Sello Editorial
Universidad Nacional
Abierta y a Distancia

Algorithms Design

Research Group
BYTE IN DESING

School of Basic Sciences, Technology, and Engineering

Sello Editorial
Universidad Nacional
Abierta y a Distancia

Algorithms Design

Author: Anivar Chaves Torres

Research Group: Byte in design

UNIVERSIDAD NACIONAL ABIERTA Y A DISTANCIA - UNAD

Jaime Alberto Leal Afanador
Chancellor

Constanza Abadía García
Vice-Chancellor for Academic and Research

Leonardo Yunda Perlaza
Vice-Chancellor for Educational Media and Learning Mediation

Edgar Guillermo Rodríguez Díaz
Vice-Chancellor for Applicants, Students, and Alumni Services

Leonardo Evemeleth Sánchez Torres
Vice-Chancellor for Interinstitutional and International Relations

Julialba Ángel Osorio
Vice-Chancellor for Social Inclusion, Regional Development, and Community Outreach

Claudio Camilo González Clavijo
Dean, School of Basic Sciences, Technology, and Engineering

Juan Sebastián Chiriví Salomón
National Director, Research Management System (RMS)

Martín Gómez Orduz
Director, UNAD University Press

005.1

D852

Chaves Torres, Anívar

Learn to design algorithms / Anívar Chaves Torres - [1.a. ed.]. Bogotá: UNAD Publishing House/2024. (School of Basic Sciences, Technology and Engineering -ECBTI in Spanish-)

e-ISBN: 978-628-7786-14-1

1. Algorithms – teaching 2. Computer 3. Programming I. Chaves Torres, Anívar

Algorithms Design
Author: Anívar Chaves Torres

e-ISBN: 978-628-7786-14-1

Research Group: Byte in Design

School of Basic Sciences, Technology and Engineering

UNAD University Press
Universidad Nacional Abierta y a Distancia
Calle 14 sur No. 14-23
Bogotá D.C.

November 2024

Layout: Hipertexto - Netizen

How to cite this book: Chaves Torres A., (2025). *Algorithms Design*. UNAD University Press. https://doi.org/10.22490/UNAD.9786287786141.

Book Review

Algorithms Design is a didactic book aimed at those beginning their journey into computer programming.

This book covers the key topics required to design and implement solutions using computer programs, starting from basic concepts such as data types and variables to advanced topics like recursion.

Each topic is accompanied by numerous examples and exercises, allowing students to practice and thus better understand the theory well and apply it in their algorithm design.

Furthermore, the book explains different ways to represent an algorithm, either descriptively through pseudocode or graphically through flowcharts or Chapin diagrams. Examples are presented using the three methods to familiarize students with them. Additionally, the book discusses methods for verifying algorithms to assess their functionality and correct them if necessary.

Author Biography

The author is a professional who has dedicated most of his career to university teaching and research. His main areas of interest and research include teaching computer programming, databases, and education. To date, he has published six books and numerous articles, some co-authored with researchers from different universities.

Anívar Chaves Torres is a Systems Engineer, holds a master's degree in Education, and a Ph.D. in Education Sciences. He serves as a professor at Universidad Nacional Abierta y a Distancia (UNAD) in the School of Basic Sciences, Technology, and Engineering. He is a member of the Byte in Design research group.

Dedication

To my loving family
for their understanding,
support, and
trust.

CONTENT

Prologue

This is a book conceived and developed by a programming teacher, who also considers himself a lifelong student of the subject. As such, he is well aware of the difficulties students experience in learning the fundamentals of programming and algorithm design, as well as the needs of teachers for reference material that includes concepts, examples, and exercises.

While the topics covered are common in the books of programming fundamentals, they are presented here with a didactic approach, using a simple language and a level of detail that anyone can understand. This book is not intended to be just a reference document but a didactic material for independent learning.

To facilitate the learning of algorithm design, an inductive approach is proposed. An algorithm is a general solution for problems of the same type, and its steps are identified as several cases of the problem are solved. The examples presented in the chapters will apply this methodology: hypothetical values will first be proposed for the problem data and calculations will be carried out to arrive at a solution; then, variables will be defined and expressions will be established to solve the problem for any set of values.

The book is organized into nine chapters. The first chapter introduces the topic by studying the general aspects of a computer–both physically and logically–as well as the software construction process. It aims to provide students with a perspective for the study of the following seven chapters.

Chapter two covers some fundamental concepts in programming, such as data types, variables and constants, operators, and expressions. It is essential for students to have a clear understanding of these concepts before delving into algorithm design, as they will be applied in each of the following topics.

Chapter three directly addresses algorithms: concepts, characteristics, notations, design strategies, and verifica-

tion. Its purpose is to provide conceptual support, as both students and professors need to agree on what an algorithm constitutes, how it should be represented to avoid ambiguity in problem-solving, and how to verify its effectiveness.

Chapter four covers the three types of programming structures: sequential, selective, and iterative. This chapter marks the real beginning of algorithm design, as the previous chapters establish the framework for understanding and applying the structures. It shows how to read data, process it, present results, make decisions, and execute certain operations or processes repeatedly.

Chapter five studies one of the most relevant topics in programming: handling and organizing data in main memory through arrays. These storage structures are easy to use and very useful for designing algorithms that operate on datasets. This chapter details all the operations performed when working with vectors, except for searching and sorting, which are covered in a separate chapter.

Chapter six explains the ***divide-and-conquer*** strategy applied to algorithm design. This strategy consists of breaking a problem down into smaller, more manageable subproblems, for which functions or procedures are designed and invoked from a main algorithm.

Chapter seven covers searching and sorting—vital topics when working with large volumes of data. Algorithms such as linear and binary search, exchange sort, insertion sort, selection sort, bubble sort, and partitioning are explained in detail.

Chapter eight focuses on recursion. This topic is included because many programming problems are easier to solve using recursion, and many of the algorithms that students need to learn are recursive, such as quicksort and algorithms for handling dynamic data structures. From this perspective, it is important for students to be familiar with recursion from their earliest courses.

In the final chapter, chapter nine, an algorithmic solution for the Rubik's Cube or magic cube is presented. This puzzle is included based on two hypotheses: first, it is very difficult to solve the cube without knowing the sequences of moves, as each attempt to arrange one piece can disarrange another. Second, it is easier to understand and assimilate a topic or concept when there is something fun to apply it to. The algorithm for solving the cube puts into practice the topics covered from chapters two through eight. Additionally, while attempting to solve the cube, two essential qualities for learning any subject, especially algorithms, such as willpower and perseverance, are exercised.

INTRODUCTION

"Quit thy childhood,
my friend, and wake up"
Rousseau

Writing a computer program is a complex activity that requires knowledge, skills, creativity, and mastery of programming techniques and tools. Before writing a program, it is necessary to carefully consider the nature and characteristics of the problem to be solved, determine how each required operation will be executed, and organize the activities needed to achieve the expected results. Only when the solution design is complete should a computer and programming language be used to write the program's code.

This book aims to facilitate the learning of concepts and techniques to analyze problems and design solutions as a preliminary step to writing programs. Although programming is not addressed under any specific language, it provides a general description of computer architecture and the program construction process to provide the reader with a perspective that facilitates the understanding of the topics developed in the following chapters.

1.1 THE COMPUTER

A computer is an electronic machine designed for data management and processing, capable of performing complex operations at high speed, following a predetermined set of instructions.

Deitel and Deitel (2004: 4) define it as "a device capable of performing calculations and making logical decisions at speeds up to billions of times faster than humans can

achieve" [quote translated from its original in Spanish], processing data under the control of a series of instructions called computer programs.

Lopezcano (1998: 83) defines it as a "machine capable of following instructions to alter information in a desirable way" [quote translated from its original in Spanish] and to perform some tasks without user intervention.

Based on Martínez and Olvera (2000), a computer is characterized by the following functions:

- Performing arithmetic and logical operations at high speed and reliably.
- Comparing data and executing different actions depending on the result of the comparison, with great speed.
- Storing both data and instructions in memory and processing the data according to the instructions.
- Sequentially executing a set of instructions stored in memory (program) to carry out data processing.
- Transferring the results of operations to output devices such as screens and printers or to secondary storage devices for user access.
- Receiving programs and data from peripheral devices such as keyboards, mice, or some sensors.

It is a general-purpose tool that can be used in different areas, such as administrative systems, process control, control of specific devices, computer-aided design, simulation, scientific calculations, communications, and security systems (Martínez & Olvera, 2000). It can handle data in different formats, including numbers, text, images, sound, and video.

Initially, computers were used for activities requiring complex calculations or handling large volumes of information, such as in government offices and universities. Over the years, their use became widespread as the number of computers manufactured increased and their cost decreased. The use of this tool is growing, as no one wants to give up its benefits in terms of comfort, speed, and accuracy for any type of task. In addition, the market now offers a greater number of physical and logical accessories that are applied to a wider range of activities.

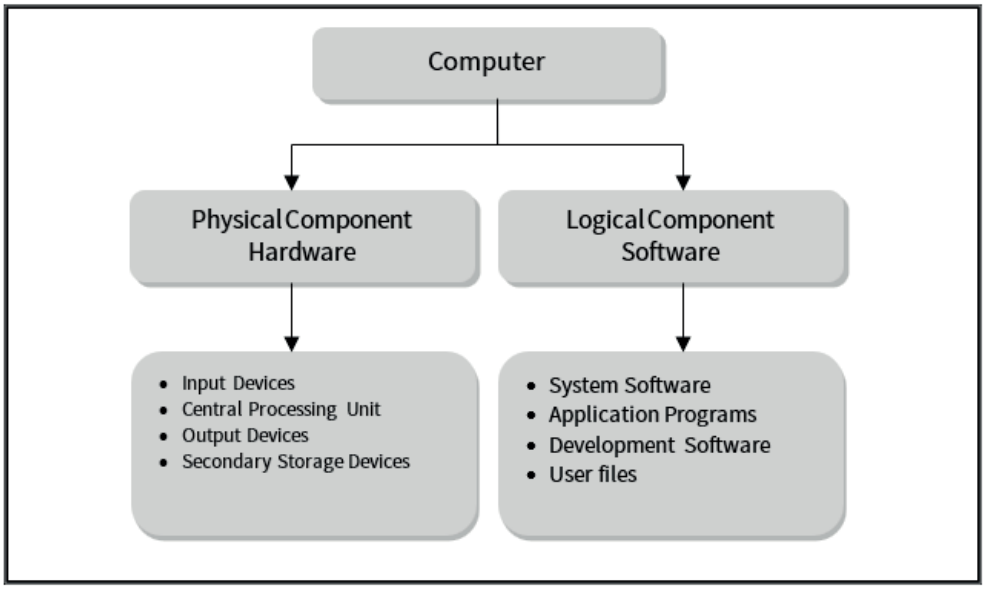
From a systems approach, the computer is more than a set of devices; what makes it a very useful tool is the relationship and interaction between its components, regulated by a set of instructions and data that constitute its guidelines.

