methods. Students learn how to analyze a problem, design an algorithm, and represent it in a structured manner. They also discover essential concepts such as variables, constants, data types, operators, input/output, and control structures.
3. **Introduction to the C Language:** This chapter aims to teach the translation of algorithmic logic into instructions executable by a computer. Students discover the C language as a practical tool for implementing their algorithms, learning the syntax of a high-level programming language, and developing functional programs. This chapter directly links algorithmic theory with programming practice.

Recommended Prerequisites
It is recommended that students have basic knowledge of web technologies and general familiarity with computer usage. These prerequisites will facilitate the assimilation of the content and the development of practical programming skills.

Keywords:
Computer Science Fundamentals, Computer Architecture, Information Encoding, Number Systems, Algorithm Design, Problem Solving, C Programming, Control Structures, Programming Fundamentals.

Avant-propos

Le présent document constitue un support pédagogique destiné aux étudiants de première année de licence (Sciences et Technologies). Il a été utilisé dans le cadre de l'enseignement entre 2018 et 2026, en tant qu'élément constitutif de l'unité d'enseignement méthodologique. Le polycopié regroupe un ensemble de cours théoriques et de travaux dirigés (TD), permettant à l'étudiant d'acquérir une compréhension progressive et appliquée des notions étudiées.

Le contenu a été élaboré conformément au programme officiel défini par le Ministère de l'Enseignement Supérieur et de la Recherche Scientifique. Il vise à fournir aux étudiants des bases solides en informatique, indispensables à la fois pour leur formation académique et pour leur future insertion professionnelle.

Objectifs du cours

Ce module a pour objectif principal de permettre aux étudiants de :

- Comprendre les concepts fondamentaux de l'informatique et de l'algorithmique ;
- Développer la logique et la méthodologie nécessaires à la conception de programmes ;
- Apprendre à programmer à l'aide d'un langage évolué.

Dans ce cadre, le langage C a été retenu pour sa simplicité, son efficacité, sa portabilité et sa proximité avec l'architecture des ordinateurs, ce qui permet à l'étudiant de relier la logique des programmes aux opérations réalisées par la machine.

Pour des raisons pédagogiques, le cours est structuré en trois chapitres principaux :

1. Introduction à l'informatique: Ce chapitre fournit les bases nécessaires pour comprendre le fonctionnement d'un ordinateur et prépare les étudiants aux notions d'algorithmique et de programmation qui seront développées dans les chapitres suivants.
2. Notions d'algorithmique: Développement de la logique et des méthodes de conception de programmes. Les étudiants apprennent à analyser un problème, concevoir un algorithme et représenter celui-ci de manière structurée. Ils découvrent également les concepts essentiels : variables, constantes, types de données, opérateurs, entrées/sorties et structures de contrôle.
3. Introduction au langage C: Ce chapitre a pour objectif l'apprentissage de la traduction de la logique algorithmique en instructions exécutables par un ordinateur.
   Les étudiants découvrent le langage C comme outil pratique pour mettre en œuvre leurs algorithmes, apprendre la syntaxe d'un langage de programmation évolué et développer des programmes fonctionnels. Ce chapitre relie directement la théorie algorithmique à la pratique de la programmation.

Connaissances préalables recommandées
Il est préférable que les étudiants disposent de notions élémentaires en technologie du Web et d'une familiarité générale avec l'utilisation d'un ordinateur. Ces connaissances faciliteront l'assimilation des contenus et le développement des compétences pratiques en programmation.

Mots-clés:
Fondements de l'informatique, Architecture des ordinateurs, Codage de l'information, Systèmes de numération, Conception d'algorithmes, Résolution de problèmes, Programmation en langage C, Structures de contrôle, Fondamentaux de la programmation.

مقدمة

يُعدّ هذا المستند دعامة بيداغوجية موجّهة لطلبة السنة الأولى ليسانس (علوم وتكنولوجيا). وقد استُخدم في إطار التدريس خلال الفترة الممتدة من 2018 إلى 2026، بوصفه عنصرًا مكوّنًا لوحدة التعليم المنهجي. ويضم هذا المطبوَع مجموعة من الدروس النظرية والأعمال الموجّهة(TD) ، بما يتيح للطالب اكتساب فهم تدريجي وتطبيقي للمفاهيم المدروسة.

تم إعداد المحتوى وفقًا للبرنامج الرسمي الذي حدّدته وزارة التعليم العالي والبحث العلمي، ويهدف إلى تزويد الطلبة بأسس متينة في الإعلام الآلي، وهي أسس ضرورية سواء لمسارهم الأكاديمي أو لاندماجهم المهني مستقبلاً.

أهداف المقياس :

يهدف هذا المقياس أساسًا إلى تمكين الطلبة من :

- فهم المفاهيم الأساسية للإعلام الآلي والخوارزميات؛
- تنمية المنطق والمنهجية اللازمة لتصميم البرامج؛
- تعلّم البرمجة باستخدام لغة برمجة متقدمة.

وفي هذا الإطار، تم اعتماد لغة البرمجة **C** لما تتميز به من بساطة وفعالية وقابلية للنقل، إضافة إلى قربها من بنية الحاسوب، مما يسمح للطالب بربط منطق البرامج بالعمليات التي تنفذها الآلة.

ولأسباب بيداغوجية، قُسّم الدرس إلى ثلاثة فصول رئيسية:

1. **مدخل إلى الإعلام الآلي :**يقدّم هذا الفصل الأسس اللازمة لفهم كيفية عمل الحاسوب، ويُمهّد للطلبة مفاهيم الخوارزميات والبرمجة التي سيتم تطويرها في الفصول اللاحقة.
2. **مفاهيم الخوارزميات :**يركّز هذا الفصل على تطوير المنطق وأساليب تصميم البرامج، حيث يتعلّم الطلبة تحليل المشكلات، وتصميم الخوارزميات، وتمثيلها بطريقة منظّمة. كما يكتشفون المفاهيم الأساسية مثل: المتغيرات، الثوابت، أنواع البيانات، المعاملات، الإدخال/الإخراج، وبُنى التحكم.
3. **مدخل إلى لغة C :** يهدف هذا الفصل إلى تعلّم كيفية ترجمة المنطق الخوارزمي إلى تعليمات قابلة للتنفيذ من طرف الحاسوب.
يتعرّف الطلبة على لغة C كأداة عملية لتطبيق خوارزمياتهم، وتعلّم صياغة لغة برمجة متقدمة، وتطوير برامج وظيفية. ويربط هذا الفصل مباشرة بين النظرية الخوارزمية والتطبيق العملي للبرمجة.

المعارف القبلية الموصى بها:

يُستحسن أن يمتلك الطلبة معارف أولية في تكنولوجيا الويب وإلمامًا عامًا باستخدام الحاسوب. فهذه المعارف من شأنها تسهيل استيعاب المحتويات وتطوير المهارات العملية في مجال البرمجة.

الكلمات المفتاحية:
أساسيات الإعلام الآلي، هندسة الحاسوب، ترميز المعلومات، أنظمة الترقيم، تصميم الخوارزميات، حل المشكلات، البرمجة بلغةC ، هياكل التحكم، أساسيات البرمجة.

Table of Contents

Chapter 1: Introduction to Computer Science

Chapter 2: Notions of Algorithms

Chapter 3: Introduction in C language

Chapter 1: Introduction to Computer Science

1.1. Introduction

This part aims to introduce the basic concepts of computer science. It helps students understand what computer science is, its evolution, and the general functioning of a computer.

It covers information encoding systems, the main hardware components, and the software part, including operating systems, programming languages, and application software.

These concepts provide an essential foundation for the rest of the course, especially for the study of algorithms and programming.

1.2. Definitions

1.2.1. What is a Computer ?

A computer is an electronic device that automatically processes information :

- **Information**: data that has been processed or organized in a meaningful way so that it can be used to make decisions or solve problems. Information can be represented in various formats, such as images, sound, video, or text.
- **Processes**: a set of operations, such as calculations, sorting, or comparisons, that transforms input data into output information.
- **Automatically**: does these operations by itself, without manual intervention for each step.

1.2.2. What is Computer Science?

Computer science is the study and design of algorithms, computation, and information systems, both in theory and in practice, usually with the help of computers, with the goal of significantly saving time and effort.

1.2.3. What is a Computer System?

It refers to all the hardware and software resources required to meet the computing needs of users. A computer system is mainly composed of two parts:

- **Hardware**: This includes the central processing unit, which is the brain of a computer, and input/output devices consisting of a keyboard, mouse, and monitor through which the user interacts with the computer.
- **Software**: This includes the programs and applications that enable the working of computers. Software provides directions to hardware on what to perform, allowing the whole system to work effectively.

Examples of computer systems include: smartphones, smart TVs, personal computers, robots, smartwatches, and gaming consoles.

1.3. Evolution of Computing and Computers

The development of computers has gone through multiple generations, characterised by major technological advancements in each:

- **1st Generation (1945–1955):** Used vacuum tubes, which are an electronic means of amplifying electrical signals. Computing devices of this era were very expensive, huge, energy-consuming, and very unreliable. These computers were rather for large organizations like the government and other big companies.
- **2nd Generation (1956–1964):** Marked by the arrival of transistors, electronic components that replaced vacuum tubes. Thanks to them, computers became smaller, faster, more reliable, and less expensive. This period saw the beginning of true miniaturization.
- **3rd Generation (1964-1971):** The use of integrated circuits, each containing thousands of transistors, characterized this era. Transistors brought about greater miniaturization and an immense increase in computing power.
- **4th Generation (starting from 1972):** Characterized by the utilization of microprocessors, the miniature and power-efficient relatives of conventional processors. This, in turn, ensured an extreme shrinking of the size of computers, increased their performance, and provided them with greater accessibility by the masses.
- **5th Generation (1980s–present):** Associated with the development of artificial intelligence and "intelligent" computer systems capable of learning, reasoning, and processing large amounts of data. This generation also includes the growth of the Internet, mobile devices, and connected objects (IoT), which allow computers and sensors to communicate and share data in real time.

1.4. Information Encoding Systems

The coding of information systems forms the foundation for communication and data processing in computing. This, therefore, offers ways of representing, storing, and transmitting information in formats really understandable by computers. No matter what type of information a computer processes, whether it is images, sounds, text, or videos, it is always represented as numbers in base 2, such as 01010011. A bit (short for binary digit, noted with lowercase b) is the smallest unit of information in a digital system and can have a value of either 0 or 1. Physically, this information can be encoded as an electrical or magnetic signal, with a certain threshold determining whether the value represents 1.

A Byte (uppercase B) consists of 8 bits and can, for example, represent a single character such as a letter or a digit. In computing, storage units increase in powers of 2. The commonly used standardized units are:

- KiloByte (KB): 2^{10} Bytes = 1024 Bytes
- MegaByte (MB): 2^{20} Bytes = 1024 KB
- GigaByte (GB): 2^{30} Bytes = 1024 MB
- TeraByte (TB): 2^{40} Bytes = 1024 GB
- PetaByte (PB): 2^{50} Bytes = 1024 TB
- ExaByte (EB): 2^{60} Bytes = 1024 PB
- ZettaByte (ZB): 2^{70} Bytes = 1024 EB
- YottaByte (YB): 2^{80} Bytes = 1024 ZB

These units allow computers to measure and organize memory and data efficiently.

For a digital piece of information to be processed by a circuit, it must be represented in a form that is compatible with that circuit. This requires choosing a base B numeration system, with B > 1. A number of numeration systems is utilized within digital technology. The four most common are the decimal system (base 10), binary system (base 2), octal system (base 8), and the hexadecimal system (base 16).
Generally, any number in base B can be expressed in a form such as:

$$(N)_B = a_n a_{n-1} \dots a_0 \,.\, a_{-1} a_{-\dots m} \text{ (1)}$$

where each coefficient a_i is a digit with a value between *0* and *B−1*.

Any number *N* can be expressed as the sum of powers of the base, with each power multiplied by a digit (coefficient) corresponding to its position. This decomposition is called the **polynomial form** of the number *N* in a numeration system with base *B*, and it is expressed as:

$$N = a_n * B^n + a_{n-1} * B^{n-1} + \cdots + a_1 * B^1 + a_0 * B^0 + a_{-1} * B^{-1} + \cdots + a_{-m} * B^{-m} \dots (2)$$

Where:

- *N* is the number being represented.
- *B* is the base of the numeration system (e.g., 2 for binary, 10 for decimal, 16 for hexadecimal).
- a_i is the digit (coefficient) at position i, with a value between 0 and B−1.
- B^i is the power of the base corresponding to that position.
- *m* represents the number of digits in the fractional part (the digits after the decimal point).

Examples

- $(345)_{10}$: this number is written in the base-10 system in its polynomial form as:

  $3*10^2 + 4*10^1 + 5*10^0$

- $(1011)_2$: this number is written in the base-2 system in its polynomial form as:

  $1*2^3+0*2^2+1*2^1+1*2^0$

- $(27.54)_8$: this number is written in the base-8 system in its polynomial form as:

  $2*8^1+7*8^0+5*8^{-1}+4*8^{-2}$

1.4.1. Decimal System (base 10)

The decimal system (base 10) is the numeration system most commonly used by humans in everyday life. It uses ten digits, from 0 to 9, and represents numbers as sums of powers of 10.

Example: Decomposition of 2316.123

The number 2316.123 corresponds to: 2316.123 =2000+300+10+6+0.1+0.02+0.003

It can be decomposed according to the powers of 10:

Power	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	…
Digits	2	3	1	6	1	2	3	…

Polynomial Representation: $2316.123 = 2*10^3 + 3*10^2 + 1*10^1 + 6*10^0 + 1*10^{-1} + 2*10^{-2} + 4*10^{-3}$

1.4.2. Binary System (base 2)

It is the fundamental language of computers because all computers and digital systems process information in this form.

The **b**inary dig**it** that can take these two states (0 or 1) is called a **bit**, and any binary combination consisting of eight bits is called a **Byte**.

Example: Decomposition of $(10011100)_2$

It can be decomposed according to the powers of 2:

Power	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Digits	1	0	0	1	1	1	0	0

Polynomial Representation: $(10011100)_2 = 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0$

1.4.3. Octal System (base 8)

The binary system is fundamental for computers, but it is long and difficult for humans to read (a sequence of 0 and 1). To simplify reading, we use systems whose base is a power of 2, which makes conversion to binary easier. Some digital calculators use the octal system directly to simplify the reading of binary numbers grouped in 3 bits (since $2^3=8$).

The octal system is a base-8 numeral system. It uses 8 symbols: 0, 1, 2, 3, 4, 5, 6, 7.

Example: Decomposition of $(237)_8$

It can be decomposed according to the powers of 8:

Power	8^2	8^1	8^0
Digits	2	3	7

Polynomial Representation: $(237)_8=2*8^2+3*8^1+7*8^0$

1.4.4. Hexadecimal System (base 16)

It is a numeral system that uses base 16, originating from the development of microcomputers. Some computing systems use the hexadecimal system directly to simplify the reading of binary numbers grouped in 4 bits (since $2^4 = 16$). It relies on 16 different symbols: the ten digits from 0 to 9, followed by the letters A (for 10), B (for 11), C (for 12), D (for 13), E (for 14), and F (for 15).

Example: Decomposition of $(3AE7)_{16}$

It can be decomposed according to the powers of 16:

Power	16^3	16^2	16^1	16^0
Digits	3	A	E	7

Polynomial Representation: $(3AE7)_{16}=3*16^3+A*16^2+E*16^1+7*16^0$

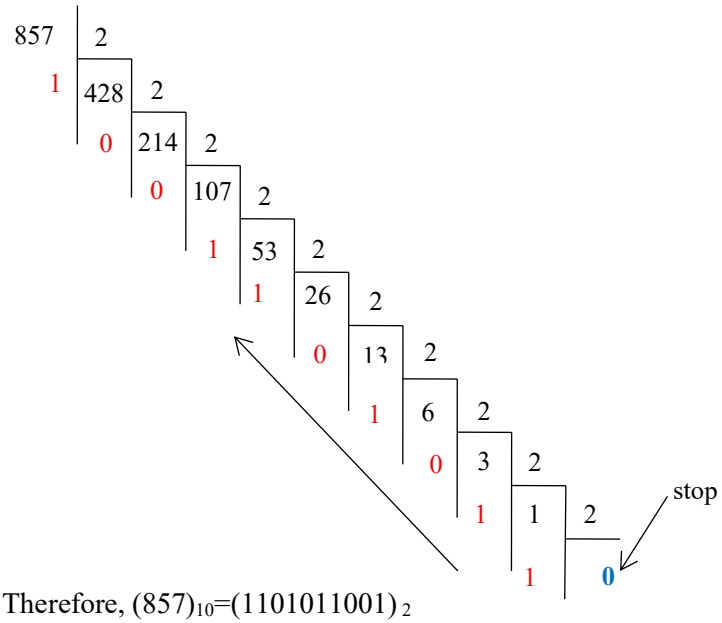
1.5.Conversions

1.5.1. Decimal / Base-B

1.5.1.1.Conversion of the Integer Number

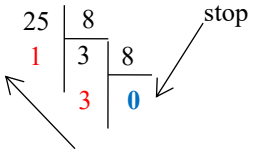
Method: To convert an integer number from base 10 to any base B (B>1), perform successive divisions by B, keeping the remainder at each step, until the quotient is 0. The final number is then written from left to right, starting with the last remainder obtained and working up to the first remainder.

Example 1: Convert 857 in Base-2



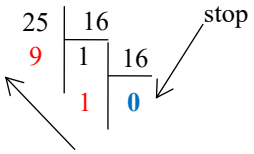
Therefore, $(857)_{10}=(1101011001)_2$

Example 2: Convert 25 in Base-8



Therefore, $(25)_{10}=(31)_8$

Example 3: Convert 25 in Base-16



Therefore, $(25)_{10}=(19)_{16}$

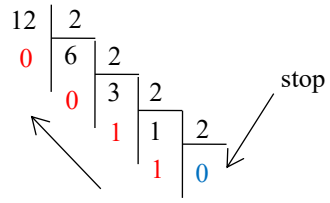
1.5.1.2.Conversion of Number with Fractional Part

Method:

- Integer part: Convert the integer part as a whole number using successive divisions by the base B.
- Fractional part: Multiply the fractional part successively by the base B (B>1), keeping the integer part obtained at each step.
- Continue this process until the fractional part becomes zero or repeats. If the fractional part does not terminate, the representation is infinite; in this case, the process may be stopped after a finite number of digits (rounding issue).
- The result of the conversion is given by the sequence of integer parts obtained, written in order of appearance after the radix point and expressed in base B.

Example 1: Convert $(12.625)_{10}$ to binary (B=2)

Step 1: Integer part (12)



Binary integer part: $\mathbf{(1100)_2}$

Step 2: Fractional part (0.625)

0.625×2= 1.25→ integer part 1
0.25×2 = 0.5 → integer part 0
0.5×2 = 1.0 → integer part 1

Binary fractional part: **$(0.101)_2$**

The result is therefore: $(12.625)_{10}=(1100.101)_2$

Example 2: Convert $(0.1)_{10}$ to octal (B=8)

Step 1: Integer part: The integer part is 0, so there is nothing to convert.

Step 2: Fractional part (0.1): Successively multiply by 8 and keep the integer part:

$0.1\times8=0.8\rightarrow$ integer part 0
$0.8\times8=6.4\rightarrow$ integer part 6
$0.4\times8=3.2\rightarrow$ integer part 3
$0.2\times8=1.6\rightarrow$ integer part 1
$0.6\times8=4.8\rightarrow$ integer part 4
$0.8\times8=6.4\rightarrow$ integer part 6

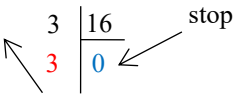
We notice that the sequence repeats: 0,6,3,1,4,6…

The result is therefore: $(0.1)_{10} \approx (0.0\underline{\mathbf{6314}}\ \underline{\mathbf{6314}}\ \ldots)_8$

Note: The decimal number $(0.1)_{10}$ does not have a finite representation in base 8 and is therefore expressed as a repeating fractional number. Its representation in base 8 is periodic, with the repeating sequence **6314**.