Superficially seen, a system requires an input or raw material, on which to execute a process or transformation to generate a different product that constitutes the output. The computer can be considered a system in that the input data undergo some type of transformation (processing) to generate new data that offers better utility to the user, where such utility is the system's objective.

Unlike any other type of system, the computer has storage capacity, allowing it to perform multiple processes on the same set of data and, also, to offer the results of the process in various formats and as many times as necessary.

The computer has two types of components: physical components such as cards, disks, and integrated circuits; and logical components: programs and data (Figure 1). The physical part is called hardware, while the logical part is called software.

Figure 1. *Composition of the Computer*

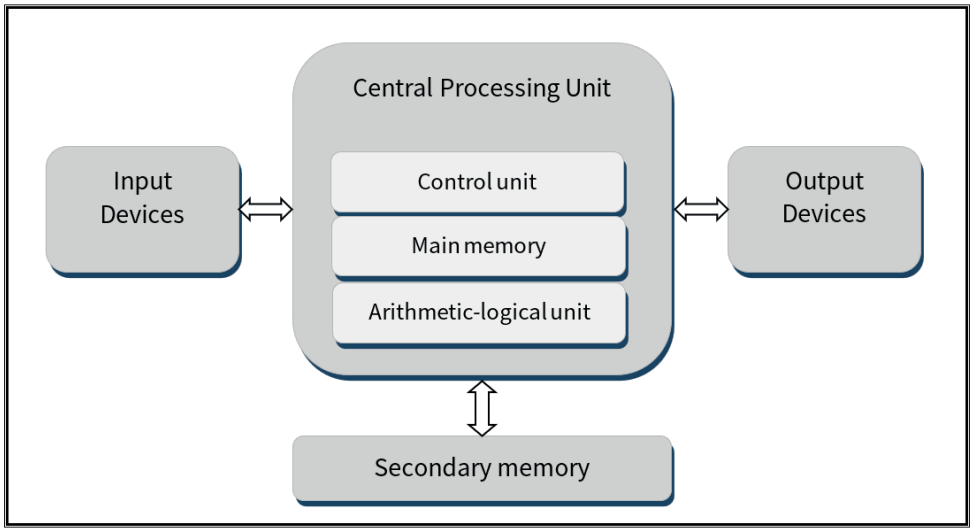


1.1.1 Physical Component

This component, commonly known by its English name: hardware, is made up of all the tangible devices and accessories that can be seen at a glance, such as cards, integrated circuits, disk drives, CD-ROM drives, monitors, printers, keyboards, mice, and speakers (Plasencia, 2008).

Hardware elements are classified according to the function they perform, which can be: data input, processing, output, or storage (Figure 2).

Figure 2. *Physical Organization of the Computer*



Input devices. Input devices receive instructions and data entered by the user and transmit them to the main memory from where the processor will access them.

The most commonly used input devices are the keyboard and mouse, followed by other less common ones such as scanners, microphones, and cameras.

The information entered through these means is transformed into electrical signals that are stored in the main memory, where it remains available to be processed or stored in permanent storage media.

Central Processing Unit. Commonly referred to as the CPU (Central Processing Unit). This is the most important part of the computer, as it is responsible for making calculations and comparisons. In the words of Becerra (1998: 1), "the processor is the one that performs the sequence of operations specified by the program" [quote translated from its original in Spanish].

This component searches for control instructions stored in memory, decodes, interprets, and executes them. It manipulates temporary storage and data retrieval,

while regulating information exchange with the outside world through input and output ports. (Lopezcano, 1998).

The central processing unit has three important parts: the control unit, the arithmetic-logical unit, and memory registers. In addition to registers (small memory spaces), the processor needs constant interaction with the main memory.

The control unit is responsible for managing all the work done by the processor; in other words, it could be said that it controls the entire operation of the computer. The functions of this unit include:

- Reading the program instructions from memory
- Interpreting every instruction read
- Reading the data referenced by each instruction from memory
- Executing each instruction
- Storing the result of each instruction

The arithmetic-logical unit (ALU) conducts a series of arithmetic and logical operations on one or two operands. The data on which this unit operates are stored in a set of registers or come directly from the main memory. The results of each operation are also stored in registers or in main memory (Carretero et al, 2001).

Registers are small memory spaces to store the data that is being processed. The main feature of registers is the speed with which the processor can access them.

The operations conducted by the ALU and the data on which they act are supervised by the control unit.

Main memory. Physically, it is a set of integrated circuits attached to a small card placed in the slot of the computer's motherboard. Functionally, the memory is the space where the processor stores the instructions of the program to be executed and the data to be processed.

Memory plays an important role together with the processor, since it minimizes the time required for any task; therefore, the more memory the computer has, the better its performance will be.

When using the term memory, it refers to RAM (random-access memory). However, there are other types of memory, such as cache memory and virtual memory, among other classifications.

RAM, or Random-Access Memory, has the advantage of fast access but the disadvantage that data is stored only temporarily. When the power supply is interrupted any reason, the data in the memory will be lost. Due to its high access speed, the more data and programs that can be loaded into memory, the faster the computer will work.

The different components of the computational unit communicate using bits. A bit is a digit in the binary system that can take one of two values: 0 or 1. Bits constitute the minimum unit of information for the computer. However, the most common term for referring to a device's memory capacity is the Byte. A byte is a collection of eight bits and can represent up to 256 characters, whether they are letters, digits, symbols, or codes known as non-printable characters, any character of the ASCII (American Standard Code for Information Interchange) system.

Since a byte (a character) is a very small unit to express the amount of information handled in a computer or in any of its storage devices, multiples of it are used, as shown in Table 1.

Table 1. *Storage Magnitudes*

Unit	Short Name	Storage Capacity
Bit	B	0 or 1
Byte	B	8 bits
Kilobyte	Kb	1024 Bytes
Megabyte	MB	1024 Kb
Gigabyte	GB	1024 Mb
Terabyte	Tb	1024 Gb

Storage devices or secondary memory. Also known as auxiliary memory, it is responsible for securely storing information, as it keeps data permanently and independently of whether the computer is running. On the contrary, the internal memory only retains information while the device is turned on. Secondary storage devices include hard drives, compact discs (CDs), digital versatile discs (DVDs), and memory chips.

Output devices. They present the results of data processing. They are the means by which the computer presents information to the user. The most common are the monitor, the printer, and the speakers.

Screen or monitor: It displays images generated according to the program or process being executed. They can be videos, graphics, photographs, or text. It is the default output, where the messages generated by the computer are presented, such as data requests, process results, and error messages.

Printer: This device prints information sent from a program on paper. Printing can be in black or color depending on the type of printer used.

Speakers: These devices amplify the audio signal generated by the computer. They are now commonly used due to the multimedia processing capacity of computing devices.

1.1.2 Logical Component

This component, commonly known as software, is made up of all the information, whether instructions or data, that makes the computer work. Without the help of software, the hardware does not perform any function.

Software is classified into four groups, depending on the task it performs: system software, application programs, development software, and user files, as shown in Figure 3.

System Software. Also known as an operating system, this is a set of programs essential for the computer to work. They manage all the resources of the computing unit and facilitate communication with the user.

Carretero et al. (2001) suggest imagining a computer stripped of software, incapable of performing any tasks. Even if a computer is equipped with the best hardware available, without program instructions in memory telling it what to do, it does nothing.

The operating system aims to simplify the use of the computer and the management of its resources, both physical and logical, in a secure and efficient way. Among its functions, three stand out:

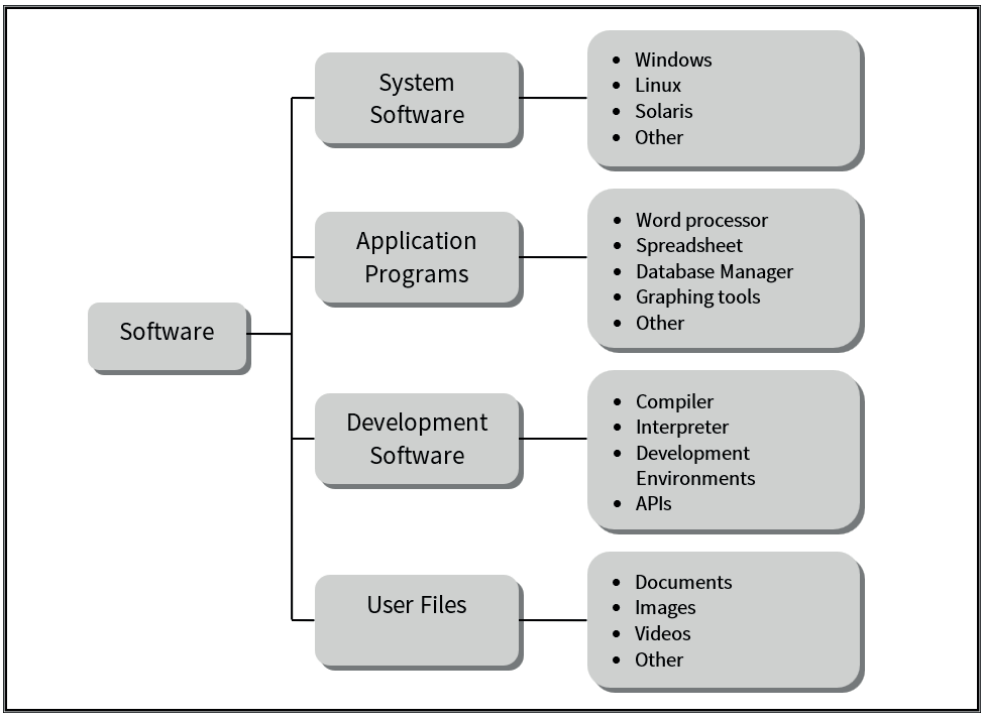
- Management of the computing unit resources
- Executing services requested by programs
- Executing user-invoked commands

To fulfill these functions, the operating system has specialized programs for various tasks, such as powering on the equipment, interpreting commands, managing input and output of information through peripherals, accessing disks, processing interrupts, and managing memory and the processor.

Some well-known operating systems include: Windows, with its different versions; Linux, with its many distributions; Netware; Unix, Solaris, among others.

Application programs. A set of programs other than system software. They manipulate the information that the user needs to process and carry out a specific task. Their purpose is to allow the user to carry out their work easily, quicky, efficiently, and accurately. This category includes several groups, such as word processors, spreadsheets, graphing tools, databases, planners, accounting programs, mathematical applications, audio and video editors and players, among others. Some examples are: Word, Excel, Access, Corel Draw, and statistical software such as SPSS and accounting software such as GC 1.

Figure 3. *Software Classification*



Development software. This category is made up of a broad range of tools for software development, including compilers, interpreters, integrated development environments, frameworks, and Application Programming Interfaces (APIs).

Most of these tools are designed to work with a certain programming language; however, there are some applications that allow the use of code from more than one language. For example, the Java platform with its JNI package and the .Net platform.

User files. This category is made up of all the files created by the user using application programs. They are generally not executable programs but rather files containing entered data or the results of processing that data. Examples of user files include images such as photographs or drawings, texts such as letters or reports, database files, and spreadsheet files such as payroll or inventory.