Example 3: Convert $(3.718)_{10}$ to Hexadecimal (B=16)

Step 1: The integer part is 0, so there is nothing to convert here.



Integer part in hexadecimal: $(3)_{16}$

Step 2: Multiply the fractional part by 16 repeatedly:

$0.718\times16=11.488 \rightarrow$ integer part = B (11 in hex)
$0.488\times16=7.8080.488\times16=7.808 \rightarrow$ integer part = **7**
$0.808\times16=12.9280.808\times16=12.928 \rightarrow$ integer part = C (12 in hex)
$0.928\times16=14.8480.928\times16=14.848 \rightarrow$ integer part = E (14 in hex)
$0.848\times16=13.5680.848\times16=13.568 \rightarrow$ integer part = D (13 in hex)
$0.568\times16=9.0880.568\times16=9.088 \rightarrow$ integer part = 9

We can stop here for an approximation.

The result is therefore: $3.718 \approx (3.B7CED9\ldots)_{16}$

Note: The decimal number $(3.718)_{10}$ cannot be represented exactly in base 16. As a rational number, its representation in base 16 either terminates or repeats periodically. When the representation is infinite, the conversion process is stopped after a finite number of digits in order to obtain an approximation.

1.5.2. Base-B / Decimal

To convert a number from base B (B>1) to decimal, use its polynomial form. Simply expand the number as a sum of each digit multiplied by the base raised to the power corresponding to its position:

$$N = a_n * B^n + a_{n-1} * B^{n-1} + \cdots + a_1 * B^1 + a_0 * B^0 + a_{-1} * B^{-1} + \cdots + a_{-m} * B^{-m} \ldots (2)$$

Examples:

Convert the following binary numbers to decimal: $(11110)_2$, $(43.07)_8$, $(2B)_{16}$.

$(11110)_2 = 1*2^4+1*2^3+1*2^2+1*2^1+0*2^0 = 16+8+4+2+0=30$

Therefore, $(11110)_2=(30)_{10}$.

$(43.07)_8=4*8^1+3*8^0+0*8^{-1}+7\cdot 8^{-2}=4*8+3*1=32+3+0+0.109375=(35.109375)_{10}$.

Therefore, $(43.07)_8=(35.10937510)_{10}$.

$(2B)_{16}=2*16^1+11*16^0=32+11=43$.

Therefore, $(2B)_{16}=(43)_{10}$.

1.5.3. Conversion from base-B1 to base-B2

1.5.3.1.Indirect Method

To convert a number from an arbitrary base B1 (B1>1) to another base B2 (B2>1) , the standard approach is to use base 10 as an intermediate step:

- First, convert the number from base B1 to base 10.
- Then, convert the resulting value from base 10 to base B2.

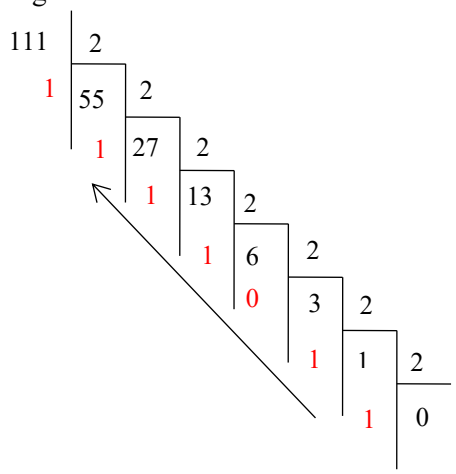
This *indirect method* is universal and always works, regardless of the relationship between B1 and B2. However, it can be time-consuming because it requires performing two full conversions instead of one.

Example: Convert $(157)_8$ to Binary using the Indirect Method

Step 1 : Octal → Decimal

$(157)_8 = 1*8^2+5*8^1+7*8^0=64+40+7= (111)_{10}$

Step 2 : Decimal → Binary



$(111)_{10} = (1101111)_2$

Therefore, $(157)_8=(1101111)_2$.

1.5.3.2. Direct method (Table)

A direct conversion is possible when the two bases are related by a power, that is:

$$B1=B2^k$$

for some integer k>0

In this case, the conversion can be performed using a common intermediate base. This situation commonly occurs with number systems derived from the binary system.

For example:

- **Tetradecimal (base 4):** $4=2^2$: Each base-4 digit maps directly to **2 bits**.
- **Octal (base 8):** $8=2^3$: Each base-8 digit maps directly to **3 bits**.
- **Hexadecimal (base 16):** $16=2^4$: Each base-16 digit maps directly to **4 bits**.

This property allows a direct table-based conversion without passing through base 10.

Example 1 : binary to octal

- Convert $(1111010011001111)_2$ to the Octal (base 8) using the Direct Method

Because: $8=2^3$, each octal digit corresponds to 3 binary bits.

Step 1: Group the binary number into groups of 3 bits, start from the right :

Binary number:

001 111 010 011 001 111

Step 2 : Convert each 3-bit group to octal using the Table 1 :

001→ 1
111→ 7
010→ 2
011→ 3
001→ 1
111→ 7

Therefore, $(1111010011001111)_2=(172317)_8$

Table 1

Octal System (Base 8)	**Binary System** (Base 2)		
$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Example 2: Octal to binary

- Convert $(7365)_8$ to Binary (base 2) using the Direct Method

Because: $8=2^3$, each octal digit corresponds exactly to 3 binary bits.

Step 1: Convert each octal digit to 3 bits using Table 1

7→ 111
3→ 011
6→110
5→101

Step 2 : Join the binary groups together :

$(7365)_8 = (111\ 011\ 110\ 101)_2$

Therefore, $(7365)_8=(111011110101)_2$

Example 3: binary to Hexadecimal

- Convert $(101010111100.11011)_2$ to Hexadecimal (base 16) using the Direct Method

Because: $16=2^4$, each hexadecimal digit corresponds to 4 binary bits.

Step 1: Group the binary number into blocks of 4 bits, from right to left for the integer part, and from left to right for the fractional part.

(1010 1011 1100. 1101 1000)

Step 2: Convert each 4-bit group to octal using the Table 2 :

1010 → A
1011 → B
1100 → C
1101 → D
1000 → 8

Therefore, $(1101010111100.11011)_2=(ABC.D8)_{16}$

Example 4 : Hexadecimal to binary

- Convert $(3F.A)_{16}$ to Binary (base 2) using the Direct Method

Table 2

hexadecimal System (Base 16)	**Binary System** (Base 2)			
$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1
E	1	1	1	0
F	1	1	1	1

Because: $16=2^4$, each hexadecimal digit corresponds exactly to 4 binary bits.

Step 1: Convert each Hexadecimal digit to 4 bits using table:

3 → 0011
F → 1111
A →1010

Step 2: Join the binary groups together : 0011 1111.1010

Therefore, $(3F.A)_{16} = (00111111.1010)_2$

Example 5: Octal to Hexadecimal

- Convert $(7453.62)_8$ to Hexadecimal (base 16) using the Direct Method

Step 1: Convert Octal to Binary

Each octal digit → 3 binary bits (Table 1)

7→ 111
4→ 100
5→ 101
3→ 011
6→ 110
2→ 010

$(7453.62)_8 = (111100101011.110010)_2$

Step 2 — Convert Binary to Hexadecimal

Because $16=2^4$, group binary digits in blocks of 4 from right to left:

$(1111\ 0010\ 1011.1100\ 1000)_2$

Now convert each 4-bit group to Hexadecimal (Table 2)

1111 → F
0010 → 2
1011→ B
1100 → C
1000 → 8

Therefore, $(7453.62)_8 = (F2B.C8)_{16}$

Example 6: Hexadecimal to octal

- Convert $(90BD5A)_{16}$ to Octal (base 8) using the Direct Method

Step 1: Convert Hexadecimal to Binary

Each hex digit → 4 binary bits (Table 2):

9 → 1001
0 → 0000
B → 1011
D → 1101
5 → 0101
A → 1010

$(90BD5A)_{16}=(1001\ 0000\ 1011\ 1101\ 0101\ 1010)_2$

Step 2 : Convert Binary to Octal

Because $8=2^3$, group binary digits in blocks of 3 from right to left**:**

$(1001\ 0000\ 1011\ 1101\ 0101\ 1010)_2$

Now convert each 3-bit group to octal:

100 → 4
100 → 4
001 → 1
011 → 3
110 → 6
101 → 5
011 → 3
010 → 2

Therefore, $(90BD5A)_{16}=(44136532)_8$

1.6. Arithmetic Operations in Base B

Arithmetic operations are performed in any base B following the same principles as in base 10.

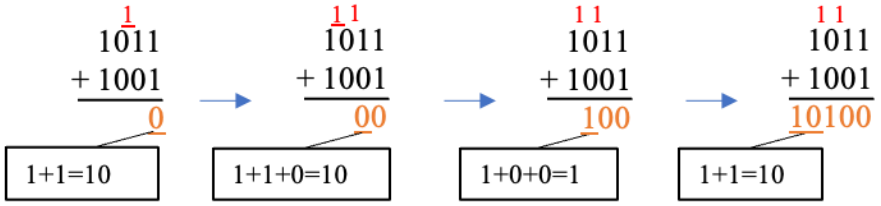
- In addition and multiplication, a carry occurs when the result is equal to or greater than the base B.
- In subtraction and division, a borrow occurs when the digit being processed is smaller than the digit to be subtracted or divided.

- In multiplication, digits are multiplied while respecting the base B; intermediate results are shifted and added as in base 10.
- In division, successive divisions are performed in base B, producing at each step a partial quotient and a remainder, following the same rules as decimal division.

1.6.1. Addition

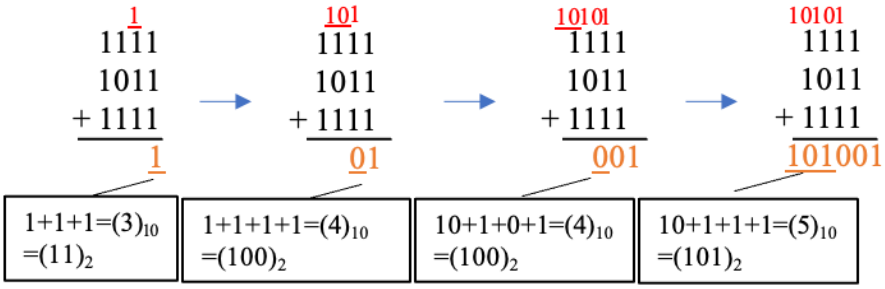
It is enough to know that:
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 with a **carry** of 1

Example 1: Compute $(1011)_2+(1001)_2$ using binary addition:



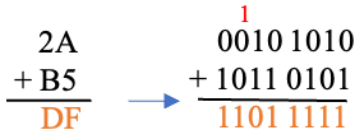
Therefore, $(1011)_2+(1001)_2=(10100)_2$

Example 2: Compute $(1111)_2+(1011)_2+(1111)_2$ using binary addition:



Therefore, $(1111)_2+(1011)_2+(1111)_2=(101001)_2$

Example 3: Compute $(2A)_{16}+(B5)_{16}$

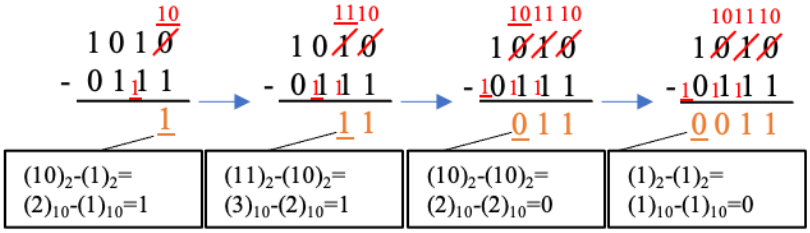


Therefore, $(2A)_{16}+(B5)_{16}= (DF)_{16}$

1.6.2. Subtraction

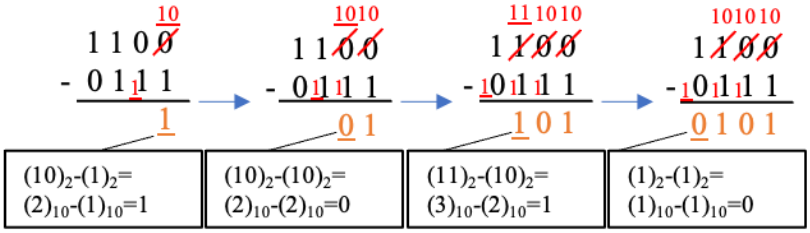
It is enough to know that:
0 − 0 = 0
0 − 1 = 1 with a **borrow** of 1
1 − 0 = 1
1 − 1 = 0

Example 1: Compute $(1010)_2$-$(0111)_2$:



Therefore, $(1010)_2$-$(0111)_2$= $(0011)_2$

Example 2: Compute $(1100)_2$-$(111)_2$:



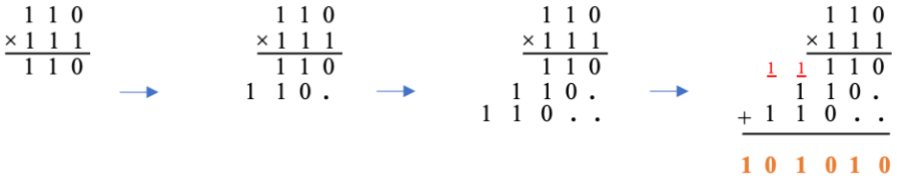
Therefore, $(1100)_2$-$(0111)_2$= $(0101)_2$

1.6.3. Multiplication

Binary multiplication is very simple. Here is the multiplication table:

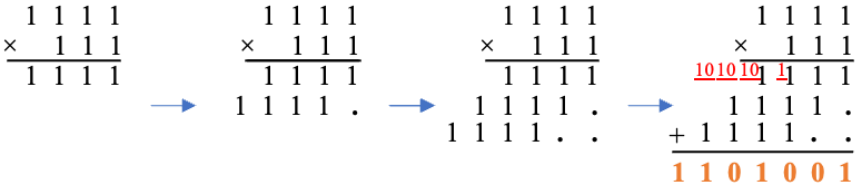
0 × 0 = 0
0 × 1 = 0
1 × 0 = 0
1 × 1 = 1

Example 1: Compute $(110)_2 \times (111)_2$:



Therefore, $(110)_2 \times (111)_2 = (101010)_2$

Example 2: Compute $(1111)_2 \times (111)_2$



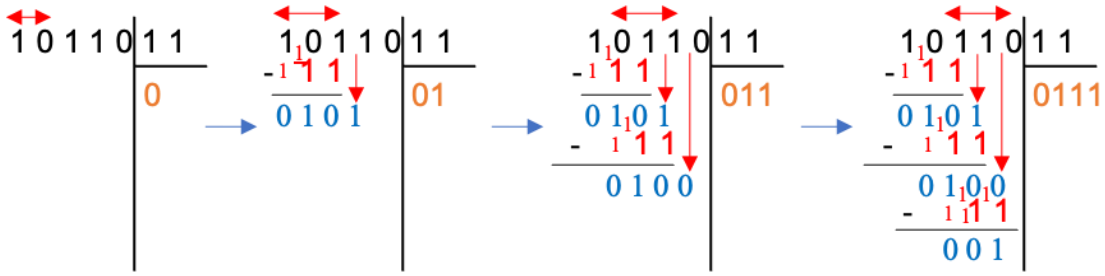
Therefore, $(1111)_2 \times (111)_2 = (1101001)_2$

1.6.4. Division

In binary division:

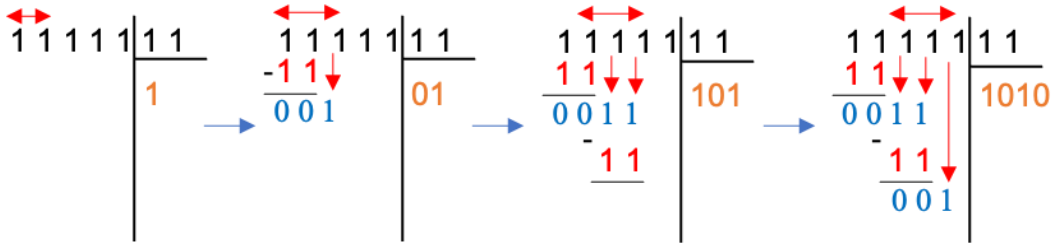
- If the divisor is larger than the dividend (or the current partial dividend), the quotient digit is 0.
- If the divisor is smaller than or equal to the dividend, the quotient digit is 1, and subtraction is performed.

Example 1: compute $(10110)_2 / (11)_2$



Therefore, $(10110)_2 / (11)_2 = (0111)_2$ remainder $(001)_2$

Example 2: Compute $(11111)_2 / (11)_2$



Therefore, $(11111)_2 / (11)_2 = (1010)_2$ remainder $(001)_2$

1.7.Character Encoding

1.7.1. ASCII Encoding

Binary representation allows computer systems to encode and manipulate numerical data. However, computers must also be able to store, process, and transmit textual information composed of alphanumeric characters. To address this need, each character is assigned a unique binary value through a standardized encoding system known as ASCII (American Standard Code for Information Interchange).

ASCII is a character encoding standard that represents characters using **8 bits** in computer memory. It relies on a predefined table that includes the most frequently used characters, such as uppercase and lowercase letters (A–Z, a–z), digits (0–9), punctuation symbols (., ,, ;, :,?), and special characters (@, $, %, etc.).

For example, the character **'A'** is represented in ASCII by the decimal value **65**, which corresponds to the binary code **01000001**.

ASCII	Caractère	ASCII	Caractère	ASCII	Caractère	ASCII	Caractère	ASCII	Caractère	ASCII	Caractère	ASCII	Caractère	ASCII	Caractère
0	NUL	32	Space	64	@	96	`	128	€	160		192	À	224	à
1	SOH	33	!	65	A	97	a	129	□	161	¡	193	Á	225	á
2	STX	34	"	66	B	98	b	130	‚	162	¢	194	Â	226	â
3	ETX	35	#	67	C	99	c	131	ƒ	163	£	195	Ã	227	ã
4	EOT	36	$	68	D	100	d	132	„	164	¤	196	Ä	228	ä
5	ENQ	37	%	69	E	101	e	133	…	165	¥	197	Å	229	å
6	ACK	38	&	70	F	102	f	134	†	166	¦	198	Æ	230	æ
7	BEL	39	'	71	G	103	g	135	‡	167	§	199	Ç	231	ç
8	BS	40	(	72	H	104	h	136	ˆ	168	¨	200	È	232	è
9	HT	41	)	73	I	105	i	137	‰	169	©	201	É	233	é
10	LF	42	*	74	J	106	j	138	Š	170	ª	202	Ê	234	ê
11	VT	43	+	75	K	107	k	139	‹	171	«	203	Ë	235	ë
12	FF	44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
13	CR	45	-	77	M	109	m	141	□	173		205	Í	237	í
14	SO	46	.	78	N	110	n	142	Ž	174	®	206	Î	238	î
15	SI	47	/	79	O	111	o	143	□	175	¯	207	Ï	239	ï
16	DLE	48	0	80	P	112	p	144	□	176	°	208	Ð	240	ð
17	DC1	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	DC2	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	DC3	51	3	83	S	115	s	147	“	179	³	211	Ó	243	ó
20	DC4	52	4	84	T	116	t	148	”	180	´	212	Ô	244	ô
21	NAK	53	5	85	U	117	u	149	•	181	µ	213	Õ	245	õ
22	SYN	54	6	86	V	118	v	150	–	182	¶	214	Ö	246	ö
23	ETB	55	7	87	W	119	w	151	—	183	·	215	×	247	÷
24	CAN	56	8	88	X	120	x	152	˜	184	¸	216	Ø	248	ø
25	EM	57	9	89	Y	121	y	153	™	185	¹	217	Ù	249	ù
26	SUB	58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú
27	ESC	59	;	91	[	123	{	155	›	187	»	219	Û	251	û
28	FS	60	<	92	\	124	|	156	œ	188	¼	220	Ü	252	ü
29	GS	61	=	93	]	125	}	157	□	189	½	221	Ý	253	ý
30	RS	62	>	94	^	126	~	158	ž	190	¾	222	Þ	254	þ
31	US	63	?	95	_	127	DEL	159	Ÿ	191	¿	223	ß	255	ÿ

Figure 1.1. ASCII code table

1.7.2. Unicode Encoding

Unlike ASCII, which is limited to character codes ranging from 0 to 255, Unicode uses much larger code values in order to support a vast number of characters. Although ASCII provides an efficient solution for encoding basic Latin characters, it is limited in its ability to represent the wide variety of characters used in different languages worldwide. These limitations led to the development of more comprehensive character encoding standards, such as Unicode, which aim to support a universal and extensible set of characters.

The Unicode standard enables the representation of characters from many different languages and writing systems, such as Arabic, Chinese, and Tifinagh. In Unicode, each character is assigned a unique code point, which may be represented using up to 32 bits, allowing the encoding of millions of distinct characters. This approach ensures consistent and universal text representation across different platforms and applications.

Examples:

- The Latin character 'A' has the Unicode code point U+0041, which corresponds to 65 in decimal and is represented in binary (16 bits) as 00000000 01000001.
- The Arabic letter 'م' has the Unicode code point U+0645, which corresponds to 1605 in decimal and is represented in binary (16 bits) as 00000110 01000101.
- The Tifinagh character 'ⵣ' (Yaz) has the Unicode code point U+2D63, which corresponds to 11619 in decimal and is represented in binary (16 bits) as 00101101 01100011.

1.8. Image Encoding: Black & White, Grayscale, and Color

Just like characters, digital images are represented in computers using binary data. The way an image is encoded in memory depends on its type and the level of detail required. Image encoding converts visual information into sequences of bits that can be stored, processed, and displayed by computer systems.

Digital images are composed of thousands or even millions of pixels (picture elements), which become visible when the image is sufficiently magnified. Each pixel in the image is represented by a set of bits, which gives rise to three main types of image encoding :

1.8.1. Black & White Images (Binary Images)

Black and white images, also called binary images, are the simplest form of digital images.

- Each pixel is represented by 1 bit:
  - 0 → black
  - 1 → white

- This type of encoding is very memory-efficient but cannot represent any shades or gradients.
- Example: A small 4×4 pixel binary image could be represented as:

```
0 1 0 1
1 0 0 1
0 0 1 0
1 1 0 0
```

1.8.2. Grayscale Images

Grayscale images can represent shades between black and white, using multiple bits per pixel (commonly 8 bits, allowing 256 levels).

- Each pixel has a value from 0 (black) to 255 (white).
- Intermediate values represent different shades of gray, allowing more realistic images.
- **Example:** A pixel value of `128` represents medium gray.
- Memory requirement: width × height × 8 bits per pixel.

1.8.3. Color Images

Color images are encoded using multiple channels, most commonly RGB (Red, Green, Blue)**.**

- Each color channel typically uses 8 bits, so one pixel = 24 bits.
- Each channel can represent 256 intensity levels, allowing more than 16 million possible colors.
- **Example:**
  - `(255, 0, 0)` → pure red
  - `(0, 255, 0)` → pure green
  - `(0, 0, 255)` → pure blue
- Memory requirement: width × height × 24 bits per pixel.

Other color models exist, such as CMYK (used in printing) or HSV (useful in image processing applications), but RGB remains the most common for digital displays.

1.8.4. Additional Remarks

- The concept of binary encoding used for characters (ASCII, Unicode) is analogous to images: each pixel is ultimately stored as a sequence of bits.
- Image files are often compressed (e.g., JPEG, PNG) to reduce memory usage while preserving visual quality.
- The choice between black & white, grayscale, or color depends on the application and available memory.

1.9. Principle of Computer Operation

When a computer starts, it first launches a special software called the Operating System (OS), such as Windows, Linux, or macOS. Without an OS, the computer would be unable to interpret user commands and display results, essentially remaining a machine with very limited functionality. The OS acts as an intermediary between the user and the hardware, interpreting commands and managing the execution of applications.

Modern operating systems typically display a desktop (home screen) containing icons that represent installed software. Clicking on these icons allows the user to launch programs. The process of opening and running a software application can be described step by step:

1. The user clicks or double-clicks on the program's icon.
2. The computer determines which software the icon corresponds to on the storage drive.
3. The program is read from the hard drive (or SSD), where it is permanently stored.
4. The data and instructions of the program are loaded into the computer's RAM (Random Access Memory).
5. The processor (CPU) executes the program instruction by instruction from RAM.
6. When the program is closed, its data is removed from RAM, freeing memory for other tasks.
7. The program remains stored on the hard drive for future use.

It is important to note that RAM, the processor, and the hard drive are all essential components of the computer's central processing unit (CPU system). RAM provides temporary storage for running programs, the processor performs the computations, and the hard drive ensures permanent storage of data and software.

1.10. Computer Hardware

Computer hardware refers to the physical components of a computer system that can be seen and touched. These components work together to execute programs, process data, and interact with the user. Hardware forms the foundation of any computer system and is essential for its proper functioning:

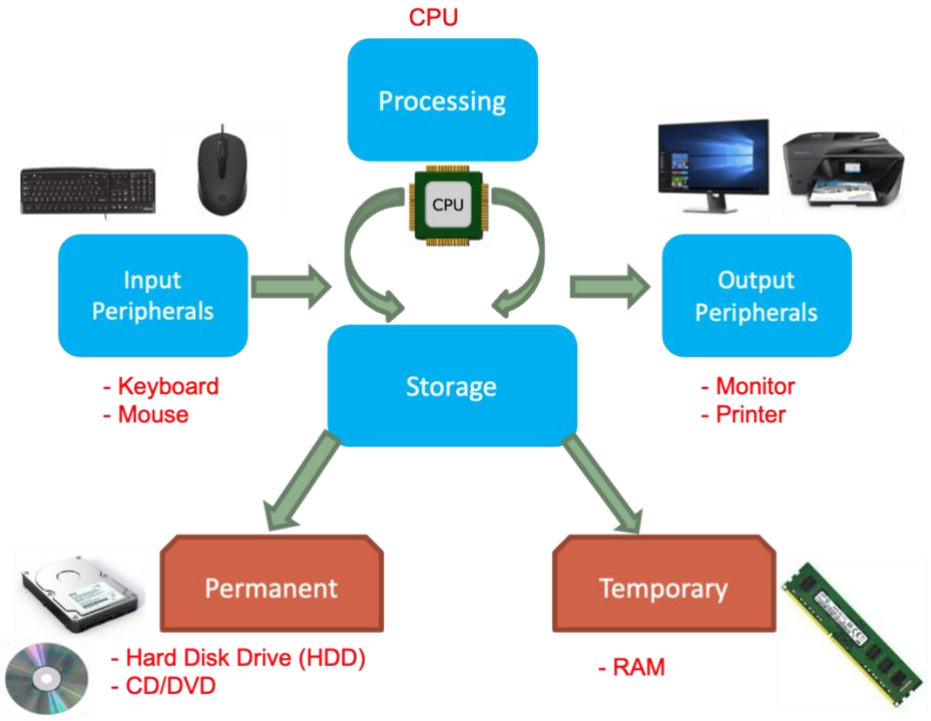


Figure 1.2. The computer hardware

1.10.1. Central processing unit

The central processing unit (CPU) case houses all the electronic components necessary for a computer to function. It contains all the essential parts that communicate with each other to run programs and process data.

- **Motherboard**: Often called the orchestra conductor of the computer, the motherboard connects all other components of the central unit, allowing them to communicate efficiently.
- **Processor (CPU)**: Known as the brain of the computer, it handles all binary calculations and executes instructions whenever a file is opened, a program is run, or data is saved. Its performance is measured in GHz (gigahertz).
- **Hard Drive**: This is the computer's permanent memory, storing all data and software. Its capacity is typically expressed in GB (gigaBytes) or TB (teraBytes).
- **RAM (Random Access Memory)**: A fast, volatile memory used to temporarily store data and programs currently in use. The information in RAM is lost when the computer is turned off. Its capacity is usually indicated in KB, MB, or GB.
- **Graphics Card**: Responsible for rendering images and videos on the screen, the graphics card's performance can also be expressed in GHz, depending on the required graphical quality.
- **Power Supply Unit (PSU)**: The computer's electrical center. It converts the main voltage (230 V AC) into lower voltages (typically 12 V DC) to supply power to all internal components via color-coded cables.
- **Input/Output Connectors (I/O ports)**: These allow the connection of external peripherals, such as the keyboard, mouse, speakers, network cables, and other devices.

1.10.2. Peripherals

Peripherals are devices that allow a computer to communicate with the outside world, either by sending information to the central unit or by receiving information from it. There are four main types of peripherals:

- **Input Peripherals**: These devices send information to the computer. Examples include the keyboard, mouse, microphone, scanner, and webcam.
- **Output Peripherals**: These devices receive information from the computer. Examples include the monitor, printer, speakers, and headphones.
- **Communication Peripherals (Input/Output)**: These devices both send and receive information simultaneously, enabling communication between computers or with networks. Examples include routers, modems, wireless connections (Wi-Fi, Bluetooth), and wired connections (RJ45 cables).
- **Storage Peripherals**: These devices store information permanently or temporarily. Examples include USB flash drives, memory cards, and external hard drives.

1.11. Computer Software

From the user's point of view, software refers to applications designed to satisfy specific needs, such as word processing, spreadsheet calculations, web browsing, graphic design, or multimedia playback.

From a computer science perspective, software is a set of instructions written in a programming language. These instructions are invisible to the user but interpretable by the machine, enabling it to perform specific tasks. A software system is not limited to a single program; it typically includes several programs, data files, and a user interface that work together.

Software cannot operate independently and always depends on hardware components such as the processor, memory, and storage devices. It acts as an essential interface between the user and the computer hardware, translating user actions into machine-executable operations.

Software is generally classified into three main categories: system software, programming languages, and application software.

1.11.1. System Software

System software is essential for the proper operation of a computer system. It manages hardware resources and provides a stable platform on which application software can run.

1. **BIOS (Basic Input/Output System)**: The BIOS is stored in the read-only memory (ROM) of the motherboard. Its primary role is to initialize hardware components and start the computer during the boot process.
2. **Operating System (OS)**: The operating system is a collection of programs responsible for managing all hardware and software resources of the computer. It controls memory management, file storage, process execution, and user interaction.

   Common operating systems include Windows (XP, Vista, 7, 8, 10, 11), macOS**,** Linux**,** UNIX, and Chrome OS.

System software requires regular updates to enhance performance, correct errors, and ensure system security.

1.11.2. Programming Languages

A programming language is a means of communication between humans and computers, used to write instructions that allow the computer to perform specific tasks.

- **Examples:**
  Common programming languages include C, C++, Java, and Python, each designed for different purposes such as system programming, software development, and data processing.
- Differences between programming languages:
  - **Low-level languages** (e.g., **C**) are closer to hardware. They provide better performance and greater control over system resources but are more complex to use.
  - **High-level languages** (e.g., **Python**) are closer to human language. They are easier to learn, write, and maintain, but generally offer less direct control over hardware.

1.11.3. Application Software

Application software is designed to perform specific tasks and meet particular user needs. Examples include:

- **Office software:** Microsoft Word, Excel, PowerPoint
- **Internet software:** Google Chrome, Mozilla Firefox, Internet Explorer
- **Multimedia software:** Adobe Photoshop, Paint, VLC Media Player
- **Games:** FIFA, Need for Speed, Super Mario
- **Security software:** Avast, Kaspersky, Norton, Avira

Application software relies on the operating system to access hardware resources and function correctly.

In summary, software plays a fundamental role in a computer system by enabling interaction between users and hardware. It allows computers to perform a wide range of tasks, from simple operations to complex data processing, making modern computing both efficient and accessible.

1.12. Exercises

Exercise 1.1

Arrange the following measurements in ascending order :
8234 KB, 5.3 MB, 66 MB, 3 Bytes, 21 bits, 7.4 GB

Exercise 1.2

Consider a digital color image with dimensions 1024×7681024×768 pixels. Each pixel is encoded using three components (Red, Green, Blue), each represented by 8 bits.

1. Calculate the total size of the image in MB.
2. Determine the total number of colors that can be represented in this image.
3. If this image is transmitted over a connection with a data rate of 100 KB/s, determine the time required for the transfer.
4. Compare the transfer time if the connection speed were 1 MB/s instead of 100 KB/s.

Exercise 1.3

Convert the following numbers to the indicated bases:

1. $(2513)_4 = (\ ?\ )_{10}$
2. $(365)_8 = (\ ?\ )_{10}$
3. $(A9F)_{16} = (\ ?\ )_{10}$
4. $(21012.21)_3 = (\ ?\ )_{10}$
5. $(0.625)_{10} = (\ ?\ )_2$
6. $(45.375)_{10} = (\ ?\ )_2$
7. $(110011.101)_2 = (\ ?\ )_{10}$
8. $(58)_{10} = (\ ?\ )_8$
9. $(93)_{10} = (\ ?\ )_{16}$
10. $(146.75)_{10} = (\ ?\ )_2$

Exercise 1.4

Express the following numbers in the indicated bases:

1. $(254)_6 = (\ ?\ )_2$
2. $(2312)_4 = (\ ?\ )_2$
3. $(1011011101)_2 = (\ ?\ )_{14}$
4. $(173.041)_9 = (\ ?\ )_7$
5. $(B3C)_{16} = (\ ?\ )_{11}$

Exercise 1.5

Determine directly, without converting through base 10, the following numbers in the indicated bases:

1. $(10111011101)_2=(?)_4=(?)_8=(?)_{16}$
2. $(3F9,B)_{16}=(?)_2=(?)_4=(?)_8$
3. $(57,34)_8=(?)_2=(?)_4=(?)_{16}$
4. $(2101021)_3=(?)_9$

Exercise 1.6

Let the decimal number be:

$$Y=5b^6+3b^4+2b^2+4b+7$$

with b>5, an integer.

1. Determine Y in base b.
2. Determine the base n knowing that:

$$(143)n=(99)_{10}$$

3. Determine the two-digit integers (xy) in base 7 that are written as (yx) in base 10.

Exercise 1.7

Perform the following operations, without converting through base 10:

1. $(11101)_2 + (10011)_2=(\ldots\ldots\ldots)_2$
2. $(1101.101)_2 + (101.011)_2=(\ldots\ldots\ldots)_2$
3. $(101.1101)_2 + (11.101)_2=(\ldots\ldots\ldots)_2$
4. $(1101.01)_2 + (101.111)_2 + (11.101)_2=(\ldots\ldots\ldots)_2$
5. $(11100.11)_2 - (101.01)_2=(\ldots\ldots\ldots)_2$
6. $(1011.01)_2 * (110)_2=(\ldots\ldots\ldots)_2$
7. $(10.111)_2 * (1011)_2=(\ldots\ldots\ldots)_2$
8. $(11100)_2 / (101)_2=(\ldots\ldots\ldots)_2$
9. $(110101)_2 / (111)_2=(\ldots\ldots\ldots)_2$
10. $(542)_8 + (317)_8=(\ldots\ldots\ldots)_8$
11. $(46.3)_8 * (5.2)_8=(\ldots\ldots\ldots)_8$
12. $(735)_8 / (7)_8=(\ldots\ldots\ldots)_8$
13. $(2B7)_{16} + (C4F)_{16}=(\ldots\ldots\ldots)_{16}$
14. $(A3F)_{16} - (56C)_{16}=(\ldots\ldots\ldots)_{16}$
15. $(F8C)_{16} / (1A)_{16}=(\ldots\ldots\ldots)_{16}$

Chapter 2: Notions of Algorithms

2.1. Introduction

This chapter presents the fundamental concepts of computer science with a focus on algorithmics. The goal is to familiarize students with the problem-solving approach in computing, using algorithms and control structures to organize and automate tasks.

It is important to note that these concepts are studied in parallel with programming, but for organizational reasons, the course is divided into two chapters:

1. **Algorithmics** : to understand the logic and design of programs.
2. **Programming in C** : to learn how to translate this logic into instructions executable by a computer, which will be detailed in the next chapter.

It provides the essential foundations for transitioning from algorithm theory to their practical implementation in a programming language.

2.2. Concept of an algorithm

Algorithm is a term of Arabic origin, honoring Muhammad Ibn Musa Al Khawarizmi (780–850), the author of a work on algebraic calculation methods. An algorithm is a structured method for solving a problem, expressed as a sequence of operations to be executed in a specific order. Implementing an algorithm in a programming language forms the basic building block of a computer program.

Algorithms are not limited to computer science, they can be observed in everyday life. For example:

- When you go shopping, you follow a sequence of steps: making a list, selecting products, and paying.
- When giving directions to a tourist, you provide ordered instructions to reach a destination.
- When following a cooking recipe, you execute a set of steps in a specific order to prepare a dish.

These everyday situations illustrate the core idea of an algorithm: a clear and ordered sequence of actions designed to achieve a specific goal.

2.3. Forms of representation

An algorithm can be expressed in different forms to make it easier to understand, communicate, and implement. The most common forms of representation are:

- **Flowchart:** A graphical representation of an algorithm using symbols such as ovals, rectangles, diamonds, and arrows to illustrate the sequence of operations, decisions, and processes. Flowcharts are particularly useful for visualizing the logic and flow of a program.

Example:

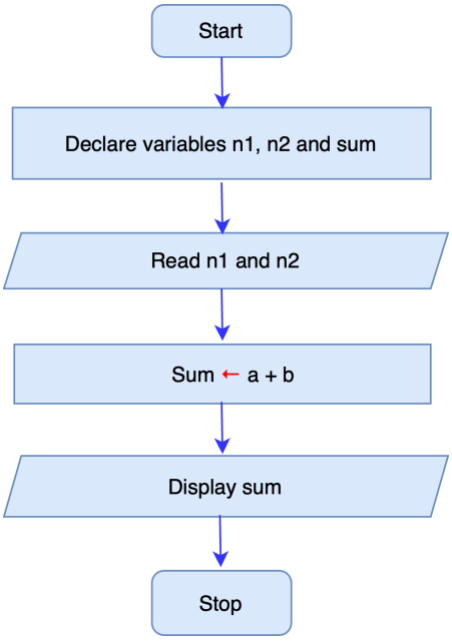


Figure 2.1 Flowchart for Adding Two Numbers

- **Pseudo code** : A textual representation of an algorithm that uses a structured, human-readable format resembling programming languages. Pseudo code allows programmers to describe the logic of an algorithm without worrying about the syntax of a specific programming language.

Example:

```
Algorithm Sum_Two_Numbers
Variables
n1, n2, sum: integer
Begin
Write("Enter A:")
Read(n1)
Write("Enter B: ")
Read(n2)
sum ← a + b
Write("The sum is:", sum)
End
```

Figure 2.2 Pseudo code for Adding Two Numbers

Both forms help in planning, analyzing, and debugging algorithms before actual implementation in a computer program.