1.2 PROGRAM CONSTRUCTION

A few decades ago, preparing computer programs was considered attractive, especially because it was rewarding not only economically and scientifically, but also as an aesthetic experience such as the composition of poetry or music (Knuth, 1969). Programming was considered an art, where execution depended more on talent than on knowledge. Nowadays there are methodologies and techniques to do so, so that anyone can learn to program if they set their mind to it.

Although the central theme of this book is algorithm design rather than programming, it is important for readers to have a general understanding of the entire process of program construction, so that they can understand the role of algorithms in programming methodology. This is particularly relevant because programs are constructed to solve a problem, and algorithms specify the solution.

In this sense, it should be noted that building a program is a complex task that requires not only knowledge but also creativity and certain skills from the developer. Programs can be as large and complex as the problems they intend to solve and although there may be some similarities between them, just as each problem has its particularities, so do programs.

Regarding problem-solving, Galve et al. (1993) mention that there is no universal method to solve any problem, since it is a creative process where knowledge, skill,

and experience play an important role. When it comes to complex problems, the important thing is to proceed systematically.

The discipline that studies all aspects related to software production, from requirement identification to maintenance after use, is Software Engineering (Sommerville, 2005). Various methodologies have been proposed within this discipline for software construction, including: Unified Process, Extreme Programming, Rapid Application Development, and Spiral Development.

Each methodology proposes different stages and activities for software construction, however, there are some phases that are common to all, even if they are given different names. To mention just three examples, Joyanes (2000) proposes eight phases: problem analysis, algorithm design, coding, compilation and execution, testing, debugging, maintenance, and documentation. Bruegge and Dutoit (2002), on the other hand, propose: requirement identification, creation of an analysis model, overall design, detailed design, implementation, and testing. Sommerville (2005) summarizes them into four: specification, development, validation, and evolution of software.

Considering the above proposals, the process of program construction is organized into six activities, as detailed below.

1.2.1 Problem Analysis

Since problems do not always have a simple and straightforward specification, half the work is knowing what problem is to be solved (Aho et al., 1988).

The first step is to ensure that the problem to be solved is well understood and to grasp its magnitude. Often, the statement of the exercise or the description of needs provided by a client is not clear enough to determine the requirements that the program must meet. Thus, the programmer must stop and ask themselves if they truly understand the problem they need to solve.

In the real world, problems are not isolated but interconnected. Therefore, to try to solve a specific problem, it is necessary to set it out clearly and precisely, establishing its scope and restrictions. It is important to know what the computer program is expected to do; that is, to identify the problem and the solution's purpose.

In this regard, Joyanes (1996) mentions that the purpose of the analysis is for the programmer to understand the problem they are trying to solve and to be able to

specify in detail the inputs and outputs of the program. To achieve this, he recommends asking two questions:

What information should the program provide?
What information is necessary to solve the problem?

Requirement identification refers to establishing each of the functions that the program must perform and the characteristics it must have. These are like the clauses of a contract; they define with some precision what needs to be done, although not how. In highly complex programs, the list of requirements can be very extensive; in smaller programs, such as those created in introductory programming courses, it is often limited to just a few. However, establishing the requirements is a fundamental task of analysis.

Once the requirements have been identified, either through a research process or through the interpretation of the problem statement, Bruegge and Dutoit (2002) recommend developing an analysis model. This involves preparing a document that records all the information obtained and generated up to that point, preferably using models and language specific to the field of programming. In small programming exercises, it will be enough to specify the input information, the calculations' formulas, and the output information.

In summary, problem analysis must at least identify the relevant information related to the problem, the operations to be performed on that information, and the new information that must be produced; that is, to know what data the program operates on, what operations, calculations, or transformations it must perform, and what results are expected.

1.2.2 Solution Design

Once the problem has been analyzed and understood, one can begin to think about the solution. Design consists of applying the available knowledge to answer the question: How can the problem be solved?

Designing a solution involves generating, organizing, and representing ideas about the problem's solution. To this end, there are design techniques and tools specific to each programming model, which allow to record and communicate the operations and processes to be developed. Under the procedural programming model, algorithms are used as a solution design; while in the functional model, the functions are used, and in the object-oriented model, the conceptual models or diagrams are used.

Ideas regarding the solution of a problem, i.e., the design of the solution, must be represented using a language familiar to the programming community, so that the design is communicable. In large software development projects, people designing are not the same people writing the program code. Thus, it is essential that the solution in the design phase be documented in such a way that others can fully understand it to carry out the following phases.

In general terms, the design phase should consider four aspects of the solution:

User Interaction: Or user interface design; this means thinking about how data will be exchanged between the program and the person using it, how the program's required data will be captured, how results will be presented to the user, and how the user will indicate what they want the program to do.

Data Processing: This means knowing the operations performed on the data according to the problem domain. For example, if it is a program to calculate a company's payroll, one must know the processes involved to obtain that payroll. It is important to remember that one cannot program a process that is unknown. The programmer must first have a clear understanding of the calculations needed to obtain the desired results. The computer will then automate the process, but the programmer must specify how to do it.

System Architecture: To solve a problem more easily, it is necessary to break it down into subproblems so that each one is easier to solve (this topic is developed in Chapter 6), but all parts must be integrated to provide a complete solution. System architecture refers to how the parts of the solution are organized and interact with each other. During the design phase, the system is broken down into different modules or subsystems, and simultaneously, the architecture is designed to show how they relate to one another.

Data Persistence: All programs operate on data and must be stored in an organized way so that the program can save and retrieve it safely. During the design phase, it is decided how the data will be stored on the secondary storage media.

1.2.3 Program Coding

This is the translation of the problem's solution expressed in design models into a series of detailed instructions in a language recognized by the computer, known as a programming language. The set of detailed instructions is referred to as source code and must be compiled to become an executable program.

Unlike design, coding can be a simple task, provided there is a detailed design and the reserved words of the language, along with its syntax and semantics, are known.

It should be noted that the stages on which the greatest effort and interest should be focused to develop the necessary skills are analysis and design. While acknowledging the importance of language proficiency, during the coding phase, a good reference book can be helpful, whereas there is no book that can assist you in analyzing the problem and designing the solution.

The coding stage includes reviewing the results generated by the source code, line by line, and this should not be confused with the program test discussed in the next section.

1.2.4 Testing and Debugging

When coding a program, partial tests are developed to verify that the instructions are written correctly and perform the expected function. However, just because a program works or runs does not mean that it fully meets the needs for which it was developed.

Human errors in computer programming are numerous and increase considerably with the complexity of the problem. The process of identifying and eliminating errors to arrive at an effective solution is called ***debugging***.

Debugging ***or testing*** is as important a task as the development of the solution itself, and should be approached with the same interest and enthusiasm. Testing should consider all possible errors that may arise during the execution and use of the program.

The purpose of testing is to identify weaknesses in the program before it is launched, allowing for error correction without the loss of information, time, or money. A successful test is one that detects errors, as all programs have them; thus, tests are designed with the goal of debugging the program, not confirming that it is well-constructed.

1.2.5 Documentation

Often, a program written by one person is used by another. Therefore, documentation serves to help understand or use the program, or to facilitate future modifications (maintenance).

The documentation of a program is the guide or written communication in various forms, whether in statements, drawings, diagrams, or procedures.
There are three types of documentation:

- Internal Documentation
- Technical Documentation
- User Manual

Internal Documentation: These are comments or messages added to the source code to make it clearer and more understandable.

In a large program, the programmer himself/herself will need the comments in the code to locate himself/herself and make the corrections or changes necessary after a while. This documentation is much more important if the changes will be made by personnel other than those who wrote the original source code.

Technical Documentation: This is a document that records relevant information regarding the problem posed and how the solution was implemented. Some data to include are:

- Description of the Problem
- Analysis Models
- Solution Design
- Data Dictionary
- Source Code (program)
- Tests Conducted

This documentation is aimed at personnel with programming knowledge. Hence, it contains technical information about the program and should not be confused with the user manual.

The construction of large and complex programs requires documentation to be developed as the analysis, design, and coding of the software advances. This is due to two reasons: first, the construction of large programs requires the work of several people and all specifications must be kept in writing; second, it is not possible to keep all project information in the memory of those responsible; physical support is necessary.

User Manual: This is a document intended for the end user that describes the program's operation step by step. It should include detailed information about the ins-

tallation process, data input, processing, and result retrieval, as well as recommendations and information about potential errors.

1.2.6 Maintenance

It refers to the changes that need be made to the program after it has been put into operation. These changes may be intended to include new processes or adapt the program to circumstances that have changed since its development.

Maintenance will be easier to carry out if proper program documentation is available.

The phases exhibited here do not necessarily occur in the order described, since many development models are iterative; however, they will always be present.

2. PROGRAMMING ELEMENTS

Programming
is the art and technique
of systematically constructing
and formulating algorithms.
Wirth[1]

Algorithms and programs are made up of series of operations carried out with data. These data are grouped into categories based on the values they can contain and the operations that can be performed on them. Operations are specified through expressions made up of operators and variables, the latter referring to the data involved in the operation.

This chapter pursues the study of those basic yet indispensable elements in the design of algorithms and the implementation of programs.

2.1 DATA TYPES

Data types are classifications used to organize the information stored in computer memory. These abstractions allow for defining minimum and maximum values for each type, thus establishing the space each requires and facilitating memory management.

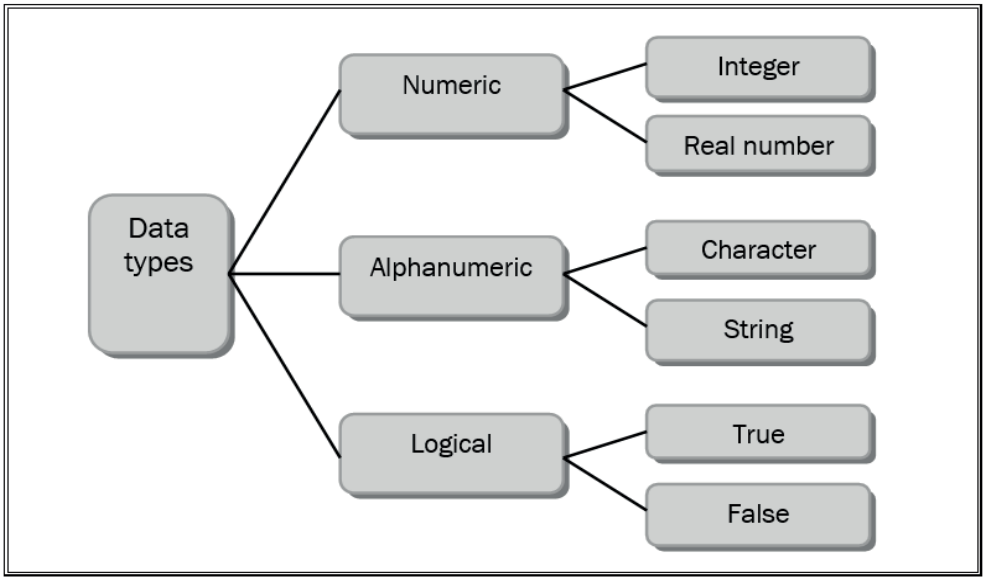
Data types are used in variable declaration and in validating the operations allowed on each type. For example, numeric data supports arithmetic operations, while strings can be concatenated.