2.4. Problem-Solving Approach and Analysis

The computer-based solution of such a problem generally involves the following phases :

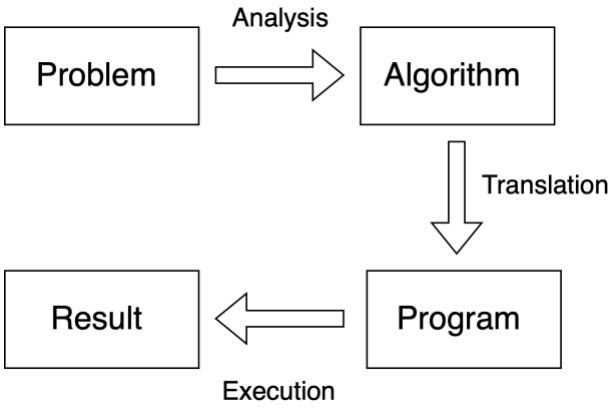


Figure 2.3 Phases of a Computer-Based Problem-Solving Process

Developing a computer program that can be executed by a machine requires following a structured and well-defined approach composed of several steps:

1. Problem Definition: A problem is usually proposed by a person or an organization that needs a computer-based solution, such as a pharmacist, a student, or a bank. In many cases, the requester may not be able to clearly express their needs in technical terms.

The goal of this step is to clearly define and reformulate the problem so that it can be correctly understood and solved.

2. Problem Specification and Analysis: The objective of this phase is to fully understand the problem statement and identify the necessary elements for its resolution, such as calculation formulas and processing rules. Problem analysis focuses on the following components:

- **Expected results (outputs):** what the program should produce.
- **Processing (operations):** the actions required to obtain the results.
- **Required data (inputs):** the information needed to perform the processing.

3. Algorithm Design: Once the analysis is complete, the solution is expressed by organizing the instructions in a logical and sequential order. This step leads to the creation of an algorithm, which describes the solution independently of any programming language.

4. Program Implementation: After defining the algorithm, it must be translated into a form that the computer can execute. This is done by writing the program using a programming language, while respecting its syntax and rules.

5. Program Execution and Testing: Once compiled or interpreted, the program must be tested to verify that it functions correctly and meets the user's requirements. Testing is carried out using different sets of input data, known as test cases, in order to detect errors and validate the correctness of the results.

2.5. Data Structure

2.5.1. Data: Variables and constants

Data represents information related to an element of the problem being solved by an algorithm. Data can be classified into two main categories:

1. Variable: A variable is a memory location that stores data whose value can change during the execution of an algorithm. Each variable is characterized by:

- **Name**: the identifier used to refer to the variable.
- **Type**: defines the kind of data the variable can hold (e.g., integer, real, character).
- **Value**: the current data stored in the variable.

2. Constant: A constant is a memory location that stores data whose value cannot change during the execution of an algorithm. Each constant is characterized by:

- **Name**: the identifier used to refer to the constant.
- **Value**: the fixed data stored in the constant.

2.5.2. Data type

The data type of a variable indicates the set of values that the variable can hold. In algorithms, we usually define only the basic types, which are:

1. **Integer**: numbers without a decimal point and can be positive or negative. They can be represented on 16 bits (short integer) or 32 bits (long integer), depending on the memory allocation. The range of possible integer values is given by: $-2^{n-1} \leq N \leq +2^{n-1} - 1$, where *N* is the Integer number to express and *n* is the number of bits to represent the integer in memory. **Example:** 0, -25, 1024, 50000.
2. **Real**: numbers with a decimal point (also called floating-point numbers). They can be stored using 32 bits (single precision) or 64 bits (double precision). **Example:** 3.14, -0.75, 0.001, 2.71828.
3. **Boolean**: variables can hold only one of two possible values: true or false. These values can also be represented numerically as 1 (true) or 0 (false). Booleans are commonly used for decision-making in algorithms.
4. **Characters (Char/String)**:The char type stores a single character, which can be alphabetical, numeric, or symbolic. The string type represents a sequence of characters, i.e., multiple characters grouped together to form words or sentences. **Example (char):** 'A', '7', '?'; **Example (string):** 'Hello', '123', 'Algorithm'.

2.5.3. Identifier

Each data item (variable or constant) manipulated by an algorithm is identified by a unique name called an identifier.

An identifier is an alphanumeric string that can contain:

- Alphabetical characters (a–z, A–Z),
- Numeric digits (0–9),
- The underscore symbol (_).

The identifier must follow these rules:

- Cannot start with a digit,
- Contain no special characters (such as spaces) or punctuation,
- An identifier is assigned to a single object, and the same identifier cannot be used for multiple objects.
- Not be a reserved word of the algorithmic language (such as algorithm, begin, end, variables, not, or, if, else, for, etc.).

Examples:

- `Var123`
- `My_variable`
- `name`
- `Number_of_phone`

2.5.4. Declaration of Variables and Constants

Declaration allows the computer to be informed of the existence of a data item; that is, it requests the computer's permission to reserve a memory space where the information can be stored and retrieved.

2.5.4.1. Declaration of a Constant

A constant is characterized by its identifier and its fixed value.

Syntax:

```
Constant Constant_identifier = value
```

Examples:

```
Constant PI = 3.14
Constant num_Month = 12
Constant university_name = " UMAB"
```

2.5.4.2. Declaration of a Variable

A variable is characterized by its name, its value, and its type.

Syntax:

```
Variables Variable_identifier: Type
```

Examples:

```
Variables student_id: integer
Variables exam_score: real
Variables last_name, first_name: string
Variables valid: boolean
Variables op: char
```

2.6. Operators

2.6.1. Assignment Operator

Assignment is an operation that consists of giving a value to a variable. The value assigned can be:

- A specific value,
- A value stored in another variable,
- A value calculated using arithmetic operators.

In many algorithmic representations, the assignment is indicated by a left-pointing arrow (←).

Syntax:

```
Variables Variable_identifier ← value
```

Examples:

```
X, Y, Z: Integer
Name   : String
X ← 5             // The variable X receives the value 5
Y ← X             // The variable Y receives the value stored in X
Z ← X + Y         // The variable Z receives the result of X plus Y
Name ← "Lina"     // The variable Name receives the value LINA
```

2.6.2. Arithmetic Operations

Arithmetic operations are used to perform mathematical calculations on numeric data. They are essential in algorithms for processing values, computing results, and solving problems.

The usual arithmetic operators are:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
div	Integer division (returns the result as an integer)
mod	Modulo (remainder of the division)
^	Power

Arithmetic operations can be combined in expressions, which are sets of values linked by operators and equivalent to a single value.

Examples:

```
A, D: Real
B, C: Integer
A ← 5/2          // The variable A receives the value 2.5
B ← 5 div 2     // The variable B receives the value 2
C ← 5 Mod 2    // The variable C receives the value 1
D ← 5 ^ 2     // The variable D receives the value 25
```

2.6.3. Relational Operators

Relational operators, also called comparison operators, are used to compare two values. The result of a comparison is always a boolean value: True or False.

Common relational operators include:

Operator	Meaning
=	Equal to
<>	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Examples:

```
A, B, C: Boolean
A ← 10 > 2 // The variable A receives the value True
B ← 8 = 10 // The variable B receives the value False
C ← (19 mod 7) = (10 div 2) // The variable C receives the value True
```

2.6.4. Logical Operators

Logical operators (also called boolean operators) are used to combine or modify boolean expressions (True/False). The logical operators used in algorithmics are: AND, OR, and NOT.
Let A and B be two boolean variables. The following table illustrates the different truth values obtained by combining the values of A and B using logical operators:

A	B	A AND B	A OR B	NOT A	NOT B
True	True	True	True	False	False
True	False	False	True	False	True
False	True	False	True	True	False
False	False	False	False	True	True

Examples:

```
A, B: integer
C, D, E: Boolean
A ← 10
B ← 2
C ← (A > B) AND (A>20) // The variable C receives the value False
D ← (A=10) OR (B<10) // The variable D receives the value True
E ← Not(C) // The variable E receives the value True
```

2.6.5. Operator Precedence

Operator precedence defines the order in which operators are evaluated in an expression. Operators with higher precedence are executed before operators with lower precedence.

Typical Precedence Rules (from highest to lowest):

1. Parentheses **()** : expressions inside parentheses are evaluated first,
2. Power **^** : exponentiation,
3. Unary operators : **NOT**, negative sign **-**.
4. Multiplication *****, Division **/**, Integer Division **div**, Modulo **mod**,
5. Addition **+** and Subtraction **,**
6. Relational (comparison) operators **(>, <, >=, <=, =, <>)**,
7. Logical **AND**,
8. Logical **OR**

Examples:

```
X,Y: Integer
Z : Boolean
X ← 3 + 4 * 2          // X = 11, multiplication first
Y ← (3 + 4) * 2        // Y = 14, parentheses change the order
Z ← NOT X>Y AND TRUE // Z = FALSE NOT(X>Y) evaluated first
```

2.7. Input/Output Operations

Input/Output operations (I/O) are an essential part of any algorithm, allowing the system to interact with the user, read data from files, or communicate with other devices.

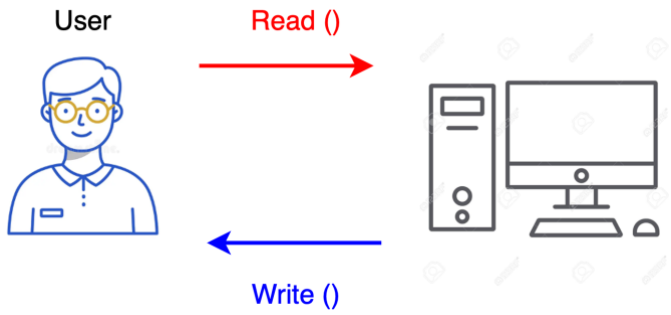


Figure 2.4 User-Computer I/O Flow

2.7.1. Write (Output Operation)

The Write instruction is used to display the value of an expression on an output device (such as the screen). An expression can be:

- A numeric variable,
- The result of an operation involving one or more variables,
- A string of characters; in this case, the string must be enclosed in quotation marks.

Syntax:

```
Write(Variable)
Write("Message ")
Write("Message ", Variable)
```

Examples:

```
Write(7) // It means displaying on the screen the number 7.
Write(A) // It means displaying on the screen the content of variable A.
Write ("Hello ") //It means displaying on the screen the following message:
Hello.
Write(" the value of A is equal to : ", A) // It means displaying on the
screen the following message: The value of A is equal to: followed by the
content of variable A.
```

2.7.2. Read (Input Operation)

The Read instruction is used to ask the user to provide information. Each piece of information entered by the user is stored in a variable according to its data type.

Syntax:

```
Read(Variable1)
Read(Variable1, Variable2,…)
```

```
Read(A) // Ask the user for the value of A
Read(First_Name, Last_Name, Age) // Ask the user for the first name, last name, and Age.
```

Notes:

- During the execution of the *Read* instruction, the machine waits for the user to provide a value before continuing to execute the algorithm.
- Input can only be assigned to variables.
- The data entered must match the type of the variable that receives it.

2.8. Structure of an Algorithm

An algorithm must follow a well-defined and logical structure in order to be clear, understandable, and easy to implement in a programming language. This structure is generally composed of the following parts:

- **Header:** allows the algorithm to be identified with a unique name (identifier)
- **Declarations:** This section contains the declaration of all data elements used by the algorithm,
- **Body** (Statements): the section containing the instructions to be executed (Input, Processing, and Output). All Statements must be written in a logical and sequential order.

Note: All algorithm statements must be placed between the keywords **Begin** and **End**, which clearly indicate the start and termination of the algorithm.

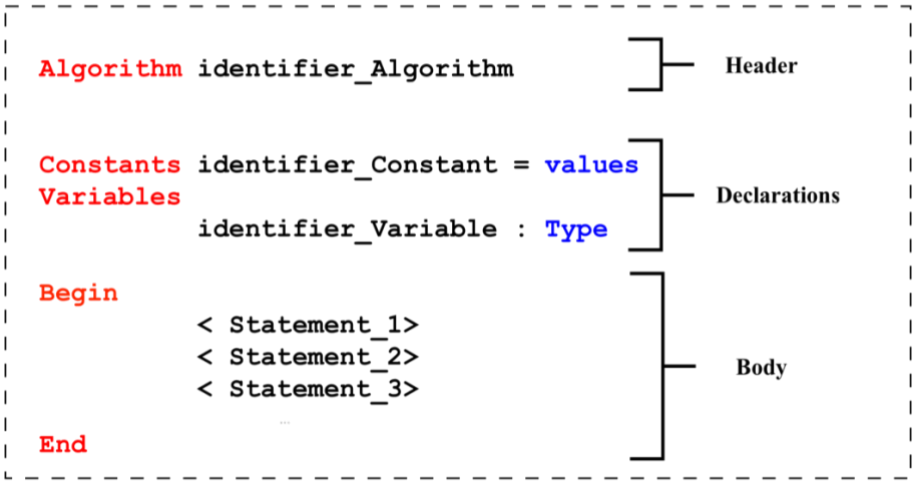


Figure 2.5 Structure of an Algorithm

Example : This example illustrates the general structure of an algorithm used to compute the surface of a disk. The detailed statements will be introduced in the following sections.

```
Algorithm     Surface_disc
Constants     Pi=3.14
Variables
              R,S: real
Begin

<Statements>
…
end
```

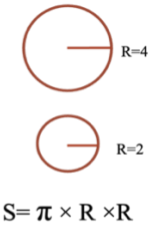


Figure 2.6 Example of an Algorithm

2.9. Control Structures

So far, we have used sequential algorithms, where instructions are executed one after another. However, to solve more complex problems, it is necessary to control the flow of execution. Control structures allow making decisions or repeating instructions based on conditions.

2.9.1. Conditional Control Structure

2.9.1.1. Simple conditional structure

The Simple conditional structure is a structure in which the instructions are executed according to the outcomes of the conditions.

Syntax:

```
If   Condition_is_true   then
      Statements
EndIf
```

If the condition is true, then the instruction block will be executed; otherwise, it will be ignored.

Example: Write an algorithm that divides two integers.

```
Algorithm Divide_Two_Integers
Variables
    A, B : integer
Begin
    Write("Enter the numerator:")
    Read(A)
    Write("Enter the denominator: ")
    Read(B)
    If B <>0 Then
            Write("The result of the division is: ", A / B)
    EndIf
End
```

2.8.1.2. Alternative conditional structure

An alternative structure (also called a selection structure) allows an algorithm to choose between two or more possible paths based on a condition.

Syntax*:*

```
if   Condition_is_true   then
      Statement_1
else
      Statement_2
EndIf
```

If the condition is true, then instruction block 1 is executed and instruction block 2 is ignored; otherwise, instruction block 2 is executed and instruction block 1 is ignored.

Example: Write an algorithm that divides two integers.

```
Algorithm Divide_Two_Integers
Variables
    A, B : integer
Begin
    Write("Enter the numerator:")
    Read(A)
    Write("Enter the denominator: ")
    Read(B)
    If B <>0 Then
            Write("The result of the division is: ", A / B)
    Else
        Write("Error: Division by zero is not allowed")
    EndIf
End
```

2.8.1.3. Nested alternative structure

It is a conditional structure where an if statement is placed inside another if statement to test more than one condition.

Syntax:

```
if Condition1_is_true then
      Statement_1
else
    if Condition2_is_true then
      Statement_2
    else
      Statement_3
    endif
endif
```

Example: Write an algorithm that reads a number and displays whether it is positive, negative, or zero.

```
Algorithm Check_Number_Sign
Variables
    num : real
Begin
    Write("Enter a number: ")
    Read(num)
    If num > 0 Then
        Write("The number is positive")
    Else If num < 0 Then
        Write("The number is negative")
    Else
        Write("The number is zero")
     EndIf
  EndIf
End
```

2.8.1.4. Multiple-choice structure: Case

When the nesting of alternative conditional statements becomes too complex, using a multiple-choice structure (Case structure) becomes necessary.
This structure is particularly useful when the same variable needs to be compared with several specific values (not ranges).

Syntax:

```
Case variable of
Value 1: Statement_1
Value 2: Statement_2
…
…
Value n: Statement_n
Else
     anotherStatement
EndCase
```

Example: Write an algorithm that allows you to enter a number between 1 and 5 and translate this number into a letter.

```
Algorithm chiffres
variables
      nbr: integer
begin
      write("Enter an integer between 1 and 5: ")
              read(nbr)
            case nbr  of
             1    : write("One")
```

```
            2    : write("two")
            3    : write("Three")
            4    : write("Four")
            5    : write("Five")
          else
          write("You may have entered another number");
          endCase
end
```

2.9.2. Repetitive Control Structures

Repetitive structures(Loops) allow the execution of a sequence of instructions multiple times. In a loop, the number of repetitions can be known and fixed in advance, or it can depend on a condition that determines when the loop should stop and exit.

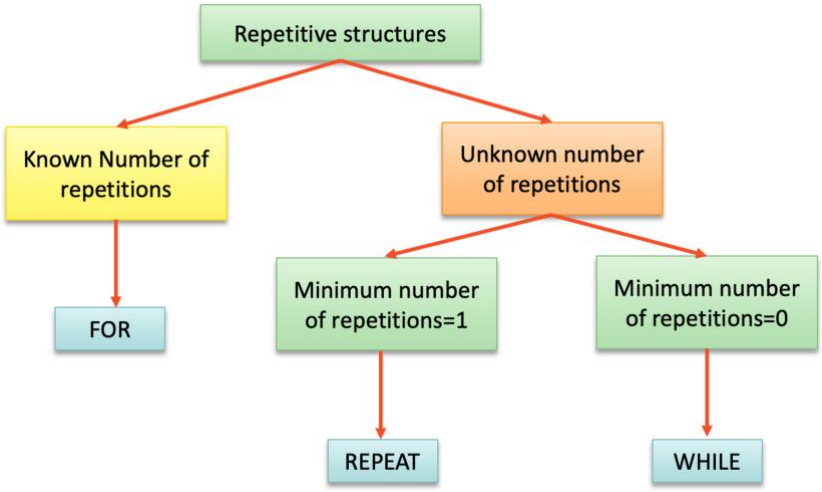


Figure 2.7 Repetitive Control Structure Selection Diagram

2.9.2.1.The FOR loop

The FOR loop is a complete iterative control structure. It allows the execution of a sequence of instructions a known and fixed number of times. This loop uses a control variable (counter) characterized by:

- Its initial value,
- Its final value,
- Its step of variation (the counter is incremented automatically).

Syntax:

```
For cpt ← iv To fv Step val Do
Statement_1
Statement_2
...
Statement_n
End For
```

Where,

1. **cnt :** the counter (an integer variable),
2. **iv:** initial variable (an integer variable),
3. **fv:** final variable (an integer variable),
4. **val**: the increment step (an integer value, usually equal to 1).

Note:

- The number of repetitions= **fv-iv+1**
- In a FOR loop, the step instruction is used to define the increment (or decrement) value of the control variable at each iteration:
    - If step is not specified, the increment is equal to 1 by default.
    - If step is specified, the control variable changes according to the given value.

Example 1: This example displays the numbers from 1 to 5 using a FOR loop. The FOR loop executes a known number of iterations automatically.

```
Algorithm DisplayNumbers
Variables
    i : Integer
Begin
    For i ← 1 to 5 do
        Write(i)
      End For
End
```

Example 2: Write an algorithm that display the even numbers from 1 to N.

```
Algorithm DisplayEvenNumbers
Variables
    i, N : Integer
Begin
    Read(N)
    For i ← 2 to N step 2 do
        Write(i)
    End For
End
```

Using the For loop:

The FOR loop is commonly used when the number of iterations is known in advance. Typical applications include:

- Filling and displaying a table or an array (reading values, storing them, and displaying the results)
- Traversing an array or a character string, such as:
  - computing the sum or product of elements,
  - searching for a specific value or character,
  - testing elements (e.g., checking if a number is positive, negative, or even),
  - counting occurrences of a value or a character.
- Performing repeated calculations (e.g., calculating a factorial, powers, or generating a multiplication table)
- Displaying repetitive patterns or messages (e.g., printing a message a fixed number of times, generating sequences of numbers).

2.9.2.2. The WHILE loop

The WHILE loop is an iterative control structure based on a stopping condition. It allows the repeated execution of an instruction block as long as a Boolean condition remains true. At each iteration, the condition is evaluated: if it is **TRUE**, the loop body is executed and the condition is evaluated again; if it becomes **FALSE**, the loop stops and control exits the loop. Unlike the FOR loop, the control variable (counter) is updated manually inside the loop.