Programming languages that handle data types define three elements: primitive or simple types (numbers, characters, and Booleans), the set of values they can handle, and the allowed operations. From primitive data, composite types can be implemented, such as strings, arrays, and lists (Appleby and Vandekppple, 1998).

1 Anívar Nestor Chaves Torres

Among primitive data types, there are three types: numeric, alphanumeric, and logical or boolean. Numeric data can be integers or real numbers, while alphanumeric data can be characters or strings, as shown in Figure 4.

Figure 4. *Data Types*



2.1.1 Numeric Data

Numeric data types represent quantities or quantifiable information, such as the number of students in a course, an employee's salary, a person's age, the value of an appliance, the area of a country in square kilometers, or a student's grade.

Integer Data**.** These are data expressed as exact numbers, meaning they have no fractional component and can be positive or negative. This type is used to represent elements that cannot be fractional in the problem domain, for example:

Data	Value
The number of employees in a company	150
The subjects a student is taking	5
The number of goals scored in a soccer match	3
The number of votes received by a candidate	5678

It is important to note that the range of integer values between negative infinity and positive infinity cannot be managed in programming languages since the space reserved in RAM to store a number is limited. The minimum and maximum values that can be stored in memory vary depending on the language.

Some programming languages reserve two bytes for integer data, thus, the range of supported values is between -32,768 and 32,767.

Real Number Data. These are data expressed as numbers that include a fraction and can be positive or negative. This type of data is used to represent elements that in the problem domain have values made up of an integer part and a decimal part, for example:

Data	Value
A person's height (in meters)	1.7
Room temperature (in degrees Celsius)	18.5
A student's grade (on a scale of 5.0)	3.5
The monthly interest rate	2.6

Programming languages reserve a fixed number of bytes for real-type data, so the size of these numbers is also limited.

2.1.2 Alphanumeric Data

Alphanumeric data do not represent a numerical quantity or value nor are used to quantify but to describe or qualify an element to which they refer. For example, the color of a fruit, the address of a house, a person's name, an employee's position, or gender.

Alphanumeric data can consist of alphabet characters, numbers, and other symbols; however, even if they include digits, they cannot be operated mathematically.

Characters**.** Characters are each of the symbols included in a coding system; they can be digits, letters of the alphabet and symbols. The most well-known system today is ASCII (American Standard Code for Information Interchange). This requires one byte of memory to store each character and includes a total of 256 characters. Examples of characters are: 1, a, %, +, B, 3.

Strings**.** They are sets of characters enclosed in quotation marks (" ") that are manipulated as a single piece of data. For example:

A person's name: "Joseph"
The location of a university: "Calle 19 con 22"
A book title: "Algorithm Design"

2.1.3 Logical or Boolean Data

Logical data can only take two values: true or false. In programming, they are frequently used to refer to the fulfillment of certain conditions, such as the existence of a file, the validity of a piece of data, or the relationship between two pieces of data.

2.2. VARIABLES AND CONSTANTS

All computer programs, regardless of their function, operate on a set of data. During the execution of a program, the data that it uses or generates resides in memory. Information about where a piece of data is stored, its type, and its value is managed through the concept of a variable. If the value remains unchanged throughout the program's execution, it is called a constant.

2.2.1 Variables

To manipulate a piece of data in a computer program, it is necessary to identify it with a name and know: the memory location where it resides, the type to which it belongs, and its value. To make it easier to handle this information, there is an abstraction called a variable.

According to Appleby and Vandekopple (1998), a variable is associated with a tuple made up of the following attributes:

Identifier (variable name)
Address (memory location where the data is stored)
Data Type (specifically: set of values and operations)
Value (data stored in memory)

Identifier. An identifier is a word or sequence of characters that refers to a memory location where a piece of data is stored.

The length of an identifier and the way it is constructed can vary from one programming language to another. However, there are some recommendations to keep in mind:

- It should start with a letter between A and Z (upper or lower case)
- It should not contain whitespace
- It should relate to the data being stored in the memory location (mnemonic)
- It should not contain special characters and operators
- After the first letter, digits and the underscore character (_) can be used.

For example, to store a person's name, the identifier can be:

Person_name
Pname
Pn

It is common for the identifier to represent more than one word since similar data may correspond to different information elements. For example:

Student Name
Professor Name
Student ID
Professor ID
Student Phone
Professor Phone

In these cases, it is advisable to use a fixed number of characters from the first word combined with some from the second, applying the same technique to form all identifiers to make them easier to remember.

The above data identifiers could be formed as follows:

Data	Identifier	Or also
Student name	stud_name	sname
Professor name	prof_name	pname
Student ID	stud_ID	sid
Professor ID	prof_ID	pid
Student phone	stud_phone	sphone
Professor phone	prof_phone	pphone

In the second column, the identifiers are formed using the first three characters of each word in Spanish, while in the third column, they use the first and last characters of the first word in Spanish and only the first character of the second.

Each programmer has their own strategy for forming identifiers for variables, constants, functions, and user-defined data types. The important thing is to follow the previously mentioned recommendations and consistently use the same strategy to ease the programmer's memory work.

2.2.2 Types of Variables

Variables can be classified based on three criteria: the type of data they store, their function, and their scope.

Based on the type of data they store, variables can be integer, real, character, string, logical, or any other type defined by the programming language.

In terms of functionality, variables can be working variables, counters, accumulators, and switches.

The scope determines the space in which variables exist, they can be global or local.

Working variables. These are variables declared to save values read or calculated during the program's execution.

Example:

```
Real: Area
Integer: base, height
Height = 10
Base = 20
Area = base * height / 2
```

Counters**.** These are variables used to record the number of times an operation or group of operations is executed. The most common use is as a finite loop control variable, where they store the number of iterations. They can also record the number of records in a file or the amount of data that meets a condition.

Counters are incremented or decremented by a constant value, usually one at a time.

Example: If you want to enter the final grades of students in a group to calculate the group's average, since the group size can vary, a variable (counter) is required to record the number of grades entered, so that the average can then be calculated.

Accumulators**.** They are also called totalizers. These are variables used to store values that are read or calculated repeatedly. For example, to calculate the average grades of a group of students, the first step is to read the grades and sum them in a variable (accumulator) so that after reading all the grades, the total can be divided by the number of students to obtain the average.

In this example, a counter is referenced to know the number of students and an accumulator to sum the grades.

The key difference between an accumulator and a counter is that the accumulator does not have regular increments; it increases according to the value read or the result of an operation.

Switches**.** Also known as flags or sentinels, switches are variables that can take different values during the program's execution, and based on these values, the program can vary the sequence of instructions to be executed, meaning it can make decisions.

Example 1: A switch can be used to inform any module of the program if a specific file has been opened. The logical variable is declared and initialized to false, and when the file is opened, it changes to true. Thus, at any moment, to check if the file has been opened, you just verify the switch's status.

Example 2: If searching for a record in a file, a logical variable is declared and initialized to false, indicating that the record has not been found. After searching, when the record is located, the variable's value is changed to true. At the end of the search, the variable (switch) will inform if the record was found or not, depending on whether its value is true or false, allowing the program to know which instructions to execute.

Example 3: It is common for a program to be protected by a security system. In some cases, there are different user levels, where access to certain modules depends on the level. The program will enable or disable certain options depending on the access level, which will be stored in a variable (switch).

Local variables**.** A variable is considered local when it exists only within the module or function in which it was declared. This type of variable is useful in modular pro-

gramming, as it allows each procedure or function to have its own variables without conflicting with those declared in other modules.

By default, all variables are local to the scope where they are declared. If you want to declare them as global you need to specify them as such.

Global variables**.** These are variables declared to be used in any module[2] of the program. They exist from the time they are declared until the program execution ends.

The concept of local and global variables is less useful when solving problems with a single algorithm. When the algorithm is divided into subroutines, it is important to differentiate between variables that disappear at the end of a procedure or a function and those that transcend modules, as updates to one can affect others.

Programming languages generally define variables as local unless otherwise noted. Following such logic, this book proposes using the global modifier before the declaration of the variable to specify that it is global.

```
Global integer: x
```

Keep in mind not to use the same identifier for a global variable when declaring a local variable, as updating it within the subroutine could lead to confusion regarding which memory location is being accessed.

2.2.3 Variable Declaration

This operation consists of reserving enough memory space to store a piece of data of the specified type while including the identifier in the program's variable list.

To declare a variable, write the data type followed by the identifier of the variable, as follows:

```
DataType identifier
```

2 The concepts of module, subroutine, procedure, and function are related to the "divide-and-conquer" strategy applied to face the complexity of software development. It consists of breaking a large problem into smaller problems that are easier to solve, and by making the partial solutions interact, the overall problem is solved. This topic is discussed in Chapter 6.

If you want to declare multiple variables of the same type, write the data type followed by a list of identifiers separated by commas (,) as follows:

DataType identifier1, identifier2, identifier3

Example:

Integer: age
Real: height, weight, salary
String: name, surname, address

Although some programming languages allow variables to be declared when needed, it is advisable, for good programming practices, to always declare variables before using them, ideally at the beginning of the program or function.

2.2.4 Value Assignment

This operation consists of storing a piece of data in a memory location using a previously declared variable. The assigned data must correspond to the type for which the variable was declared.

An assignment expression has the form:

variable = data

Example:

Variable Declaration
String: name, address
Real: salary
Value Assignment

name = “Joseph”
address = “Carrera 24 15 40”
salary = 1000000

You can also assign the result of an expression to a variable, in the form:

variable = expression

Example:

Variables Declaration:
Integer: base, height
Real: area

Assignment:

base = 10
height = 5
Area = base * Height

An assignment expression has three parts: a variable, the equal sign, and the value or expression whose result is assigned to the variable. The variable always appears on the left of the equal sign, while the value or expression is on the right.

2.2.5 Constants

A constant refers to a memory location and is formed in the same way as a variable, except that the stored data does not change during program execution.

Constants are used to avoid writing the same values in different parts of the program; instead, a reference to the constant is used. Therefore, when it is necessary to change this value, it is enough to make change in a single line of code where the constant was defined.

Another reason for establishing a constant is that it is easier to remember and write an identifier than a value. For example, it is easier to type ***pi*** than 3.1416, especially if the value needs to be used repeatedly in the program.

2.3 OPERATORS AND EXPRESSIONS

Operators are symbols that represent operations on data. Expressions are combinations of operators and operands (data) that generate a result.

In programming, there are three types of operations: arithmetic, relational, and logical, each with a set of operators that allows the expressions to be constructed.

2.3.1 Arithmetic Operators and Expressions

Arithmetic operators are applied to numeric data and allow for arithmetic operations. They are represented in Table 2.

Table 2. *Arithmetic Operators*

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
Mod	Module

Arithmetic expressions combine the operators in Table 2 with numeric data to generate a new number as a result. For example, if you have the base and height of a triangle, you can calculate its area using an arithmetic expression.