Syntax:

```
While condition Do
Statement_1
Statement_2
...
Statement_n
End While
```

Note :

- Instructions may never be executed if the test is not satisfied,
- the minimum number of executions in the While loop is equal to **zero**,
- The number of repetitions in the loop is not known in advance.

Example: This example displays the numbers from 1 to 5 using a WHILE loop. The WHILE loop executes as long as the condition is TRUE. The counter is updated manually. The condition is checked before each iteration.

```
Algorithm DisplayNumbers
Variables
    i : Integer
Begin
```

```
    i ← 1
    While i <= 5 do
        Write(i)
        i ← i + 1
      End While
End
```

2.8.2.3. The REPEAT loop

The **REPEAT…UNTIL** loop is an iterative control structure based on a stopping condition. It allows the repeated execution of an instruction block until a Boolean condition becomes true. At each iteration, the condition is evaluated: if it is **FALSE**, the loop continues and the next iteration is executed; if it is **TRUE**, the loop stops and control passes to the instruction following the REPEAT…UNTIL loop. Unlike the FOR loop, the control variable (counter) must be updated manually inside the loop.

Syntax:

```
Repeat
Statement_1
Statement_2
...
Statement_n
Until <condition>
```

Note :

- In the loop repeat...until we never use begin and end;
- The number of repetitions in the repeat...until loop is not known in advance,
- The repeat loop is executed at least once.

Example: This example displays the numbers from 1 to 5 using a REPEA loop. The **REPEAT…UNTIL** loop executes the instruction block at least once, and continues until the condition becomes TRUE. The counter is updated manually. The condition is checked after each iteration.

```
Algorithm DisplayNumbers
Variables
    i : Integer
Begin
    i ← 1
    Repeat
        Write(i)
        i ← i + 1
    Until i > 5
End
```

2.9. Exercises

Exercise 2.1

Determine the successive values of the variables after each statement in the following algorithms.

```
Variables
   A, B, C :
integer
   D : Boolean
Begin
   A ← 17
   B ← 5
   C ← A + B
   A ← A - B
   B ← B * 2
   C ← C div 3
   A ← A mod B
   D ← (A > 40) OR
(C < B)
End
```

```
Variables
   A, B, C :
Boolean
Begin
   A ← True
   B ← A
   A ← False
   B ← not A
   C ← A and B
   C ← not (A
or B)
End
```

```
Variables
   A, B :
integer
   C : real
   D, E : char
   Q : Boolean
Begin
   A ← 7
   D ← 'k'
   B ← A + 1
   C ← B / 2
   C ← C - 2
   E ← 's'
   A ← B
   Q ← D > E
End
```

```
Variables
   A, B, C : string
Begin
   A ← '5'
   B ← A + 'a'
   C ← B + A
End
```

Will we obtain the same result if we replace the last instruction by:

C ← A+B

Justify your answer.

Exercise 2.2

Let there be three variables A, B, and C such that:
A is of type integer
B is of type character
C is of type Boolean
For each of the statements below:

- Indicate whether it is Valid or Invalid.
- Provide a brief explanation.

```
1. A = 15 ;
2. B = 'K' ;
3. C = 7 > 10 ;
4. A = 'Z' ;
5. B = 3 ;
6. C = 0 ;
7. A = A + 8 ;
8. B = B + 'M' ;
9. C = not (A > 20) ;
10.   A = B mod 2 ;
11.   C = A + 2 ;
12.   B = 'Q' ;
13.   A = C ;
14.   C = (B > 'A') and (A < 20) ;
15.   B = B + 5 ;
```

Exercice 2.3

Write an algorithm that:

1. Asks the user to enter three marks.
2. Calculates the sum of the marks.
3. Calculates the average of the marks.
4. Displays both the sum and the average.
5. Run the algorithm for the following marks: 10, 15, 9.

Exercice 2.4

Write an algorithm that:

1. Asks the user to enter their name.
2. Asks the user to enter their age.
3. Displays the message: Hello <name>, you are <age> years old and welcome to the University of Mostaganem. Replace <name> and <age> with the user's actual input.

Exercice 2.5

Write an algorithm that calculates the sum, product, difference, division, integer division and division remainder of two integers entered on the keyboard.
Note: Make sure that division by zero does not occur.

Exercice 2.6

Write an algorithm that:

1. Asks the user to enter two numbers**,** `num1` and `num2`.
2. Determines and displays the maximum of the two numbers using an alternative structure.
3. Rewrite the algorithm using a conditional structure.
4. Modify the algorithm to also check if the two numbers are equal**,** and display a message: Both numbers are equal.

Exercice 2.7

Write an algorithm that:

1. Asks the user to enter two numbers, `X` and `Y`.
2. Determines whether the product of `X` and `Y` is positive, negative, or zero.

Note: You must determine the sign of the product without calculating the actual product; use only the signs of `X` and `Y` to decide.

Exercice 2.8

Write an algorithm that reads a letter and determines whether it is a consonant or a vowel.

Exercice 2.9

Write an algorithm that reads `current_day` (the number of the current day from 0 to 6) and `N` (the number of days to add). The algorithm should calculate and display the day of the week after `N` days.

Example:

- Input: `current_day = 3` (Wednesday), `N = 10`
- Output: `The day after 10 days will be Saturday`

Exercice 2.10

Using the multiple-choice instruction (case), write an algorithm that asks the user to enter a number between 1 and 7, then displays the corresponding day of the week. If the number entered is not within this range, the program should display "Error".

Exercice 2.11

Write an algorithm that calculates the sum of the first ten (10) positive integers.

Exercice 2.12

Write an algorithm that asks the user to enter a number and then prints its multiplication table from 1 to 10.

Example*:* If the user enters 5, the program should display:

```
Table of 5 :
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
…
5 x 10 = 50
```

Exercice 2.13

Write an algorithm that asks the user for a value for a real `x` and a positive integer `y` and displays the result of x^y.

Exercice 2.14

Write an algorithm that:

- Asks the user to enter a positive integer `n`.
- Calculates the sum of the first `n` squares $1^2 + 2^2 + 3^2 + \ldots + n^2$.
- Displays all intermediate results as the squares are added.

Exercice 2.15

An astronaut is preparing for takeoff. The countdown starts at 50 seconds.
Write an algorithm that displays the countdown from 50 to 0, decreasing by 3 at each step (step = -3).
After the countdown, display the message: "Launch! ".

Exercice 2.16

Write an algorithm that asks the user to enter a number `n` and a limit `m`, then displays all multiples of `n` from 1 up to `m`.
Example:
Input: n=3, m=10
Output:
3, 6, 9

Chapter 3: Introduction in C language

3.1. Introduction

The C programming language was developed in the early 1970s by Dennis Ritchie at Bell Labs. It is a structured, powerful, and portable language, meaning that C programs can run on different types of computers with minimal or no modification. Its efficiency and precise memory control make it widely used for operating systems, embedded software, scientific applications, and high-performance computing.

Learning C builds directly on the algorithmic concepts studied earlier. Variables, constants, operators, conditional statements, and loops translate naturally into C instructions. As a result, algorithms designed on paper or as flowcharts can be easily implemented in C, bridging the gap between theory and practice.

Programming in C requires a development environment (IDE) or a compiler. Popular IDEs include Code::Blocks, Dev-C++, Visual Studio, while GCC is a common choice for command-line or online compilation.

In this chapter, we focus on practical implementation, using C to apply the concepts of variables, constants, operators, conditions, and loops. To avoid repetition, the theoretical details will not be revisited.

This chapter aims to help the student establish a clear connection between algorithmic design and its practical implementation in C, following a structured and straightforward approach.

3.2.The structure of a program C

A typical C program begins by including the necessary libraries, contains the main function `main()`, and executes a block of instructions enclosed within `{ }`. Each instruction in C must end with a semicolon `;`.

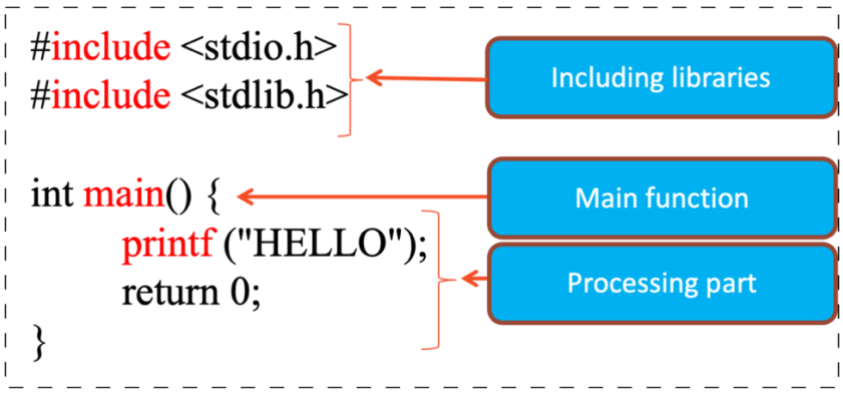


Figure 3.1 Structure of a program C

1. Including Libraries (`#include`)

Libraries provide pre-written functions that the program can use. For example, `#include <stdio.h>` allows the use of the `printf()` function to display text on the screen. The most commonly used libraries are the following:

- **stdlib.h**: (standard library) miscellaneous functions (exit, system, malloc, free);
- **stdio.h**: (standard input-output) display text on the screen, read input from the keyboard;
- **math.h**: mathematical functions (absolute value, power, cosine, etc.);
- **time.h**: time-related functions (hour, date, etc.).

2. The `main()` Function

The `main()` function is the entry point of a C program, execution starts here.

- The `int` before `main()` indicates that the function returns an integer value.
- `return 0;` signals that the program finished successfully.

3. Blocks of Instructions `{ … }`

- Curly braces `{ }` define a block of instructions.
- All statements inside the block must end with a semicolon `;`.

5. Example: "Hello World" Program

Here is a simple example demonstrating the basic structure of a C program:

```
#include <stdio.h>   // Include standard library for input/output functions

int main() {          // Main function: program execution starts here
    printf("Hello World\n");  // Display text on the screen
    return 0;         // End of the program
}
```

3.3. Compilation

Compilation is the process that translates a program's source code, written by the programmer, into machine-executable code before it is run. This process is carried out by a compiler, which analyzes the code, checks its conformity to the language rules, and produces an executable file.

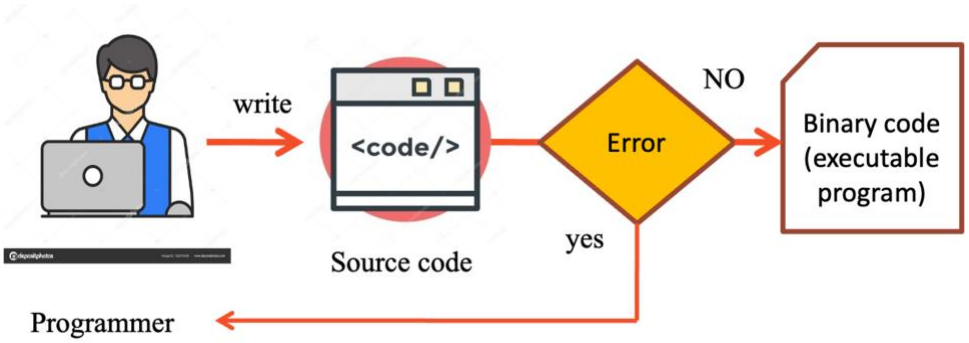


Figure 3.2. The Compilation Process

A programming language defines a set of rules:

- **Syntax**: how to write a program correctly;
- **Semantics**: the meaning assigned to each instruction or program structure.

The compiler plays a key role in learning C:

- It detects syntax errors, such as a missing semicolon or misplaced braces.
- It can also report semantic errors, where instructions are syntactically correct but logically inconsistent, for example using an undeclared variable.
- Once the code is corrected, the compiler generates an executable program that can be run on the computer.

Understanding compilation is therefore essential for moving from writing a C program to executing it on a machine.

3.4. Declaration and Use of Variables and Constants

In C, the concepts of variables and constants that we studied in algorithmics are directly implemented through declaration and initialization statements. Each variable or constant must have:

- a type (indicating the kind of data it can hold),
- a **name** (identifier),
- and optionally an initial value.

3.4.1. Variable Declaration

The most common data types in C are:

- **int** : for integers,
- **long** : for large integers,
- **float** : for real numbers (single precision),
- **double** : for real numbers (double precision),

- **char** : for a single character,
- **bool** : for logical values (true / false).

Syntax:

```
Type Variable_identifier;
```

Example:

```
int age;
float temperature;
char grade;
```

3.4.2. Constant Declaration

A constant is a fixed value that does not change during the execution of a program. In C, the **const** keyword is used.

Syntax:

```
const Type Constant_identifier = value;
```

Example:

```
const float PI = 3.14;
const int MAX = 100;
```

3.4.3. Initialization and Assignment

A variable can be initialized at the time of its declaration or assigned a value later using the = symbol.

Syntax:

```
Variable_identifier = Value;
```

Example:

```
int score = 10;    // Initialization
score = 15;        // Assignment
```

3.5. Operators

In C, operators allow performing calculations, comparisons, and logical operations on variables and constants, just as we studied in algorithmics. The syntax may change, but the logic remains the same.

3.5.1. **Arithmetic Operators**

They are used for numerical calculations:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder of integer division)
++	Increment
--	Decrement

Example 1:

```
int a = 10, b = 3;
int sum = a + b;       // 13
int P = a * b;  // 30
int R = a % b; // 1
a ++//11
b --//2
```

Example 2: The following table illustrates how variables X, Y, Z, and W change after each instruction.

Instruction	X	Y	Z	W
`X = 5 + 3;`	8			
`Y = ++X;`	9	9		
`Z = X * Y++;`	9	10	81	
`W = pow(Z, 2);`	9	10	81	6561
`X /= 3;`	3	10	81	6561
`W = ++W - Z;`	3	10	81	6481
`Z -= Y--;`	3	9	71	6481
`Y = (X-- + --Z) / Y;`	2	8.11	70	6481

Notes:

- ++X increments X before its value is used.
- Y++ increments Y after its value is used.
- pow(Z, 2) computes Z^2 (the library #include <math.h> must be included at the beginning of the program.)
- Compound operators like /= and -= simplify arithmetic assignments.
- This example demonstrates operator precedence, evaluation order, and the effect of increment/decrement operators.

3.5.2. Relational Operators

They are used to compare values and return 1 for true and 0 for false:

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Example 1 :

```
int x = 5, y = 7;
int result = (x < y); // result is 1 (true)
```

Example 2: The table shows C instructions with logical expressions and their corresponding results.

INSTRUCTIONS	Result
`B = 7 > 4;`	1
`B = 5 == 10;`	0
`B = 8 <= 8;`	1
`B = (20 % 6) != 2;`	0
`B = (4*12) < pow(3,4);`	1
`B = (15 - 5) >= (2*5 - 3);`	1

3.5.3. Logical operators

They are used to combine conditions:

- `&&` → logical AND
- `||` → logical OR
- `!` → logical NOT

Example :

```
int a = 1, b = 0;
int test = (a && !b); // test is 1 (true)
```

3.5.4. Operator Precedence

As in algorithms, some operators have priority:

- The arithmetic operators `* / %` are evaluated before `+ -`.
- Parentheses `()` can be used to force an order of evaluation.

Example :

```
int result = 2 + 3 * 4;     // result = 14
int correct = (2 + 3) * 4; // correct = 20
```

3.6. Input / Output operations

In C, input and output operations allow the program to communicate with the user. These concepts correspond directly to the Read and Write instructions we studied in algorithms, but with the syntax specific to the C language.

3.6.1. Displaying with printf ()

The `printf()` function allows you to display text, messages, and variable values on the screen.
Syntax:

```
printf ("Text… ", var, cons, expr, …) ;
```

The first part inside the double quote corresponds to the **formatted text**, which may include the text to be displayed, format specifiers, and escape characters.
The second part consists of variables, constants, or expressions whose values are to be displayed on the screen.

Examples :

```
printf ("Algeria") ; //Means display on the screen the following message:
Algeria
printf ("A = %d", A) ;  //Means display on the screen A= followed by the
content of the variable A.
printf ("The coordinates are : %f, %f ", X, Y) ; //Means display on the screen
the following message : The coordinates are : followed by the contents of the
variables X and Y.
```

Notes:

- `%d` is a format specifier for integers.
- `\n` adds a new line.

You can also display multiple values or combined messages:

```
int a = 5, b = 7;
printf("a = %d, b = %d\n", a, b);
```

1. Escape Characters

Escape characters allow controlling the layout of the output, such as inserting a new line or a tab, without displaying additional text:

The escape characters	Type
"\n"	New line (Line break)
"\t"	Horizontal tab
"\'"	Display a single quote (apostrophe)
"\n""	Display a double quote
"\\"	Display a backslash
"%%"	Display a percent sign (%)

2. Format Specifiers

Format specifiers indicate the type of the values (variables, constants, or expressions) to be displayed. Each format specifier must correspond to the type of the associated value :

Escape Characters	Type
"%d"	int
"%ld"	long
"%f"	float or double
"%c"	char

3.6.2. Reading with scanf()

The `scanf()` function allows you to read a value entered by the user and store it in a variable.

Syntax:

```
scanf ("Format_1, … ", &var1, …) ;
```

The first part inside the double quote corresponds to the **format specifiers**, which define the type and number of values to be read.
The second part corresponds to the **address operator (&)**, which specifies the memory location where the input value will be stored, **followed by the variable name**, which identifies the variable that will receive the entered value.

Examples:

```
scanf (   "%d ",&x  ); //To ask for the value of x (x is an integer variable)
scanf (   "%c ",&y  );  //To ask for the value of y (y is a char variable)
```

```
scanf ("%d %d %d " , &day, &month, &year) ; //To ask for the day, month,
and year
```

Note: During the execution of the `scanf()` function, the computer waits for the user to provide a value in order to continue executing of the program.

Example: `printf () – scanf ()`
We want to write a program that calculates the area of a disk.
1. Provide the declaration instructions.
2. Provide the instructions that ask the user to enter the data values.
3. Provide the processing instructions.
4. Provide the instructions that display the result.

```
#include <stdio.h>
int main() {
    const float PI = 3.14159;  // Declaration of the constant PI
    float R;                // Declaration of the variable for the radius
    float A;                  // Declaration of the variable for the area
    // Ask the user to enter the radius
    printf("Enter the radius of the disk: ");
    scanf("%f", &R);
    // Calculate the area
    A = PI * R * R;
   // Display the result
    printf("The area of the disk is: %.2f\n", A);
    return 0;
}
```

Notes:

- In a format specifier such as %.2f, the value .2 indicates the number of digits displayed after the decimal point. For example, %.2f displays a real number with two digits after the decimal point.
- To improve program readability and make it easier for humans to understand, comments can be added in C using double slashes // for single-line comments or /* ... */ for multi-line comments. They are not executed by the computer; they serve only as notes for the reader.

3.7. Control Structures in C

In C, control structures allow you to direct the flow of execution of a program, exactly as we studied in algorithmics. The logic remains the same; only the syntax changes.

3.7.1. Conditional Structures

Conditional statements allow the execution of a block of instructions only if a condition is true.
In this section, you will understand the role and use of conditional structures. More specifically, we will study four types of conditional structures:

- The Simple conditional structure with a single choice (**if simple**),
- The Alternative conditional structure (two choices **if–else**),
- The Nested alternative structure (multi-choice using **if – else if – else if – else).**
- The Switch conditional structure with multiple cases (Equivalent to a multiple-choice structure).