Example:

Variable Declaration

Real: base, height, area

Value Assignment

base = 10
Height = 15

Arithmetic expression to calculate the area:

base * height / 2;

Assignment of the result of an arithmetic expression to a variable:
area = base * height / 2

The result of this expression is 75 and is stored in the variable area.

Arithmetic operators can operate on both integer and real data types, except for the ***Mod*** operator, which only applies to integers.

Some authors, such as Becerra (1998) and Cairó (2005), differentiate regular division (/) and integer division, using the reserved keyword ***div*** for the latter. In this book, the symbol / is used for all division operations. If the operands are integers, the result will also be an integer, which is referred to as integer. If either of the operands is real, the result will be of real type.

The Mod (modulus) operator returns the remainder of an integer division. Example:

10 Mod 2 = 0 since 10 / 2 = 5 and the remainder is 0
10 Mod 4 = 2 since 10 / 4 = 2 and the remainder is 2

Combining different arithmetic operators in the same expression can lead to ambiguity, meaning there may be multiple interpretations and therefore multiple results. To avoid such ambiguity, operators have a hierarchy that determines which one is executed first when there are several in the same expression. To alter the order of execution determined by the hierarchy, parentheses are needed.

As shown in Table 3, the hierarchy of operators start with parentheses, which allow for the construction of subexpressions that are developed first. Then, we have multiplication, division, and modulus, which have a higher level of hierarchy than addition and subtraction. This means that if an expression contains both multiplication and addition, multiplication is executed first, followed by addition. When several operators of the same level, such as addition and subtraction, are present, execution occurs from left to right.

Examples:

3 * 2 + 5 = 6 + 5 = 11
3 * (2 +5) = 3 * 7 = 21

6 + 4 / 2 = 6 + 2 = 8
(6 + 4) / 2 = 10 / 2 = 5

5 * 3 + 8 / 2 – 1 = 15 + 4 - 1 = 18
5 * (3 + 8) / (2 - 1) = 5 * 11 / 1 = 55

10 + 12 Mod 5 = 10 + 2 = 12
(10 + 12) Mod 5 = 22 % 5 = 2

Table 3. *Hierarchy of Arithmetic Operators*

Priority level	Operator	Operation
1	()	Grouping
2	*, /, Mod	Multiplication, division and modulus
3	+, -	Addition, subtraction, increment and decrement

2.3.2 Relational Operators and Expressions

Relational operators allow comparisons between data of the same type. Relational expressions yield a logical data type: true or false. These expressions are mainly

used for making decisions or controlling loops. The operators are shown in Table 4.

Table 4. *Relational Operators*

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

Some examples of relational expressions are:

Variable declaration and initialization

Integer x = 5;

Relational expressions:

x < 10 = true
x > 10 = false
x = 10 = false

Variable declaration and initialization

Real a = 3, b = 1

Relational expressions:

a = b = false
a > b = true;
a < b = false
a <> b = true

2.3.3 Logical Operators and Expressions

These operators are used only to operate on logical or boolean data (true or false). Logical expressions yield another logical data type. The operators are shown in Table 5.

Table 5. *Logical Operators*

Operator	Operation
AND	Conjunction
OR	Disjunction
NOT	Negation

Table 6 shows the results of operating logical data with the AND, OR, and NOT operators.

Since logical data is not common in the real world, most logical expressions are built from relational expressions. Some examples are:
Real grade = 3.5;

(grade >= 0) **AND** (grade <= 5.0) = True

Integer a = 15, b = 8;

(a > 10) **OR** (b > 10) = True
NOT (b > 5) = false

Table 6. *Result of Logical Operations*

Operand 1	Operator	Operand 2	=	Result
True	AND	True	=	True
True		False		False
False		True		False
False		False		False
True	OR	True	=	True
True		False		True
False		True		True
False		False		False
	NOT	True	=	False
		False		True

2.3.4 General Hierarchy of Operators

Arithmetic, relational, and logical operators can be mixed in the same expression. In these cases, it is necessary to pay special attention to their hierarchy to ensure that each one operates on valid types and does not generate runtime errors. For this purpose, it is advisable to consider the hierarchical order corresponding to each operator to establish the execution order and enforce priority for expressions to be developed correctly. The hierarchy of all the operators studied in this chapter is presented in Table 7.

Table 7. *Hierarchy of Operators*

Priority	Operators
1	()
2	*, /, Mod, NOT
3	+, -, , AND
4	>, <, >=, <=, <>, =, OR

Note that operators at the same level are executed from left to right.

3. ALGORITHMS

Time is a master of ceremonies
that always ends up placing us
where we belong;
we move forward,
stop and move backwards
according to its orders.
Saramago[3]

The word algorithm comes from the Latin translation of the Arabic term Al-***Khowarizmi***, which means "originating from ***Khwarizm***," the ancient name for the Aral Sea (Galve et al., 1993). On the other hand, Al-***Khowarizmi*** was the name of a Persian mathematician and astronomer who wrote a treatise on the manipulation of numbers and equations in the 9th century (Brassard and Bratley, 1997), (Joyanes et al., 1998).

3.1 CONCEPT OF ALGORITHM

Different definitions of algorithms have been proposed, including: "a finite and unambiguous set of steps expressed in a certain order that, for some initial conditions, allow solving the problem in a finite time" (Galve et al., 1993: 3); or, "a set of rules to perform a calculation, either by hand, or more frequently, on a machine" (Brassard and Bratley, 1997: 2).

Similarly, it has been considered to be a well-defined computational procedure that takes a value or a set of values as input and produces some value or set of values as output. Thus, an algorithm is a sequence of computational steps to transform input into output. (Comer et al., 2001).

According to Becerra (1998), an algorithm is the way to solve a problem in a computer; it is a group of instructions written according to syntactic rules provided by a programming language.

For Hermes (1984:19), "an algorithm is a general procedure that obtains the answer to any suitable problem through a simple calculation according to a specified me-

3 [Translation of the original epigraph in Spanish]

thod," and he later mentions that this procedure must be clearly specified so that there is no room for the imagination and creativity the executor.

An algorithm can also be defined as "a series of detailed and unambiguous operations to be performed step by step, leading to the solution of a problem" (Joyanes, 1992: 2).

When a problem is said to have an algorithmic solution, it means that it is possible to write a computer program that will obtain the correct answer for any set of input data (Baase and Gelder, 2002).

For a problem's solution to be translatable into a programming language, the steps expressed in the algorithm must be detailed, so that each involves a trivial operation; that is, the steps do not involve processes requiring an algorithmic solution. If this situation arises, the algorithm must be refined, meaning it should be redeveloped for the specific task in question.

If the problem to be solved is very large or complex, it is advisable to break it down into tasks that can be addressed independently and are easier to solve. This is called modular design.

As an example, consider Euclid's algorithm for calculating the Greatest Common Divisor (GCD). The GDC of two integers is the largest number that divides both, which can also be one of the numbers, as every number is divisible by itself.

This algorithm establishes specific steps that, when executed a certain number of times, regardless of the proposed numbers, will always find the greatest common divisor. The solution is presented later in the four notations explained in this chapter.

3.2 CHARACTERISTICS OF AN ALGORITHM

An algorithm must have at least the following characteristics:

- **Precision**: This means that the operations or steps of the algorithm must be developed in a strict order, as the development of each step must follow a logical sequence.

- **Definiteness:** The result of executing an algorithm depends exclusively on the input data; that is, whenever the algorithm is executed with the same set of data, the result will be the same.

- **Finiteness**: This characteristic implies that the number of steps in an algorithm, no matter how large or complicated the problem it solves, must be limited. Every algorithm, regardless of the number of steps it includes, must reach an end. To make this characteristic evident, the representation of an algorithm always includes start and end steps.

- **Notation**: For the algorithm to be understood by any interested person, it must be expressed in one of the commonly accepted forms to reduce the ambiguity inherent in natural language. Some well-known notations include pseudocode, flowcharts, Nassi/Shneiderman diagrams, and functional representations.

- **Correctness**: The algorithm must be correct, i.e., it must meet the need or solve the problem for which it was designed. To ensure that the algorithm achieves its goal, it must be tested; this is called verification, and a simple way to verify it is through desk checking.

- **Efficiency**: Discussing the efficiency or complexity of an algorithm involves evaluating the computational resources required to store data and to perform operations against the benefit it offers. The less resources required, the more efficient the algorithm.

If the description of a procedure or the statement of a solution to a problem lacks these characteristics, it cannot technically be considered an algorithm. For example, cooking recipes are very similar to algorithms in that they describe the procedure for preparing some kind of food. Some authors, such as López, Jeder and Vega (2009), have taken cooking recipes as examples of algorithms. However, they are rarely defined enough to be an algorithm, as they often leave room for the subjectivity of the person executing the actions.

The fundamental difference between a recipe and an algorithm is that the former is not defined, as shown in the example in Table 8, taken from Lopez, Jeder and Vega (2009).

Table 8. *First Algorithm for Preparing a Cup of Coffee*

1	Turn on a burner
2	Place a pot of milk on the burner
3	Wait for the milk to boil
4	Place coffee in a cup
5	Pour some milk into the cup and stir
6	Pour more milk into the cup until full
7	Add sugar to taste

These types of actions cannot be carried out automatically by a computer, as they require human judgment. To make it an algorithm, in the sense supported by this book, it would need to be written as shown in Table 9.

Table 9. *Second Algorithm for Preparing a Cup of Coffee*

1	Begin
2	Turn on a burner
3	Place a pot with 200 ml of milk on the burner
4	Place 10 g of instant coffee in a cup
5	Place 10 g of sugar in the cup
6	Mix sugar and coffee
7	Wait until the milk boils
8	Turn off the burner
9	Pour the milk into the cup
10	Mix until the coffee and sugar dissolve
11	End

The first version corresponds to a recipe, the second to an algorithm. In the second version, the beginning and end are specified, the steps have a logical order, the activities are defined (with explicit quantities) so that if it is applied several times with the same ingredients, the same result will be obtained. In other words, the final product depends on the algorithm and not on the person executing it.

3.3 NOTATIONS FOR ALGORITHMS

An algorithm is defined as the set of steps to find the solution to a problem or to perform a calculation. These steps can be expressed in different forms, such as descriptions, pseudocode, diagrams, and mathematical functions. The simplest form is

undoubtedly prose description; however, natural language is very broad and ambiguous, which makes it very difficult to achieve a precise description of the algorithm. Given that clarity is a fundamental characteristic of algorithm design, it is preferable to use pseudocode. Graphical representation is done through diagrams that are easy to understand and verify; the main types include flowcharts and N-S diagrams. Mathematical representation is typical of the functional programming model and consists of expressing the solution through mathematical functions, preferably using Lambda Calculus[4].

3.3.1 Textual Description

This is undoubtedly the simplest way to write an algorithm; it involves creating a list of activities or steps in prose form, without applying any notation.

This way of presenting an algorithm has the disadvantage of leading to inaccuracies and ambiguities due to the broadness of language. It is appropriate when writing the preliminary version of an algorithm and when the focus is on logic rather than on how the ideas are communicated. Once the algorithm is completed, it is advisable to convert it into one of the notations explained below.