3.7.1.1. Simple conditional structure

Syntax:

```
if   Condition_is_true
      Statements;
```

Example: Using the conditional structure, write an algorithm that divides two integers.

```
#include <stdio.h>
int main() {
    int A, B;
    double R;
      printf("Enter the numerator: ");
      scanf("%d", &A);
      printf("Enter the denominator: ");
      scanf("%d", &B);
      if (B != 0) {
                R = (double)A / B;
                printf("The result of the division is: %.2f\n", R);
       }
return 0;
}
```

Note :

- In C syntax, if there are multiple instructions to execute within an if or else block, you must enclose them in { },
- In the statement `R = (double)A / B, (double)` is a **cast** that converts the integer A to a floating-point number before division, so that the result R is real and not truncated. This operation is called **casting**.

3.7.1.2. Alternative conditional structure

Syntax:

```
if   Condition_is_true
      Statement_1;
else
       Statement_2;
```

Example: Using the alternative structure, write a program that divides two integers.

```
#include <stdio.h>
int main() {
    int A, B;
    double R;
      printf("Enter the numerator: ");
      scanf("%d", &A);
      printf("Enter the denominator: ");
       scanf("%d", &B);
            if (B != 0) {
                        R = (double)A / B;
                        printf("The result of the division is: %.2f\n", R);
             } else {
                       printf("Error: Division by zero is not allowed.\n");
                 }
  return 0;
}
```

3.7.1.3. Nested alternative structure

Syntax:

```
if (Condition1) {
    Statement_1;
} else {
    if (Condition2) {
        Statement_2;
    } else {
      if (Condition3) {
        Statement_3;
    } else {
        Statement_4 ;
    }
}
```

Example: Write an algorithm that reads a number and displays whether it is positive, negative or zero

```
#include <stdio.h>
int main() {
    float num;
      printf("Enter a number: ");
      scanf("%f", &num);
      if (num > 0) {
                  printf("The number is positive.\n");
       } else
            if (num < 0) {
                        printf("The number is negative.\n");
```

```
            } else {
                        printf("The number is zero\n");
                    }
    return 0;
}
```

3.7.1.4. Switch structure

Syntax:

```
switch (expression) {
    case Value_1:   Statement_1; break;
    case Value_2:   Statement_1; break;
     ..     .. :   ..
    case Value_n:   Statement_n; break;
    default:        OtherStatement;
}
```

Note:

- break : Allows you to exit the **switch** after executing the corresponding case.
  Without break, execution continues into the following cases (fall-through).
- default: Optional.
  Executes if none of the cases match the expression.

Example: Using Switch structure, write a program in C language that allows you to enter a number between 1 and 5 and translate this number into a letter

```
#include <stdio.h>
int main() {
    int number;
    printf("Enter a number between 1 and 5: ");
    scanf("%d", &number);
    switch(number) {
        case 1: printf("One\n"); break;
        case 2: printf("Two\n"); break;
        case 3: printf("Three\n"); break;
        case 4: printf("Four\n"); break;
        case 5: printf("Five\n"); break;
        default: printf("Invalid number, Please enter a number between 1 and
5 \n");
    }
    return 0;
}
```

3.7.2. Repetitive Structures

In this section, you will understand the role and use of repetitive structures, also called loops, which allow the repeated execution of a block of instructions. More specifically, we will study three types of loops:

- The **for** loop, which is used when the number of repetitions is known in advance
- The **while** and **do…while** loops, which are used when the number of repetitions depends on a condition.

The same examples used in algorithmics can be translated directly into C.

3.7.2.1. FOR loop

Syntax:

```
for (initialization; condition; increment) {
Statements
}
```

- **Initialization:** This instruction is executed before the first iteration of the loop.
- **Condition:** As long as this condition is true, the FOR loop continues to execute.
- **Increment:** This instruction is executed at the end of each iteration to update the loop counter.

Example 1: This example displays the numbers from 1 to 5 using a FOR loop.

```
#include <stdio.h>

int main() {
    int i;
    for (i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Example 2: Write an algorithm that display even numbers from 1 to N.

```
#include <stdio.h>

int main() {
    int i, N;

    printf("Enter N: ");
    scanf("%d", &N);

    for (i = 2; i <=N ; i += 2) {
        printf("%d\n", i);
    }
    return 0;
}
```

3.7.2.2.WHILE loop

Syntax:

```
while(condition) {
Statements;
}
```

Example: Ask the user to enter a positive number. Repeat until the user enters a number greater than 0.

```
#include <stdio.h>
int main() {
    int number = 0;
    while (number <= 0) {
        printf("Enter a positive number: ");
        scanf("%d", &number);
    }
    return 0;
}
```

Explanation :

- The loop does not know in advance how many times it will run,
- The condition `number <= 0` is tested before each iteration,
- If the user enters a positive number at the first try, the loop will not repeat,
- This is a typical use case of a **while** loop**,** where the number of repetitions depends on user input.

Note: The condition is tested before each iteration; therefore, the loop may not be executed at all if the condition is false from the beginning.

3.7.2.3. DO…WHILE loop

Syntax*:*

```
do{
Statements;
}while(condition);
```

Example: Ask the user to enter a number greater than 10. Repeat until the user enters a valid number.

```
#include <stdio.h>
int main() {
    int number;
    do {
        printf("Enter a number greater than 10: ");
        scanf("%d", &number);
    } while (number <= 10);
    return 0;
}
```

Explanation:

- The block inside do { … } is executed at least once, even if the first input is valid,
- After each iteration, the condition number <= 10 is checked,
- If the condition is true, the loop repeats; if false, the loop stops.

3.7.3. Sequence breaks

In some situations, it is necessary to stop the execution of a loop even when its repetition condition remains true. This is achieved by using execution-control instructions.

These instructions are particularly useful when several criteria must be evaluated to decide whether a block of instructions should continue to run.

In the C programming language, there are four main instructions that allow the programmer to control or interrupt the normal flow of a program, depending on the required level of intervention:

- break: immediately terminates the loop,
- continue: skips the current iteration and moves directly to the next one,
- goto: performs a jump to a labeled statement,
- return: ends the execution of a function and optionally returns a value

Two standard library functions can also be used to change the order of program execution:

- **exit (int status):** immediately terminates the execution of the program.
- **longjmp(jmp_buf env, int val):** allows returning to a previously defined restart point in the program.

1. break: Allows you to immediately exit a loop (for, while, or do…while), even if the loop condition is not yet false. Its purpose is to stop the loop when a certain condition is met before the normal end.

Example :

```
#include <stdio.h>

int main() {
    int i;

    for(i = 1; i <= 10; i++) {
        if(i == 5) {
            break;
        }
        printf("%d ", i);
    }

    return 0;
}
```

Explanation:
The break statement immediately terminates the loop when i becomes equal to 5. Therefore, the numbers displayed are: **1, 2, 3, 4**.

2. continue: Allows you to immediately move to the next iteration of the loop, skipping the rest of the block for the current iteration. Its purpose is to ignore certain values or conditions without stopping the entire loop.

Example :

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;  // skip the rest of this iteration when i equals 3
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Explanation: The loop ignores 3 and displays: 1, 2, 4, 5.

3. goto: Allows you to jump directly to a label in the code. Its use is sometimes for exiting complex nested loops or handling errors, but there are often better alternatives (such as **break**).

Example :

```
#include <stdio.h>
int main() {
    int i = 1;
    while (i <= 5) {
        if (i == 4) {
            goto end_loop; // Stop the loop when i equals 4
        }
        printf("%d\n", i);
        i++;
    }
end_loop:
    printf("Loop exited using goto.\n");
    return 0;
}
```

Explanation:

- The while loop starts with i = 1.
- The loop continues as long as i <= 5.

- As soon as i == 4, the statement goto end_loop; jumps directly to the label end_loop:, and the loop stops.

4. return: ends the execution of the current function and transfers control back to the function that called it.

Example :

```
#include <stdio.h>
int main() {
    printf("Start\n");
    return 0;  // Ends the program
    printf("This line will not be executed\n");
}
```

Explanation:

- The program prints Start,
- When return 0; is executed, the program ends immediately,
- Any code after return is ignored.

3.8. Exercises

Exercise 3.1

Write a program in C language that calculates and displays the area of a triangle for which you must enter the lengths of the three sides.
Use the formula : $Area^2 = P(P-A)(P-B)(P-C)$ where A, B, C are the lengths of the three sides and P is the half-perimeter of the triangle (knowing that the perimeter = A+ B+C).

Exercise 3.2

We want to write a program in C language that converts a temperature given in degrees Celsius to a temperature in degrees Fahrenheit using the formula: $F=\frac{180}{100} \times C + 32$
We assume that the Celsius temperature is an integer (type int).
Write a program that declares a variable C of type int and a variable F of type float, then calculates and displays the corresponding value of F.

Exercise 3.3

Write a program in C language titled *Item_Billing*. The program allows the user to enter the number of items and their unit price before tax. The VAT (Value Added Tax) rate will always be 20.6%. If the total amount exceeds 1000 DA, a discount of 10% will be applied. The program should display the dialogue as follows:
Number of items: …
Unit price: …
Total amount: …

Discount: …
Net amount to pay: …

Exercise 3.4

Write a program in C language that allows the user to enter from the keyboard the sex (M/F), weight P, height T, and age A, and calculates and displays:

- The BMI (Body Mass Index), the obesity indicator, calculated as: $\frac{\mathrm{P}}{\mathrm{T}^2}$
- The ideal weight (IW) of an individual using the formula:
  - For men:

$$\boldsymbol{IW = \frac{(3.T - 250).(A + 270)}{1200}}$$

  - For women:

$$IW = \frac{\left(\frac{T}{2} - 30\right).(A + 180)}{200}$$

- The program should then display, according to the BMI, one of the following messages:
  - Less than 18.5: "You are underweight; your ideal weight is …"
  - 18.5 to 25: "You have a normal body weight; your ideal weight is …"
  - 25 to 30: "You are overweight; your ideal weight is …"
  - 30 to 35: "You have moderate obesity; your ideal weight is …"
  - 35 to 40: "You have severe obesity; your ideal weight is …"
  - More than 40: "You have morbid or massive obesity; your ideal weight is …"

Exercise 3.5

Write a C program to convert a given integer (in seconds) to day, hours, minutes and seconds.

Exercise 3.6

Write a program in C language titled "SEASON" that reads the number of a month and displays the corresponding season.
We consider that:

- December, January, and February correspond to Winter.
- March, April, and May correspond to Spring.
- June, July, and August correspond to Summer.
- September, October, and November correspond to Autumn.

Example: If the month number is 7, the program displays: Summer.

Exercise 3.7

Using a multiple-choice statement (switch / case), write a program in C language that allows the user to enter two real numbers and an arithmetic operation, namely:

- * : Perform a multiplication

- + : Perform an addition
- - : Perform a subtraction
- / : Perform a division

Then, based on the user's choice, display a message showing the operation and the result, respecting the following dialogue format:
A + B = …
A - B = …
A *x* B = …
A / B = …

Exercise 3.8

Write a program in C language that displays the solution(s) of a quadratic equation of the form $ax^2 + bx + c$.
Note: Use the functions pow () and sqrt() to calculate the power and the square root.

Exercise 3.9

Write a program in C language that implements the following game for two players:

1. Player 1 enters a secret integer.
2. Player 2 tries to guess the number.
3. Player 2 has unlimited attempts.
4. After each guess, the program indicates whether the secret number is higher or lower than the guessed number.
5. When Player 2 guesses the correct number, display a congratulatory message along with the number of attempts taken (the score).

Exercise 3.10

Write a program in C language that asks the user to enter N numbers, one at a time. At the end, the program should:

1. Display the largest number among the ones entered.
2. Show the position of that number in the sequence of inputs.

Example:
Enter number 1: 13
Enter number 2: 17
...
Enter number 10: 5
The largest number is: 17
It was the 2nd number entered

Note: Use loop structures such as **For**, **While**, or **Repeat** to read the numbers and determine the maximum.

Exercise 3.11

Write a program in C language that reads a sequence of numbers ending by 0 and calculates and displays how many even and how many odd numbers there are.

Exercise 3.12

Consider the following algorithm:

```
   Algorithm Sum_of_Proper_Divisors
   Variables
       N, Sn, i : integer
   Begin
       Write("Enter N: ")
       Read(N)
       Sn ← 0
       For i ← 1 To (N div 2) Do
           If N mod i = 0 Then
               Sn ← Sn + i
           EndIf
       EndFor
       Write ("The sum is: ", Sn)
End
```

- Execute the algorithm with N = 6 then N = 8
- Determine what this algorithm do.
- Translate into C language.
- Rewrite the program using a While loop instead of the For loop.

General Conclusion

This handout has provided a structured overview of the fundamental concepts in computer science and programming, in line with the educational objectives of the module. Through the progressive study of theoretical concepts, practical exercises, and learning the C language, students have acquired both a logical and practical understanding of programming, as well as methodological skills for designing and developing programs.

The course ensures an effective transition from algorithmic theory to practical implementation on a computer, thus preparing students for both academic and professional applications. Mastery of the foundational concepts presented provides a solid basis for tackling more advanced topics in computer science and developing essential technical skills in the field of science and technology.

Solutions to Exercises

Exercise 1.1

1) Convert all measurements to Bytes

Measurement	Conversion to Bytes
21 bits	$21 \div 8 \approx 2.625$ Bytes
3 Bytes	3 Bytes
8234 KB	$8234 \times 1024 = 8\,432\,896$ Bytes
5.3 MB	$5.3 \times 1024^2 = 5\,555\,968$ Bytes
66 MB	$66 \times 1024^2 = 69\,207\,168$ Bytes
7.4 GB	$7.4 \times 1024^3 = 7\,946\,327\,552$ Bytes

2) Arrange in ascending order

1. 21 bits
2. 3 Bytes
3. 5.3 MB (5 555 968 Bytes)
4. 8234 KB (8 432 896 Bytes)
5. 66 MB (69 207 168 Bytes)
6. 7.4 GB (7 946 327 552 Bytes)

Exercise 1.2

- Image dimensions: 1024×768 =786 432 pixels
- Color encoding: 3 components (Red, Green, Blue), 8 bits each

1.Total size of the image in MB:

Each pixel = 3 components × 8 bits = 24 bits

786 432 pixels ×24 bits =18 874 368 bits

Convert to Bytes (1 Byte = 8 bits): 18 874 368 ÷8=2 359 296 Bytes

Convert to MegaBytes (1 MB = 1024^2 Bytes): $2\,359\,296 \div 1024^2 \approx 2.25$ MB

Therefore, total size of the image : **2.25 MB**

2. Total number of colors:

Each component has 8 bits → $2^8 = 256$ possible values

Total number of colors=256×256×256=16 777 216

Therefore, total number of colors: **16 777 216 colors**

3. Transfer time at 100 KB/s

Convert image size to KB : 2.25 MB=2.25×1024=2304 KB

Time required: 2304/100=23.04 seconds

Therefore, Transfer time = 23.04 secon

4. Transfer time at 1 MB/s

Convert speed to KB/s: 1 MB/s=1024 KB/s

Time required: 2304 ÷ 1024=2.25 seconds

Therefore, Transfer time = 2.25 seconds

Exercise 1.3

1. $(2513)_4 \rightarrow (?)_{10}$

$(2513)_4=2\cdot4^3+5\cdot4^2+1\cdot4^1+3\cdot4^0=2\cdot64+5\cdot16+1\cdot4+3\cdot1=128+80+4+3=215$

Therefore, $(2513)_4 = (215)_{10}$

2. $(365)_8 \rightarrow (?)_{10}$

$(365)_8=3\cdot8^2+6\cdot8^1+5\cdot8^0 =3\cdot64+6\cdot8+5\cdot1=192+48+5=245$

Therefore, $(365)_8 = 245_{10}$

3. $(A9F)_{16} \rightarrow (?)_{10}$

$(A9F)_{16}=10\cdot16^2+9\cdot16^1+15\cdot16^0=10\cdot256+9\cdot16+15=2560+144+15=2719$

Therefore, $(A9F)_{16}= (2719)_{10}$

4. $(21012.21)_3 \rightarrow (?)_{10}$

$(21012.21)3=2\cdot3^4+1\cdot3^3+0\cdot3^2+1\cdot3^1+2\cdot3^0+2\cdot3^{-1}+1\cdot3^{-2}=2.81+1.27+2+0.666+0.111=194.777$

Therefore, $(21012.21)_3 = (194.777)_{10}$

5. $(0.625)_{10} \rightarrow (?)_2$

$0.625 \times 2 = 1.25 \rightarrow 1$

$0.25 \times 2 = 0.5 \rightarrow 0$

$0.5 \times 2 = 1.0 \rightarrow 1$

Therefore, $(0.625)_{10} = (0.101)_2$

6. $(45.375)_{10} \rightarrow (?)_2$

Integer part: 45 → divide by 2:

45 ÷ 2 = 22 r **1**

22 ÷ 2 = 11 r **0**

11 ÷ 2 = 5 r **1**

5 ÷ 2 = 2 r **1**

2 ÷ 2 = 1 r **0**

1 ÷ 2 = 0 r **1**

Integer part = **101101**

Fractional part: 0.375

$0.375 \times 2 = 0.75 \rightarrow$ **0**

$0.75 \times 2 = 1.5 \rightarrow$ **1**

$0.5 \times 2 = 1.0 \rightarrow$ **1**

Fractional part = .011

Therefore, $(45.375)_{10} = (101101.011)_2$

7. $(110011.101)_2 \rightarrow (?)_{10}$
Integer part: 110011.101= $1.2^5+1.2^4+0.2^3+0.2^2+1.2^1+1.2^0+1.2^{-1}+0.2^{-2}+1.2^{-3}$== 32+16+0+0+2+1+ 0.5 + 0 + 0.125 = 51.625
Therefore, $(110011.101)_2 = (51.625)_{10}$

8. $(58)_{10} \rightarrow (?)_8$
Divide by 8:
58 ÷ 8 = 7 r **2**
7 ÷ 8 = 0 r **7**
Therefore, $(58)_{10} = (72)_8$

9. $(93)_{10} \rightarrow (?)_{16}$
Divide by 16:
93 ÷ 16 = 5 r **13 → D**
5 ÷ 16 = 0 r **5**
Therefore, $(93)_{10} = (5D)_{16}$

10. $(146.75)_{10} \rightarrow (?)_2$
Integer part: 146 → divide by 2:
146 ÷ 2 = 73 r **0**
73 ÷ 2 = 36 r **1**
36 ÷ 2 = 18 r **0**
18 ÷ 2 = 9 r **0**
9 ÷ 2 = 4 r **1**
4 ÷ 2 = 2 r **0**
2 ÷ 2 = 1 r **0**
1 ÷ 2 = 0 r **1**
Integer part = 10010010
Fractional part: 0.75
0.75 × 2 = 1.5 → **1**
0.5 × 2 = 1.0 → **1**
Fractional part = .11
Therefore, $(146.75)_{10} = (10010010.11)_2$

Exercise 1.4

1. $(254)_6 = (?)_2$

Step 1: Convert from base 6 to base 10
$(254)_6 = 2\times6^2+5\times6^1+4\times6^0 = 72+30+4 = (106)_{10}$
Step 2: Convert from base 10 to base 2
106÷2=53, r **0**
53÷2=26, r **1**

26÷2=13, r **0**
13÷2=6, r **1**
6÷2=3, r **0**
3÷2=1, r **1**
1÷2=0, r **1**
$(106)_{10}= (1101010)_2$
Therefore, $(254)_6 = (1101010)_2$

2. $(2312)_4 = (?)_2$
Step 1: Convert from base 4 to base 10
$(2312)4=2\times4^3+3\times4^2+1\times4^1+2\times4^0 =128+48+4+2=(182)_{10}$
Step 2: Convert from base 10 to base 2
182÷2=91, r **0**
91÷2=45, r **1**
45÷2=22, r **1**
22÷2=11, r **0**
11÷2=5, r **1**
5÷2=2, r **1**
2÷2=1, r **0**
1÷2=0, r **1**
$(182)_{10}=(10110110)_2$
Therefore, $(2312)_4 = (10110110)_2$

3. $(1011011101)_2=(?)_{14}$
Step 1: Convert from base 2 to base 10
$\mathbf{(1011011101)_2}=1.2^9+0.2^8+1.2^7+1.2^6+0.2^5+1.2^4+1.2^3+1.2^2+0.2^1+1.2^0=512+128+64+16+8+4+1=(733)_{10}$
Step 2: Convert from base 10 to base 14
733÷14=52, r **5**
52÷14=3, r **10**(A)
3÷14=0, r **3**
$(733)_{10}=(3A5)_{14}$
Therefore, $(1011011101)_2=(3A5)_{14}$

4. $(173.041)_9=(?)_7$

$(173.041)_9=1\times9^2+7\times9^1+3\times9^0+0\times9^{-1}+4\times9^{-2}+1\times9^{-3}=(147.0513)_{10}$
Integer part: 147→ divide by 7:
147÷7=21, r **0**
21÷7=321÷7=3, r **0**
3÷7=03÷7=0, r **3**
Integer part = 300
Fractional part: 0. 0513
0.0513×7=**0**.3591
0.3591×7=**2**.5137

0.5137×7=**3**.5959
0.5959×7=**4**.1713
0.1713×7=**1**.1991
Fractional part: 0. 02342
Therefore, $(173.041)_9 \approx (300.02341)_7$

5. $(B3C)_{16}=(?)_{11}$

Step 1: Convert from base 16 to base 10
$(B3C)_{16}$ $=11\times16^2+3\times16^1+12\times16^0=2816+48+12=(2876)_{10}$
Step 2: Convert from base 10 to base 11
2876÷11=261 remainder 5
261÷11=23 remainder 8
23÷11=2 remainder 1
2÷11=0 remainder 2
$(2876)_{10} = (2185)_{11}$
Therefore, $(B3C)_{16}=(2185)_{11}$

Exercise 1.5

1. **$(10111011101)_2=(?)_4=(?)_8=(?)_{16}$**

Base 2 → Base 4
Group binary digits by **2 bits** (from right to left):
01 01 11 01 11 01
According to the Table 1, we have:
01_2 =1, 01_2=1, 11_2=3, 01_2=1, 11_2=3, 01_2=1Therefore, $(10111011101)_2=(113131)_4$

Base 2 → Base 8
Group binary digits by **3 bits:** 010 111 011 101
101_2=5, 011_2=3, 111_2=7, 010_2=2
Therefore, $(10111011101)_2=(2735)_8$

Base 2 → Base 16
Group binary digits by **4 bits**: 0101 1101 1101
According to the Table 2, we have:
0101_2=5, 1101_2=D, 1101_2=D
Therefore, $(10111011101)_2=(5DD)_{16}$

2. $(3F9.B)_{16}=(?)_2=(?)_4=(?)_8$

Base 16 → Base 2
Each hexadecimal digit → **4 binary bits**:
3=0011, F=1111, 9=1001, B=1011
Therefore, $(3F9.B)_{16}=(1111111001.1011)_2$

Table 1

Octal System (Base 8)	**Binary System** (Base 2)		
$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Table 2

hexadecimal System (Base 16)	**Binary System** (Base 2)			
$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1
E	1	1	1	0
F	1	1	1	1

Base 2 → Base 4

Group the **binary digits in pairs** (2 bits each):

- Integer part: $001111111001_2 \Rightarrow 11\ 11\ 11\ 10\ 01$
- Fractional part: $1011_2 \Rightarrow 10\ 11$
- 11 → 3
- 11 → 3
- 11 → 3
- 10 → 2
- 01 → 1

Fractional part:

- 10 → 2
- 11 → 3

$(1111111001.1011)_2 = (33321.23)_4$

Therefore, $(3F9.B)_{16} = (33321.23)_4$

Base 2 → Base 8

Group the binary digits in triples (3 bits each):

- Integer part: $001111111001_2 \Rightarrow 001\ 111\ 111\ 001$
- Fractional part: $1011_2 \Rightarrow 101\ 100$

Convert each group to octal:

- 001 → 1
- 111 → 7
- 111 → 7
- 001 → 1

Fractional part:

- 101 → 5
- 100 → 4

$(1111111001.1011)_2 = (1771.54)_8$

Therefore, $(3F9,B)_{16} = (1771.54)_8$

3. $(57.34)_8 = (?)_2 = (?)_4 = (?)_{16}$

1. Base 8 → Base 2

Each octal digit → **3 binary bits**:

- Integer part: 5=101,7=111 → $57_8 = 101111_2$
- Fractional part: 3=011,4=100

Therefore, $(57.34)_8 = (101111.011100)_2$

2. Base 2 → Base 4

Group **binary digits in pairs** (2 bits each):

- Integer part: $101111_2 \Rightarrow 10\ 11\ 11 \rightarrow 233_4$
- Fractional part: $011100_2 \Rightarrow 01\ 11\ 00 \rightarrow 0.130_4$

$(101111.011100)_2 = (233.130)_4$
Therefore, $57.34)_8 = (233.130)_4$

3. Base 2 → Base 16
Group binary digits in 4 bits:

- Integer part: $101111_2 \Rightarrow 0010\ 1111 \rightarrow 2F$
- Fractional part: $011100_2 \Rightarrow 0111\ 0000 \rightarrow 70$

$(101111.011100)_2 = (2F.70)_{16}$
Therefore, $(57.34)_8 = (2F.70)_{16}$

4. $(2101021)_3 = (?)_9$

1. *Group the ternary digits in **pairs of 2 from right to left**:*
   02 10 10 21
2. Replace each pair with the **value from the Base 9 column**:
   - 02 → 2
   - 10 → 3
   - 10 → 2
   - 21 → 7

Therefore, $(2101021)_3 = (2337)_9$

Base 9	**3^1**	**3^0**
0	0	0
1	0	1
2	0	2
3	1	0
4	1	1
5	1	2
6	2	0
7	2	1
8	2	2

Exercise 1.6

1. Determine Y in base b
A number expressed in decimal as:
$Y = 5b^6 + 3b^4 + 2b^2 + 4b + 7$

We can rewrite it in base **b** by using the coefficients of powers of bb:

- The coefficient of b^6 is **5**
- The coefficient of b^5 is **0**
- The coefficient of b^4 is **3**
- The coefficient of b^3 is **0**
- The coefficient of b^2 is **2**
- The coefficient of b^1 is **4**
- The coefficient of b^0 is **7**

So, in **base b**, the number is: $Y = (5030247)_b$

2. Determine the base n knowing $(143)_n = (99)_{10}$

The general formula for converting from base nn to decimal:
$(143)n = 1 \cdot n^2 + 4 \cdot n + 3$
Set equal to decimal 99:

$1 \cdot n^2+4 \cdot n+3=99$
$n^2+4n+3=99$
$n^2+4n-96=0$
Solve the quadratic equation $n^2+4n-96=0$ using the discriminant:
$\Delta=4^2-4 \cdot 1 \cdot (-96)=16+384=400$
Two solutions:
$n1=-4+202=162=8$
$n2=-4-202=-242=-12$(not valid)
So, the **base is n=8**

3. Determine the two-digit integers (xy) in base 7 that are written as (yx) in base 10
Let the number in **base 7** be $(xy)_7$. Its decimal value is:
$(xy)_7=x \cdot 7+y$
We are told that in decimal, it is written as $(yx)_{10}=10y+x$
So the equation is:
$7x+y=10y+x$
$7x-x=10y-y$
$6x=9y$
$2x=3y$
$y=(2/3)x$
Since x and y are digits in base 7:

- $x=1,2,3,4,5,6x=1,2,3,4,5,6$
- $y=0,1,2,3,4,5,6y=0,1,2,3,4,5,6$

The fraction $y=2x/3y=2x/3$ must be integer. Try possible x:

- $x=1 \Longrightarrow y=2/3$ **invalid**
- $x=2 \Longrightarrow y=4/3x=2 \Longrightarrow y=4/3$ **invalid**
- $x=3 \Longrightarrow y=2x=3 \Longrightarrow y=2$ **valid**
- $x=4 \Longrightarrow y=8/3x=4 \Longrightarrow y=8/3$ **invalid**
- $x=5 \Longrightarrow y=10/3x=5 \Longrightarrow y=10/3$ **invalid**
- $x=6 \Longrightarrow y=4x=6 \Longrightarrow y=4$ **valid**

Other values of x give non-integer y.

So the solutions are: **$(xy)_7=(32)_7$ and $(64)_7$**

Exercise 1.7

1. $(11101)_2 + (10011)_2$

```
 11101
+10011
-------
110000
```

Therefore, $(11101)_2 + (10011)_2=(110000)_2$

2. $(1101.101)_2 + (101.011)_2$:

```
 1101.101
+ 101.011
------------
10011.000
```

Therefore, $(1101.101)_2 + (101.011)_2 = (10011.000)_2$

3. $(101.1101)_2 + (11.101)_2$

```
 101.1101
+ 11.1010
------------
1001.0111
```

Therefore, $(101.1101)_2 + (11.101)_2 = (1001.0111)_2$

4. $(1101.01)_2 + (101.111)_2 + (11.101)_2$

```
 1101.01
+ 101.111
---------
10011.001
+  11.101
----------
10110.110
```

Therefore, $(1101.01)_2 + (101.111)_2 + (11.101)_2 = (10110.110)_2$

5. $(11100.11)_2 - (101.01)_2$

```
11100.11
- 101.01
--------
 10111.10
```

Therefore, $(11100.11)_2 - (101.01)_2 = (10111.10)_2$

6. $(1011.01)_2 \times (110)_2$

```
        101101
×          110
----------------
        000000    ← 101101 × 0
+      101101.    ← 101101 × 1 (left shift)
      101101..    ← 101101 × 1 (one more shift)
----------------
     100001110
```

We had removed 2 fractional bits, so we place the decimal point back two positions from the right: $100001110 \Longrightarrow 1000011.10$

Therefore, $(1011.01)_2 \times (110)_2 = (1000011.1)_2$

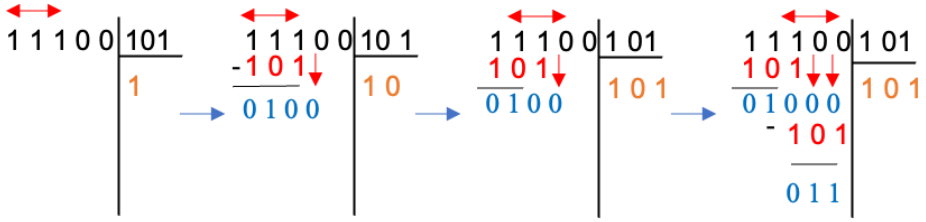
7. $(10.111)_2 \times (1011)_2$

```
          10111
        ×  1011
      ----------------
          10111        ← 10111 × 1
    +    10111.        ← 10111 × 1   (shift)
         00000..       ← 10111 × 0
        10111...       ← 10111 × 1   (shift)
      ----------------
        11111101
```

We had 3 fractional bits, so we place the decimal point back three positions from the right:
$11111101 \Longrightarrow 11111.101_2$

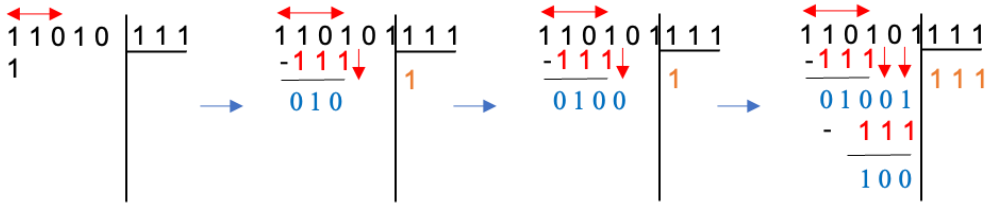
Therefore, $(10.111)_2 \times (1011)_2 = (11111.101)_2$

8. $(11100)_2 / (101)_2$



Therefore: $(11100)_2/(101)_2=(101)_2$ remainder $(11)_2$

9. $(110101)_2 /(111)_2$



Therefore: $(110101)_2 / (111)_2 = (111)_2$ remainder $(100)_2$

10. $(542)_8 + (317)_8$

$542_8=101\ 100\ 010_2$

$317_8=011\ 001\ 111_2$

```
      101100010
   +  011001111
    ------------
     1000110001
```

$1000110001_2 \Rightarrow 001\ 000\ 110\ 001 \Rightarrow 1061_8$

Therefore: $(542)_8 + (317)_8 = (1061)_8$

11. $(46.3)_8 \times (5.2)_8$

$46.3_8 = 100110.011_2$

$5.2_8 = 101.010_2$

```
      100110011
×        101010
----------------
      000000000  ← bit 0 du multiplicateur = 0
     100110011   ← bit 1 =1, décalé d'un bit à gauche
    000000000    ← bit 2 =0
   100110011     ← bit 3 =1, décalé de 3 bits
  000000000      ← bit 4 =0
 100110011       ← bit 5 =1, décalé de 5 bits
----------------
11001001011110
```

100110.011 → 3 fractional bits

101.010 → 3 fractional bits

Total fractional bits = 6 → binary point 6 bits from the right

Final result with binary point:

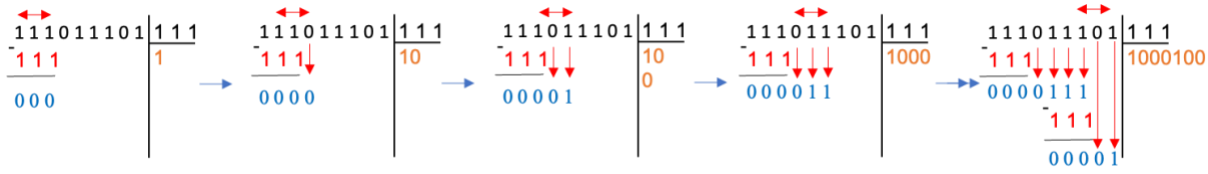
$11001001.011110_2 \Rightarrow 011\ 001\ 001.011\ 110_2 \Rightarrow 311.36_8$

Therefore: $(46.3)_8 \times (5.2)_8 = (311.36)_8$

12. $(735)_8 / (7)_8$

$735_8 = 111011101_2$

$7_8 = 111_2$



$1000100_2 \Rightarrow 1\ 000\ 100 \Rightarrow 104_8$

Therefore: $(735)_8 / (7)_8 = (104)_8$ remainder $(1)_8$

13. $(2B7)_{16} + (C4F)_{16}$

$2B7_{16} = 001010110111_2$

$C4F_{16} = 110001001111_2$

```
 001010110111
+110001001111
-------------
 111100000110
```

$111100000110_2 \Longrightarrow 1111\ 0000\ 0110 \Longrightarrow F06_{16}$

Therefore: $(2B7)_{16} + (C4F)_{16} = (F06)_{16}$

14. $(A3F)_{16} - (56C)_{16}$

$A3F_{16}=101000111111_2$

$56C_{16}=010101101100_2$

```
   101000111111
 - 010101101100
  ---------------
   010011010011
```

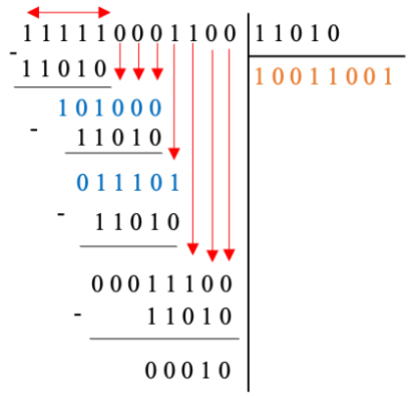
$010011010011_2 \Longrightarrow 0100\ 1101\ 0011 \Longrightarrow 4D3_{16}$

Therefore: $(A3F)_{16} - (56C)_{16} = (4D3)_{16}$

15. $(F8C)_{16} / (1A)_{16}$

$F8C_{16}=111110001100_2$

$1A_{16}=11010_2$



$10011001_2 \Longrightarrow 1001\ 1001 \Longrightarrow 99_{16}$

Therefore: $(F8C)_{16} / (1A)_{16} = (99)_{16}$ remainder $(2)_{16}$

Exercise 2.1

Algorithm Test1

Statement	A	B	C	D
A ← 17	17	–	–	–
B ← 5	17	5	–	–
C ← A + B	17	5	22	–
A ← A – B	12	5	22	–
B ← B * 2	12	10	22	–
C ← C div 3	12	10	7	–
A ← A mod B	2	10	7	–
D ← (A>40) OR (C<B)	2	10	7	True

Algorithm Test2

Statement	A	B	C
A ← True	True	–	–
B ← A	True	True	–
A ← False	False	True	–
B ← not A	False	True	–
C ← A and B	False	True	False
C ← not (A or B)	False	True	False

Algorithm Test3

Statement	A	B	C	D	E	Q
A ← 7	7	–	–	–	–	–
D ← 'k'	7	–	–	k	–	–
B ← A + 1	7	8	–	k	–	–
C ← B / 2	7	8	4.0	k	–	–
C ← C – 2	7	8	2.0	k	–	–
E ← 's'	7	8	2.0	k	s	–
A ← B	8	8	2.0	k	s	–
Q ← D > E	8	8	2.0	k	s	False

Algorithm Test4

Statement	A	B	C
A ← '5'	5	–	–
B ← A + 'a'	5	5a	–
C ← B + A	5	5a	5a5

Will C := A + B give the same result?

No, it will give '55a' instead of '5a5', because string concatenation depends on the order.

Exercise 2.2

Statement	Valid / Invalid	Explanation
`A = 15;`	Valid	Integer assigned to integer → correct.
`B = 'K';`	Valid	Character assigned to character → correct.
`C = 7 > 10;`	Valid	Boolean expression → evaluates to False.
`A = 'Z';`	Invalid	Character cannot be assigned to integer.
`B = 3;`	Invalid	Integer cannot be assigned to character.
`C = 0;`	Invalid	Boolean cannot store an integer.
`A = A + 8;`	Valid	Arithmetic on integers → valid.
`B = B + 'M';`	Valid	Character concatenation → valid
`C = not (A > 20);`	Valid	Logical negation of a Boolean expression → valid.
`A = B mod 2;`	Invalid	Modulo cannot be applied to a character.
`C = A + 2;`	Invalid	Arithmetic cannot be assigned to Boolean.
`B = 'Q';`	Valid	Character assigned to character → valid.
`A = C;`	Invalid	Boolean cannot be assigned to integer.
`C = (B > 'A') and (A < 20);`	Valid	Logical expression with comparison → Boolean assignment valid.
`B = B + 5;`	Invalid	Cannot add integer to a character directly.

Exercise 2.3

```
Algorithm Calculate_Average
Variables
    mark1, mark2, mark3 : real
    Sum, AVG : real
Begin
    write("Enter the first mark: ")
    read(mark1)
    write ("Enter the second mark: ")
    read (mark2)
    write ("Enter the third mark: ")
    read (mark3)
    Sum ← mark1 + mark2 + mark3
    AVG ← sum / 3
    write ("Sum = ", Sum)
    write ("Average = ", AVG)
End
```

2.Run the algorithm for the following marks: 10, 15, 9.

Memory box:

N1	10
N2	15
N2	09
Sum	34
AVG	11.33

Screen

```
Enter the first mark:
10
Enter the second mark:
15
Enter the third mark:
09
Sum = 34
Average = 11.33
```

Exercise 2.4

```
Algorithm Student
Variables
   name : string
   age  : integer
Begin
   write ("Enter your name: ")
   read (name)
   write ("Enter your age: ")
   read (age)
   write ("Hello ", name, ", you are ", age,
         " years old and welcome to the University of Mostaganem.")
End
```

Exercice 2.5

```
Algorithm Operations
Variables
    A, B : integer
    sum, pro, diff : integer
    di : real
    int_div, r: integer
Begin
    Write("Enter the first integer: ")
    Read(A)
    Write("Enter the second integer: ")
    Read(B)
    sum ← A + B
    pro ← A * B
    diff ← A - B
    Write("Sum = ", sum)
    Write("Pro = ", pro)
    Write("Difference = ", diff)
    If B <> 0 Then
        di ← A / B
        int_div ← A div B
        r ← A mod B
```

```
        Write("Division = ", di)
        Write("Integer Division = ", int_div)
        Write("Remainder = ", r)
    Else
        Write("Division by zero is not allowed")
    EndIf
End
```

Exercise 3.6

1. Alternative structure

```
Algorithm Find_Maximum
Variables
    num1, num2 : integer
Begin
    Write("Enter the first number: ")
    Read(num1)
    Write("Enter the second number: ")
    Read(num2)
    If num1 > num2 Then
        Write("The maximum number is: ", num1)
    Else
        Write("The maximum number is: ", num2)
    EndIf
End
```

2. Conditional structure

```
Algorithm Maximum_Number
Variables
      num1, num2 Max : real
Begin
    Write("Enter the values of A and B: ")
    Read(num1, num2)
    Max ← num1
    If Max < num2 Then
        Max ← num2
    EndIf
    Write("The maximum is: ", Max)
End
```

3.

```
Algorithm Max_With_Equality
Variables
    num1, num2 : integer
Begin
    Write("Enter the first number: ")
    Read(num1)
    Write("Enter the second number: ")
    Read(num2)
```

```
    If num1 > num2 Then
        Write("Maximum = ", num1)
    Else
        If num2 > num1 Then
            Write("Maximum = ", num2)
        Else
            Write("Both numbers are equal")
        EndIf
    EndIf
End
```

Exercise 3.7

```
Algorithm Sign_of_Product
Variables
    X, Y : real
Begin
    Write("Enter the first number (X): ")
    Read(X)
    Write("Enter the second number (Y): ")
    Read(Y)
    If X = 0 Or Y = 0 Then
        Write("The product is zero.")
    Else
        If (X > 0 And Y > 0) Or (X < 0 And Y < 0) Then
            Write("The product is positive")
        Else
            Write("The product is negative")
        EndIf
    EndIf
End
```

Exercise 3.8

```
Algorithm Vowel_or_Consonant
Variables
    letter : char
Begin
    Write("Enter a letter: ")
    Read(letter)
If letter = 'a' Or letter = 'e' Or letter = 'i' Or letter = 'o' Or letter =
'u' Or letter = 'A' Or letter = 'E' Or letter = 'I' Or letter = 'O' Or letter
= 'U' Then
        Write("The letter is a vowel.")
    Else
        Write("The letter is a consonant.")
    EndIf
End
```

Exercice 2.9

```
Algorithm Multiples_of_n
Variables
    n, m, i : integer
Begin
    Write("Enter the value of n: ")
    Read(n)
    Write("Enter the limit m: ")
    Read(m)
    For i ← 1 To m
        If i mod n = 0 Then
            Write(i)
        EndIf
    EndFor
End
```

Exercice 2.10

```
Algorithm Day_of_Week
Variables
    N : integer
Begin
    Write("Enter a number between 1 and 7: ")
    Read(N)
    Case N Of
        1 : Write("Monday")
        2 : Write("Tuesday")
        3 : Write("Wednesday")
        4 : Write("Thursday")
        5 : Write("Friday")
        6 : Write("Saturday")
        7 : Write("Sunday")
        else
            Write("Error")
    EndCase
End
```

Exercice 2.11

```
Algorithm Sum_First_10_Integers
Variables
    i, sum : integer
Begin
    sum ← 0
    For i ← 1 To 10
        sum ← sum + i
    EndFor
    Write("The sum of the first 10 positive integers is: ", sum)
End
```

Exercice 2.12

```
Algorithm Multiplication_Table
Variables
    N, i, result : integer
Begin
    Write("Enter a number: ")
    Read(N)

    Write("Table of ", N, " :")
    For i ← 1 To 10
        result ← N * i
        Write(N, " x ", i, " = ", result)
    EndFor
End
```

Exercice 2.13

```
Algorithm Power_Calculation
Variables
    x : real
    y, i : integer
    result : real
Begin
    Write("Enter a real value x: ")
    Read(x)
    Write("Enter a positive integer y: ")
    Read(y)
    result ← 1
    For i ← 1 To y
        result ← result * x
    EndFor
    Write("The result of x^y is: ", result)
End
```

Exercice 2.14

```
Algorithm Sum_of_Squares
Variables
    n, i : integer
    sum, square : integer
Begin
    Write("Enter a positive integer n: ")
    Read(n)
    sum ← 0
    For i ← 1 To n
        square ← i * i
        sum ← sum + square
```

```
        Write(i, "^2 = ", square, " → Current sum = ", sum)
    EndFor
    Write("The sum of the first ", n, " squares is: ", sum)
End
```

Exercice 2.15

```
Algorithm Countdown_Astronaut
Variables
    i : integer
Begin
    Write("Countdown from 50 to 0:")

    For i ← 50 To 0 Step -3
        Write(i)
    EndFor

    Write("Launch!")
End
```

Exercice 3.1

```
#include <stdio.h>
#include <math.h>

int main()
{
    float A, B, C;
    float P, Area2, Area;
    /* Read the three sides */
    printf("Enter side A: ");
    scanf("%f", &A);
    printf("Enter side B: ");
    scanf("%f", &B);
    printf("Enter side C: ");
    scanf("%f", &C);
    /* Calculate the half-perimeter */
    P = (A + B + C) / 2;
    /* Calculate Area² using the given formula */
    Area2 = P * (P - A) * (P - B) * (P - C);
    /* Calculate the area */
    Area = sqrt(Area2);
    /* Display the result */
    printf("The area of the triangle is: %.2f\n", Area);
    return 0;
}
```

Note :

- The real area is obtained using sqrt.

- <math.h> is required for the sqrt() function.

Exercice 3.2

```
#include <stdio.h>

int main()
{
    int C;
    float F;

    /* Read the temperature in Celsius */
    printf("Enter the temperature in Celsius: ");
    scanf("%d", &C);

    /* Convert Celsius to Fahrenheit */
    F = (180.0 / 100.0) * C + 32;

    /* Display the result */
    printf("Temperature in Fahrenheit: %.2f\n", F);

    return 0;
}
```

Note:

- C is declared as int (as required).
- F is declared as float.
- 180.0 / 100.0 is written with decimals to ensure floating-point division.
- The result is displayed with two decimal places.

Exercice 3.3

```
#include <stdio.h>

int main()
{
    int numberOfItems;
    float unitPrice, totalAmount, discount, netToPay;
    const float VAT = 20.6; // VAT rate in percent

    // Input: number of items and unit price
    printf("Number of items: ");
    scanf("%d", &numberOfItems);

    printf("Unit price: ");
    scanf("%f", &unitPrice);

    // Calculate total amount before discount
```

```
    totalAmount = numberOfItems * unitPrice;

    // Apply discount if total exceeds 1000 DA
    if (totalAmount > 1000.0)
        discount = totalAmount * 0.10; // 10% discount
    else
        discount = 0.0;

    // Net amount to pay = total after discount + VAT
    netToPay = (totalAmount - discount) * (1 + VAT / 100);

    // Display results
    printf("Total amount: %.2f DA\n", totalAmount);
    printf("Discount: %.2f DA\n", discount);
    printf("Net amount to pay: %.2f DA\n", netToPay);

    return 0;
}
```

Exercice 3.4

```
#include <stdio.h>

int main() {
    char sex;
    float P, T, BMI, IW;
    int A;

    // Input: sex, weight, height, and age
    printf("Enter your sex (M/F): ");
    scanf(" %c", &sex);  // note the space before %c to skip whitespace

    printf("Enter your weight (P in kg): ");
    scanf("%f", &P);

    printf("Enter your height (T in cm): ");
    scanf("%f", &T);

    printf("Enter your age (A in years): ");
    scanf("%d", &A);

    // Convert height from cm to meters for BMI calculation
    float T_m = T / 100.0;

    // Calculate BMI
    BMI = P / (T_m * T_m);

    // Calculate Ideal Weight (IW)
    if (sex == 'M' || sex == 'm') {
```

```
        IW = (3 * T - 250) * (A + 270) / 1200.0;
    } else if (sex == 'F' || sex == 'f') {
        IW = (T / 2 - 30) * (A + 180) / 200.0;
    } else {
        printf("Invalid sex input.\n");
        return 1; // terminate program
    }
    // Display BMI and Ideal Weight
    printf("\nBMI = %.2f\n", BMI);
    // Display message based on BMI
    if (BMI < 18.5) {
        printf("You are underweight; your ideal weight is %.2f kg.\n", IW);
    } else if (BMI >= 18.5 && BMI < 25) {
        printf("You have a normal body weight; your ideal weight is %.2f
kg.\n", IW);
    } else if (BMI >= 25 && BMI < 30) {
        printf("You are overweight; your ideal weight is %.2f kg.\n", IW);
    } else if (BMI >= 30 && BMI < 35) {
        printf("You have moderate obesity; your ideal weight is %.2f kg.\n",
IW);
    } else if (BMI >= 35 && BMI < 40) {
        printf("You have severe obesity; your ideal weight is %.2f kg.\n",
IW);
    } else { // BMI >= 40
        printf("You have morbid or massive obesity; your ideal weight is %.2f
kg.\n", IW);
    }

    return 0;
}
```

Exercice 3.5

```
#include <stdio.h>

int main() {
    // Variables
    int total_seconds;
    int days, hours, minutes, seconds;

    // Input
    printf("Enter total number of seconds: ");
    scanf("%d", &total_seconds);

    // Calcul des jours
    days = total_seconds / 86400;           // 1 day = 24*60*60 = 86400 seconds
    total_seconds = total_seconds % 86400;  // reste après les jours

    // Calcul des heures
    hours = total_seconds / 3600;           // 1 hour = 3600 seconds
```

```
    total_seconds = total_seconds % 3600;   // reste après les heures

    // Calcul des minutes
    minutes = total_seconds / 60;           // 1 minute = 60 seconds
    seconds = total_seconds % 60;           // reste en secondes

    // Affichage du résultat
    printf("Equivalent time: %d day(s) %d hour(s) %d minute(s) %d second(s)\n",
           days, hours, minutes, seconds);

    return 0;
}
```

Exercice 3.6

Using if...else

```
#include <stdio.h>

int main() {
    int month;

    // Input: month number
    printf("Enter the month number (1-12): ");
    scanf("%d", &month);

    // Determine the season using if...else
    if (month == 12 || month == 1 || month == 2) {
        printf("Winter\n");
    }
    else if (month >= 3 && month <= 5) {
        printf("Spring\n");
    }
    else if (month >= 6 && month <= 8) {
        printf("Summer\n");
    }
    else if (month >= 9 && month <= 11) {
        printf("Autumn\n");
    }
    else {
        printf("Invalid month number!\n");
    }

    return 0;
}
```

Using switch

```
#include <stdio.h>

int main() {
    int month;

    printf("Enter the month number (1-12): ");
    scanf("%d", &month);

    switch (month) {
        case 12: case 1: case 2:
            printf("Winter\n");
            break;
        case 3: case 4: case 5:
            printf("Spring\n");
            break;
        case 6: case 7: case 8:
            printf("Summer\n");
            break;
        case 9: case 10: case 11:
            printf("Autumn\n");
            break;
        default:
            printf("Invalid month number!\n");
    }

    return 0;
}
```

Exercice 3.7

```
#include <stdio.h>
int main() {
    float A, B, result;
    char op;

    // Input: two numbers
    printf("Enter the first number (A): ");
    scanf("%f", &A);

    printf("Enter the second number (B): ");
    scanf("%f", &B);

    // Input: operation
    printf("Enter the operation (+, -, *, /): ");
    scanf(" %c", &op);

    // Perform calculation based on the operation
```

```
    switch(op) {
        case '+':
            result = A + B;
            printf("%.2f + %.2f = %.2f\n", A, B, result);
            break;
        case '-':
            result = A - B;
            printf("%.2f - %.2f = %.2f\n", A, B, result);
            break;
        case '*':
            result = A * B;
            printf("%.2f x %.2f = %.2f\n", A, B, result);
            break;
        case '/':
            if (B != 0) {
                result = A / B;
                printf("%.2f / %.2f = %.2f\n", A, B, result);
            } else {
                printf("Error: Division by zero is not allowed.\n");
            }
            break;
        default:
            printf("Invalid operation!\n");
    }
    return 0;
}
```

Exercise 3.8

```
#include <stdio.h>
#include <math.h>

int main() {
    // Variables
    double a, b, c;
    double delta, n1, n2;

    // Input
    printf("Enter coefficient a: ");
    scanf("%lf", &a);
    printf("Enter coefficient b: ");
    scanf("%lf", &b);
    printf("Enter coefficient c: ");
    scanf("%lf", &c);

    // Check that a is not zero
    if (a == 0) {
        printf("This is not a quadratic equation.\n");
        return 0;
```

```
    }

    // Calculate the discriminant (delta)
    delta = pow(b, 2) - 4*a*c;

    if (delta > 0) {
        // Two distinct real solutions
        n1 = (-b + sqrt(delta)) / (2*a);
        n2 = (-b - sqrt(delta)) / (2*a);
        printf("Two real solutions: n1 = %.2lf, n2 = %.2lf\n", n1, n2);
    }
    else if (delta == 0) {
        // One real double solution
        n1 = -b / (2*a);
        printf("One real solution: n = %.2lf\n", n1);
    }
    else {
        // Discriminant is negative → no real solution
        printf("No real solutions exist (delta < 0).\n");
    }

    return 0;
}
```

Exercise 3.9

```
#include <stdio.h>

int main() {
    int secretNumber, guess;
    int attempts = 0;

    // Player 1 enters the secret number
    printf("Player 1, enter the secret number: ");
    scanf("%d", &secretNumber);

    printf("Player 2, try to guess the number!\n");

    // Player 2 guesses until correct
    do {
        printf("Enter your guess: ");
        scanf("%d", &guess);
        attempts++;

        if (guess < secretNumber) {
            printf("The secret number is higher!\n");
        } else if (guess > secretNumber) {
            printf("The secret number is lower!\n");
        } else {
```

```
            printf("Congratulations! You guessed the number %d in %d
attempts.\n", secretNumber, attempts);
        }

    } while (guess != secretNumber);

    return 0;
}
```

Exercise 3.10

1. For loop

```
#include <stdio.h>

int main() {
    int N, i, position;
    double number, largest;

    // Ask the user how many numbers they will enter
    printf("Enter the number of values (N): ");
    scanf("%d", &N);

    if (N <= 0) {
        printf("The number of values must be positive.\n");
        return 0;
    }

    // Initialize largest and position
    printf("Enter number 1: ");
    scanf("%lf", &largest);
    position = 1;

    // Loop through the rest of the numbers
    for (i = 2; i <= N; i++) {
        printf("Enter number %d: ", i);
        scanf("%lf", &number);

        if (number > largest) {
            largest = number;
            position = i; // store the position of the largest number
        }
    }

    // Display results
    printf("The largest number is: %.2lf\n", largest);
    printf("Its position in the sequence is: %d\n", position);

    return 0;
}
```

2. While loop :

```
#include <stdio.h>

int main() {
    int N, i = 1, position = 1;
    double number, largest;

    // Ask for the number of values
    printf("Enter the number of values (N): ");
    scanf("%d", &N);

    if (N <= 0) {
        printf("The number of values must be positive.\n");
        return 0;
    }

    // Read the first number
    printf("Enter number 1: ");
    scanf("%lf", &largest);

    // Read the remaining numbers
    i = 2;
    while (i <= N) {
        printf("Enter number %d: ", i);
        scanf("%lf", &number);

        if (number > largest) {
            largest = number;
            position = i;
        }

        i++;
    }

    // Display results
    printf("The largest number is: %.2lf\n", largest);
    printf("Its position in the sequence is: %d\n", position);

    return 0;
}
```

3. Do while loop

```
#include <stdio.h>

int main() {
    int N, i = 1, position = 1;
    double number, largest;
```

```
    // Ask for the number of values
    printf("Enter the number of values (N): ");
    scanf("%d", &N);

    if (N <= 0) {
        printf("The number of values must be positive.\n");
        return 0;
    }

    // Read the first number
    printf("Enter number 1: ");
    scanf("%lf", &largest);

    // Read the remaining numbers
    i = 2;
    do {
        printf("Enter number %d: ", i);
        scanf("%lf", &number);

        if (number > largest) {
            largest = number;
            position = i;
        }

        i++;
    } while (i <= N);

    // Display results
    printf("The largest number is: %.2lf\n", largest);
    printf("Its position in the sequence is: %d\n", position);

    return 0;
}
```

Exercise 3.11

```
#include <stdio.h>

int main() {
    int num;         // Variable to store the input number
    int even = 0;    // Counter for even numbers
    int odd = 0;     // Counter for odd numbers

    // Message to the user
    printf("Enter numbers (end with 0):\n");

    // Loop to read numbers until the user enters 0
    do {
        scanf("%d", &num);   // Read a number
```

```
        // Check that the number is not 0 (stop value)
        if (num != 0) {
            // Check if the number is even
            if (num % 2 == 0)
                even++;
            else
                odd++;
        }
    } while (num != 0);

    // Display the results
    printf("Number of even values: %d\n", even);
    printf("Number of odd values: %d\n", odd);

    return 0;               // Normal program termination
}
```

Exercise 3.12

1. Execute the algorithm with N = 6

Memory case:

i	N	Sn
1	6	1
2		3
3		6

Screen:

```
Enter N:
6
The sum is:
6
```

*1. Execute the **a**lgorithm with N = 8*

Memory case:

i	N	Sn
1	8	1
2		3
3		3
4		7

Screen:

```
Enter N:
8
The sum is:
7
```

2. What does this algorithm calculate?
It calculates the sum of all divisors of N excluding itself.

3. Translate into C language.

```
#include <stdio.h>
int main() {
    int N, Sn = 0, i;
    // Input
    printf("Enter N: ");
    scanf("%d", &N);
    // For loop to calculate sum of proper divisors
    for (i = 1; i <= N / 2; i++) {
        if (N % i == 0) {
            Sn += i;  // Add i to Sn
        }
    }
    // Output
    printf("The sum is: %d\n", Sn);
    return 0;
}
```

4. Rewrite using a While loop

```
#include <stdio.h>

int main() {
    int N, Sn = 0, i;

    // Input
    printf("Enter N: ");
    scanf("%d", &N);

    // Initialize counter
    i = 1;

    // While loop to calculate sum of proper divisors
    while (i <= N / 2) {
        if (N % i == 0) {
            Sn += i;  // Add i to Sn
        }
        i++;  // Increment counter
    }
    // Output
    printf("The sum is: %d\n", Sn);

    return 0;
}
```

References

1. Assabaa, M (2018). *Chapitre 1: Systèmes de Numération et Codage des Nombres*. Institut des Sciences et Techniques Appliquées (ISTA), UFMC1.
2. Cormen, T. H. (2013). *Algorithmes: Notions de base*.
3. Flynn, J. P. (2020). *Computer Science: An Overview* (12th Edition). Pearson.
4. Franchitti, J.-C. (2024). *Introduction to Computer Science*. OpenStax. Available online at https://openstax.org/books/introduction-computer-science/pages/1-introduction
5. Leiserson, C. E., Stein, C., & Cormen, T. H. (2017). *Algorithmique: cours avec 957 exercices et 158 problèmes*.
6. Liang, Y. D. (2014). *Introduction to Programming Using C* (2nd Edition). Pearson.
7. Mueller, J. P., & Massaron, L. (2017). *Les algorithmes pour les Nuls* (grand format).

Online Resources:

1. https://www.geeksforgeeks.org/c-programming-language/
2. https://www.tutorialspoint.com/cprogramming/index.htm
3. https://www.khanacademy.org/computing/computer-science
4. https://www.tutorialspoint.com/computer_fundamentals/computer_quick_guide.htm
5. https://www.purebasic.com/french/documentation/reference/ascii.html