Example 1. Greatest Common Divisor

One of the oldest known algorithms is the one designed by Euclid to calculate the Greatest Common Divisor (GCD) between two numbers. The GCD (a,b)—read as the greatest common divisor of the numbers a and b—consists of identifying a number c that is the largest number that divides both a and b exactly.

This algorithm expressed descriptively would be:

Read the two numbers, then divide the first by the second and take the remainder. If the remainder is zero, then the GCD is the second number. If the remainder is grea-

4 The Lambda Calculus is a formal system that defines a notation for computable functions, facilitates the definition and evaluation of expressions and is therefore the theoretical foundation of the functional programming model. It was developed by Alonso Church in 1936.
Euclid (Tyre, 330 - Alexandria, 300 B.C.), Greek mathematician. By order of the pharaoh Ptolemy I Sóter (the savior), he wrote the work Elements, composed of 13 volumes in which he systematized the mathematical knowledge of his time; other works are also attributed to him, such as: Optics, Data, On the divisions, Phenomena and elements of music (Gran Enciclopedia Espasa, 2005: 4648).

ter than zero, divide the second number over the remainder of the last division and continue dividing until the remainder is zero. The solution will be the number used as the divisor in the last division.

3.3.2 Pseudocode

Pseudocode is a mixture of a programming language and natural language (Spanish, English, or any other language) in the presentation of an algorithmic solution. This notation allows for structured algorithm design while using the language and vocabulary familiar to the designer.

Pseudocode approximates the structure and syntax of a given programming language, which facilitates the understanding of the algorithm and its coding, as there is a close correspondence between the organization of the algorithm and the syntax and semantics of the language.

Considering that natural language is very broad, while programming languages are limited, it is necessary to use a set of instructions that represent the main structures of programming languages. These instructions are referred to as reserved words and cannot be used as variable identifiers.

Some reserved words are:

String	Function	Logic	Repeat
Character	Do	While	Return
Decrement	Until	For	Switch
Integer	Increment	Procedure	If
Write	Begin	Real	Else
End	Read		

Indentation

One of the difficulties with pseudocode is that it is not easy to see programming structures such as branches and loops, as instructions appear one after the other, regardless of the sequence in which they execute.

Indentation consists of leaving a space or tab for instructions within a programming structure to indicate their dependency and to show where a structure begins and ends.

Advantages of using Pseudocode

Some advantages of this notation are the following:

- It takes up less space
- It allows to represent repetitive operations easily
- It is very easy to translate from pseudocode to program code in a programming language
- When indentation is applied, it clearly shows the dependence of the actions on the programming structures.

In Table 10, the Euclidean algorithm is presented in pseudocode notation. This example shows the application of the concept of indentation to highlight the dependence of the instructions relative to the "while" structure and the start - end of the algorithm.

Table 10. *Algorithm for Obtaining the GCD*

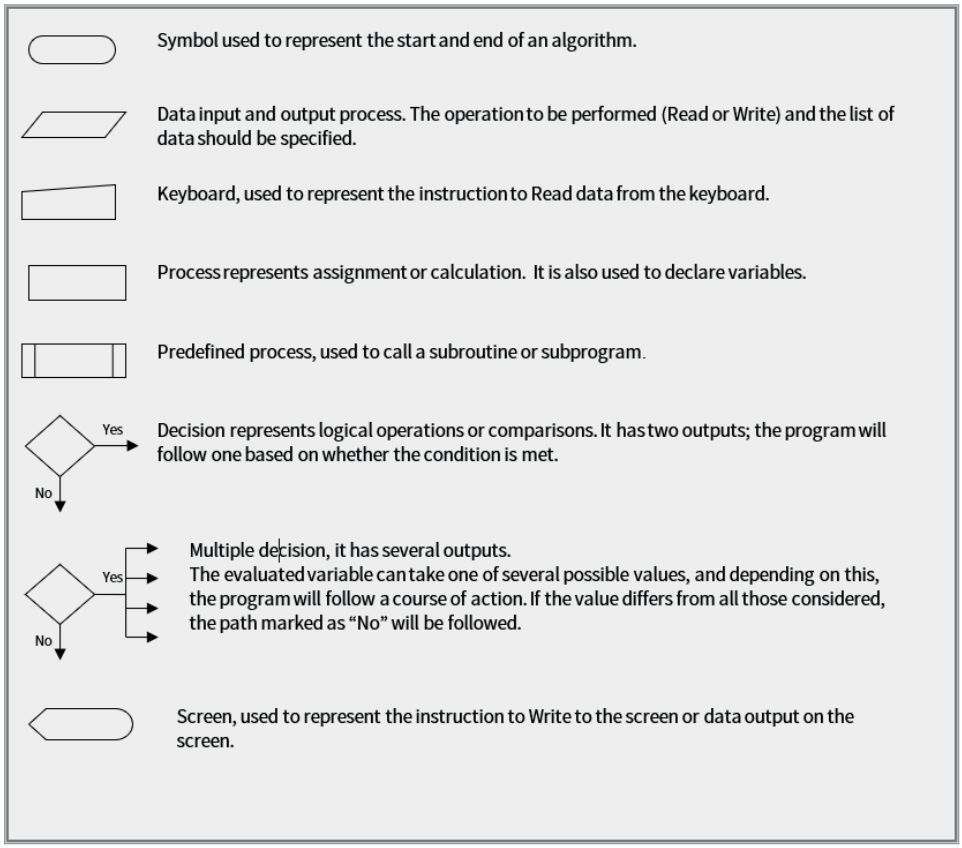
```
1     Begin
2         Integer: a, b, c=1
3         Read a, b
4         While c>0 do
5             c = a Mod b
6             a = b
7             b = c
8         End while
9         Write "GCD = ", a
10    End algorithm
```

In this algorithm, the two numbers a and b are read, and variable c is initialized to 1 to ensure that the loop executes. Successive divisions are performed, first with the input numbers and then taking the divisor as the dividend and the remainder as the divisor, until an integer division occurs. When this happens, the number that has acted as the divisor will be the solution or GCD.

3.3.3 Flowchart

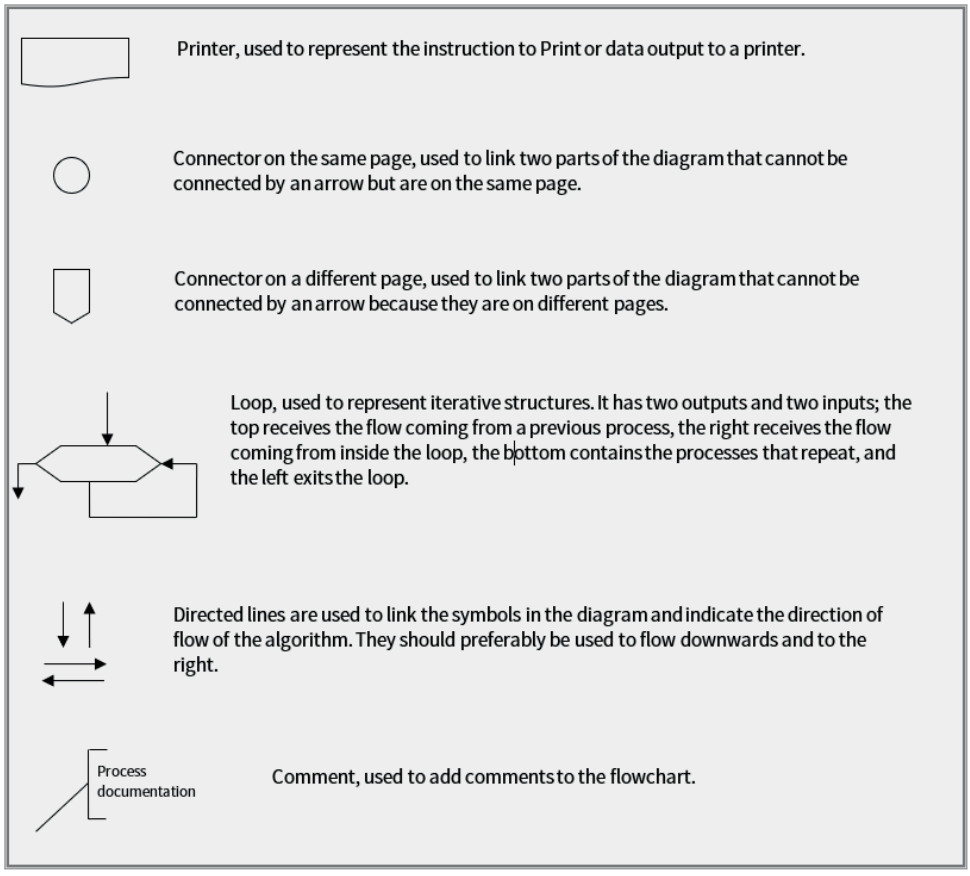
This is the graphical representation of an algorithm using a set of symbols that represent the basic operations and structures of programming. The set of symbols used to design flowcharts is shown in Figure 5.

Figure 5. *Symbols Used for Designing Flowcharts*



The representation using flowcharts makes the algorithm easy to understand, as its structure and flow of operations are visible. However, when the algorithm is long, constructing the flowchart can be challenging. Although connectors can be used inside and outside the page, their use can affect the understanding of the algorithm's logic.

Figure 5. *(Continued)*



Recommendations for constructing flowcharts

Based on Cairó (2005), here are some recommendations for properly constructing flowcharts:

- The first symbol in any flowchart must be the start symbol, and regardless of the paths taken, it should always lead to the *end* symbol.
- All symbols must be linked with directed lines, representing the inflow and outflow, except for the start, which has no inflow, and the end, which has no outflow.
- Lines should be straight, either horizontal or vertical, and should not cross. Whenever possible, use only downward and rightward lines to facilitate the reading of the algorithm. An exception to this rule is cycles.

- Every line must start with a symbol and end with a symbol.
- Connectors should be numbered consecutively and used only when necessary.
- Some symbols can be used to represent more than one instruction; therefore, it is necessary to write a label using natural language expressions. Programming language statements should not be used.
- If abbreviations are used for variables, the comment symbol should be used to describe them in detail.
- Try to use only one page for the diagram; if this is not possible, it is advisable to use numbered connectors and also number the pages.

Advantages of using flowcharts

Joyanes(1992) points out some advantages of using flowcharts for algorithm design:

- They facilitate understanding of the algorithm
- They aid documentation
- The use of standard symbols makes it easier to find equivalent statements in programming languages
- They simplify the review, testing, and debugging of the algorithm

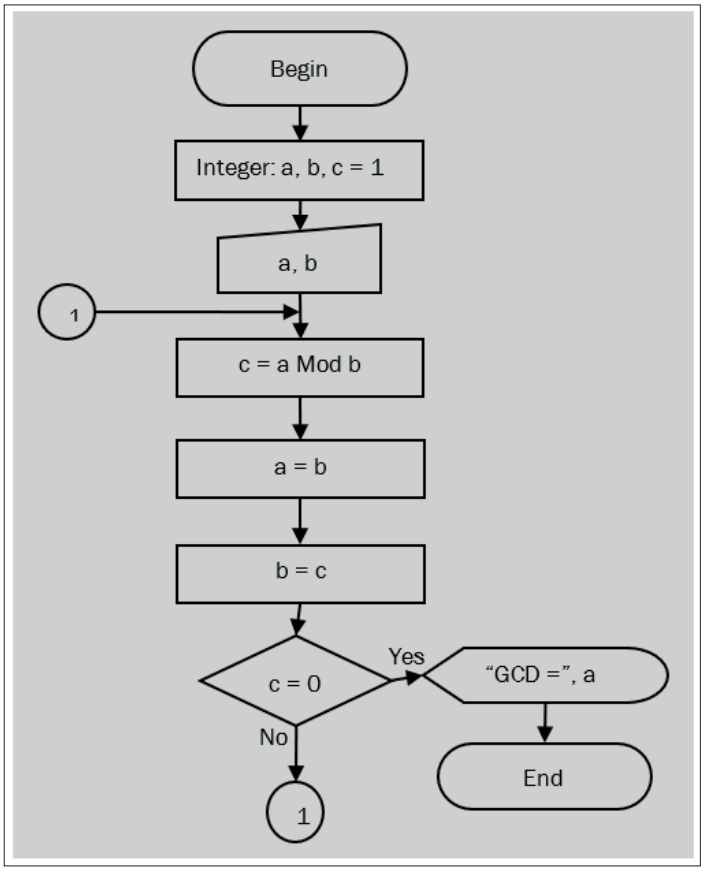
Figure 6 shows the Euclidean algorithm in flowchart notation

3.3.4 Nassi-Shneiderman Diagram

This notation was proposed by Isaac Nassi and Ben Shneiderman[5] in 1973 as an alternative to flowcharts for applying structured programming. It is also known as Chapín diagram or simply N-S diagram.

5 Isaac Nassi is an American computer expert. He has served as the Director of Systems Development at Cisco and as the Director of Research at SAP America. He has also worked for Apple Computer. Ben Shneiderman is an American mathematician/physicist and computer scientist, a professor of computer science at the Human-Computer Interaction Laboratory at the University of Maryland. His research focuses on human-computer interaction. He defined Universal Usability with consideration for the diverse characteristics of users and the technology they use

Figure 6. *Flowchart for the Euclidean Algorithm*



An N-S diagram is constructed using pseudocode instructions and a reduced set of basic shapes corresponding to sequential, selective, and iterative programming structures. It does not use arrows to establish the flow of operations; instead, it places the boxes corresponding to the instructions one after another or inside one another, making the sequence, branching, or repetition in the execution of the algorithm evident.

Nassi and Shneiderman (1973) argue that the diagrams they designed offer the following advantages:

- Decision structures are easily visible and understandable
- Iterations are visible and well-defined

- The scope of local and global variables is evident in the diagram
- Control cannot be transferred arbitrarily
- Recursion is represented in a trivial form
- Diagrams can be adapted to the peculiarities of the programming language to be used

Unlike pseudocode and flowcharts, the N-S diagram provides a more structured view of the steps in the algorithm and thus facilitates not only the next step of coding but also understanding and learning.

A program is represented by a single diagram that includes all the operations required to solve the problem, starting with a rectangle labeled “begin” and ending with another labeled “end.” To connect one diagram to another, the word “process” and the name or number of the subprogram to connect are used. The name of the program or algorithm is placed outside the diagram. These diagrams detail the input and output data and the processes that are part of the algorithm (Alcalde and García, 1992).

In N-S, sequential structures such as variable declaration, assignment, reading and writing data, and invoking subroutines are written in a rectangle, as shown in Figure 7.

The selective structure **if - then - else - end if,** is represented by placing the condition in an inverted triangle, which is in turn inside a rectangle. The execution flow corresponding to the decision alternatives is placed to the left and right, respectively. The selective structure is shown in Figure 8.

Figure 7. *Sequence Structure in N-S Notation*

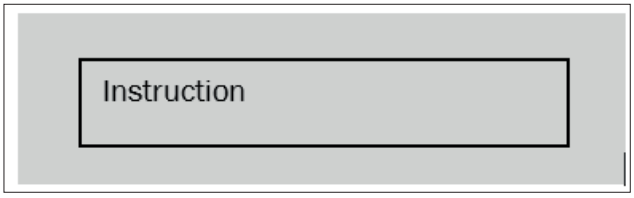
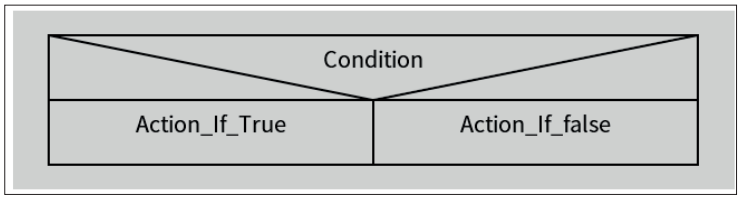
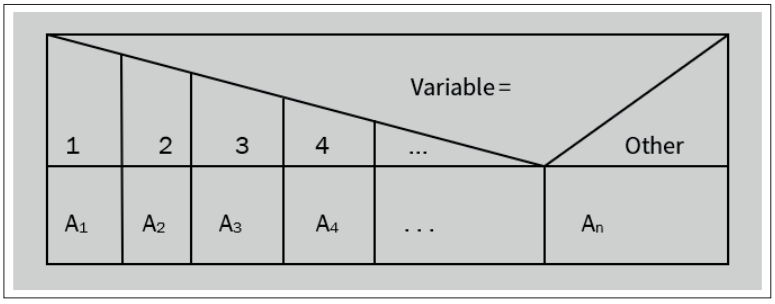


Figure 8. *Selective Structure in N-S Notation*



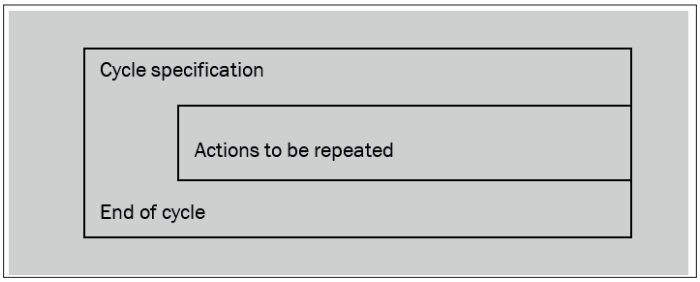
The multiple decision structure ***switch*** is represented by an inverted triangle inside a rectangle, with a rectangle below for each possible value of the variable. Figure 9 shows the relationship between the value of the variable and the action to be performed.

Figure 9. *Multiple Selection Structure in N-S Notation*



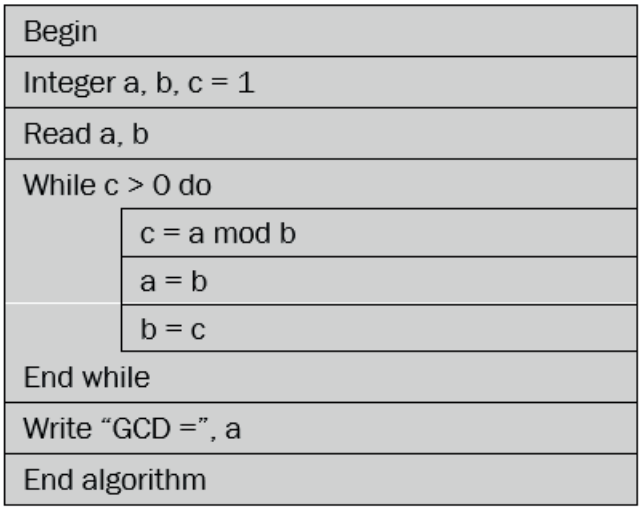
Iterative structures are represented by writing the cycle specification in one rectangle and the repeating actions in a second rectangle placed inside the first one. In the original document, the authors propose three ways to represent iterative structures: in the *while cycle*, the inner rectangle is placed at the bottom; in the *for cycle,* it is centered; and for the *repeat until cycle*, it is at the top, leaving space below for the cycle specification. In this book, to facilitate the design and interpretation of algorithms using this notation, a single representation is proposed, with the inner rectangle centered, leaving space above and below for the cycle specification, as shown in Figure 10.

Figure 10. *Iterative Structure in N-S Notation*



As an example of the Nassi-Shneiderman notation, Figure 11 presents the Euclidean algorithm to obtain the GCD.

Figure 11. *Euclidean Algorithm in N-S Notation*



3.3.5 Functional Notation

In this case, the term function indicates a rule of correspondence that associates an element from the domain set with another from the target set, as understood in mathematics; it does not refer to a subprogram as understood in imperative programming.

A function is specified by a name, its domain, range, and rule of correspondence. Let f be a function and x a value from its domain; the notation f(x) represents the ele-

ment y from the range that f associates with x. In other words, f(x) = y. The expression f(x) is known as the application of the function (Galve et al., 1993).

If f(x,y) is the Greatest Common Divisor function of two integers. The solution for the GCD of two numbers is shown in Table 11 using the Euclidean algorithm represented in functional notation.

Table 11. *Euclidean Algorithm in Functional Notation*

$$f(x,y) = \begin{cases} v & \leftarrow \text{if } (x \; Mod \; v) = 0 \\ f(y, x \; Mod \; y) & \leftarrow \text{if } (x \; Mod \; y) > 0 \end{cases}$$

The notations presented—pseudocode, flowchart, Nassi-Shneiderman diagram, and functional notation—are the most important, but there are many more. On this subject, Santos, Patiño and Carrasco (2006: 15) present, under the concept of "Techniques for Designing Algorithms," others such as flowcharts, organizational charts, decision tables, state transition diagrams, and Jackson diagrams.

3.4 STRATEGY FOR DESIGNING ALGORITHMS

To design an algorithm, it is essential to know the domain of the problem and be clear about what is expected from the algorithm. The analysis phase should provide an accurate description of the data with which the algorithm works, the operations to be carried out, and the expected results.

A computer program automatically carries out a series of operations that take a set of input data, performs the previously defined processes, and generates one or more outputs. An algorithm is the specification of such a series of operations; in other words, it defines the operations that will later be performed automatically. Consequently, to design an algorithm, it is necessary to know the operations required to achieve the expected result.

In this order of ideas, before attempting to express the solution to the problem in the form of an algorithm, it is advisable to take a set of input data, perform opera-

tions, and obtain the results manually. This exercise will help identify all the data, separate the inputs from the outputs, and establish formulas for the development of calculations that will yield the expected results.

Example 2. Even or Odd Number

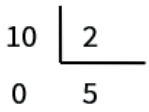
This example is about applying the strategy to design an algorithm to determine whether a number is even or odd.

As explained, before trying to write an algorithm, it is necessary to consider how to solve this problem for a particular case, without thinking about algorithms or programs.

First, establish the concept of an even number: an even number is one that, when divided by two, results in an integer with a remainder of zero.

Now, suppose the number in question is 10. How do you confirm that it is an even number?

If you think a bit, you can find two ways to do it, both involving integer division by two.



The first method consists of taking the quotient and multiplying it by two; if the result is equal to the number, which in this case is 10, you confirm that the number is even; otherwise, it is odd.

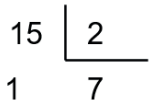
10 / 2 = 5

5 * 2 = 10 => confirmed, 10 is an even number

The second alternative is to take the remainder; if it is equal to zero, it means that the integer division is exact, confirming that the number is even; on the contrary, if the remainder is equal to one, the number is odd. To obtain the remainder, the Mod operator is used as presented in Section 2.3.1.

10 Mod 2 = 0 => confirmed, 10 is an even number

Similarly, test another number, such as 15.



Using the first method, we have:

15 / 2 = 7

7 * 2 = 14 => 15 is an odd number
Using the second method:

15 Mod 2 = 1 => 15 is an odd number

From these two examples, it follows that to solve this problem, an input is used, corresponding to the number to be evaluated, the process consists of an integer division from which the quotient or the remainder is taken depending on the programmer's preference, and the expected result is a message: "even number" or "odd number".

Now, after solving two particular cases of this problem, you can proceed to express the solution algorithmically using any of the notations explained in Section 3.3. Table 12 presents the solution in pseudocode.

Table 12. *Pseudocode for Even/Odd Number Algorithm*

```
1    Begin
2        Integer: n, r
3        Read n
4        r = n Mod 2
5        If r = 0 then
6            Write n, "is an even number"
7        Else
8            Write n, "is an odd number"
9        End if
10   End algorithm
```

In this algorithm, two variables are declared: the first to store the number to be evaluated, and the second to store the remainder of the integer division (line 4). To evaluate the content of variable r, the selective structure is used as explained in Section 4.2.2.

Figures 12 and 13 show this same algorithm through flowcharts and N-S diagrams, respectively.

Figure 12. *Flowchart of the Even/Odd Number Algorithm*

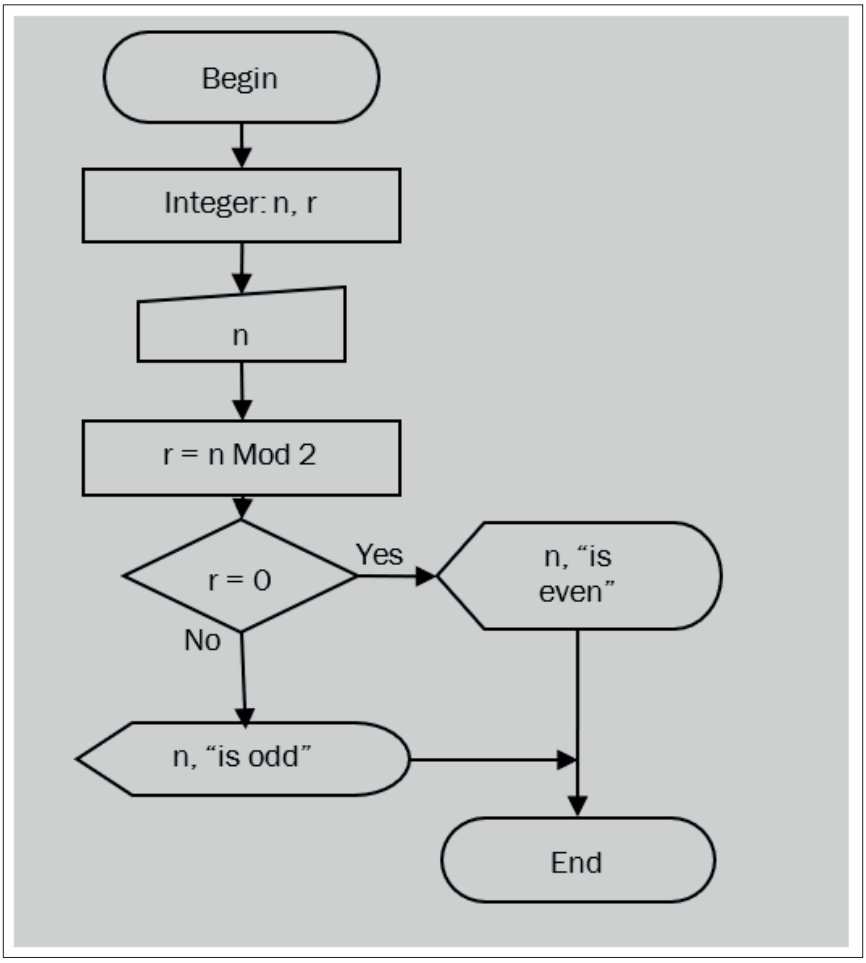
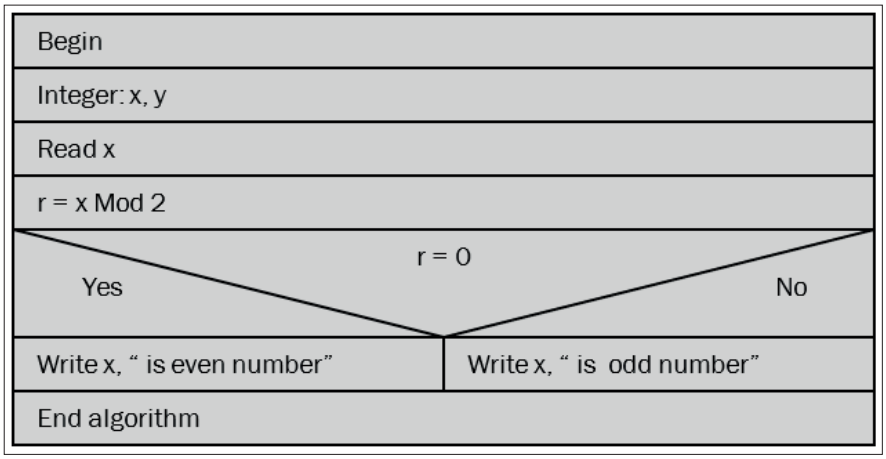


Figure 13. *Nassi-Shneiderman Diagram of the Even/Odd Number Algorithm.*



3.5 VERIFICATION OF ALGORITHMS

After designing the solution of a problem using any of the methods discussed earlier, it is necessary to verify that the algorithm works properly; that is, that it has the essential characteristics to be considered a good algorithm, such as being finite, defined, precise, correct and efficient. Algorithm verification is also known as des checking.

To verify the correctness of an algorithm, a table is created with a column for each variable or constant used, along with some additional columns to record executions, iterations, and outputs. The algorithm is then executed step by step, and the values of the variables are recorded in the table. By the end of the algorithm, it can be observed how the input data or initial values transform to generate the output data. If the process is correct and the outputs are as expected, the algorithm is deemed correct. If any of the data is not as expected, the design is reviewed, and necessary changes are made until it works properly.

Applying the above, we take the algorithm from example 2 and verify it three times with different numbers. In the first execution, the number 6 is entered; the division by 2 yields a remainder of 0, resulting in the message: "6 is an even number," which is correct. In the second execution, the number 3 is entered; upon division, a remainder of 1 Is obtained, leading to the message: "3 is an odd number." In the third execution, the number 10 is entered, the remainder of the integer division is 0, yielding

the result: “10 is an even number”.
Table 13 presents the verification performed on the algorithm from example 2, which reads a number and displays a message indicating whether it is even or odd.

Table 13. *Verification of the Even/Odd Number Algorithm*

Execution	n	R	Result
1	6	0	6 is an even number
2	3	1	3 is an odd number
3	10	0	10 is an even number

4. PROGRAMMING STRUCTURES

Knowing that a problem
can be solved in theory
by a computer
is not enough to tell us
whether it is practical to do so.
Baase and Van Gelder

Structured programming uses three types of structures: sequential, which are run one after the other in the order they are written; decision, which allows skipping parts of the code or selecting the execution flow from two or more alternatives; and iterative, which are used to repeat the execution of a certain part of the program.

4.1 SEQUENTIAL STRUCTURES

These structures are characterized by running one after another in the order they appear in the algorithm or program, such as assigning a data to a variable, reading data, or printing a data.

4.1.1 Assignment

This operation consists of saving a data in a given location in the reserved memory through the declaration of a variable.

Assignment is a relevant operation in the imperative programming paradigm, where all data, read and obtained as a result of a calculation, is stored in memory awaiting further instructions.

Example:

integer a, b, c
string: operation
a = 10
b = 15

c = a + b
operation = "sum"

The assignment of variables is discussed in more detail in Section 2.2.4.

4.1.2 Data Input

A program operates on a set of data provided by the user or retrieved from a storage device. Similarly, the algorithm requires that data be provided from which it will obtain the expected results.

Reading data consists of taking data from an external medium or a file and loading it into memory where it can be processed by the program (Joyanes, 1988). The default device is the keyboard.

For reading data, the "read" instruction is used in pseudocode and its equivalents in different notations.

In pseudocode, the "read" instruction has the following syntax

Read < list of variable identifiers>

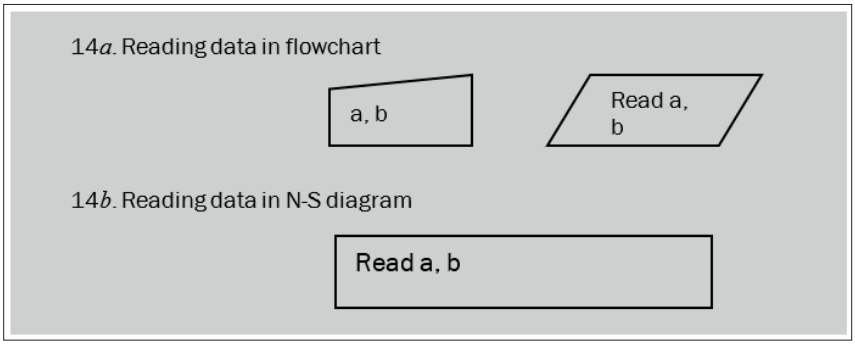
Example:

Read a, b

Where a and b are the variables that will receive the values and therefore must be declared in advance.

In flowcharts, the read instruction can be represented in two ways: using the keyboard reading symbol or through an input and output process. In N-S diagrams, all sequential structures are written inside a box, so that the reading of data is represented by the word read inside a box, as shown in Figure 14.

Figure 14. *Data Input Symbols*



Any of the two symbols in Figure 14a is valid for indicating a data read in a flowchart. Figure 14b is exclusive for N-S diagrams.

4.1.3 Data Output

Every program performs one or more tasks and generates one or more data as a result. The results are presented to the user using instructions and data output devices such as the screen or printer.

To send information from the computer memory to an output device, the pseudocode instruction is used as "write" and its equivalents in other notations.

In pseudocode data reading is written as:

Write < list of constants and variables >

Example:

Write a, b

Where a and b are previously defined variables

Write "This is a message"

When more than one variable is written, it is necessary to separate them by commas (,) and messages are written in double quotes "". If a variable is written in quotes the identifier will be shown and not the content.

Example:

String: name
Integer: age
Name = "Juan"
Age = 25

The correct way to display data is:

Write "Name: ", name
Write "Age: ", age

For which the result will be:

Name: Juan
Age: 25

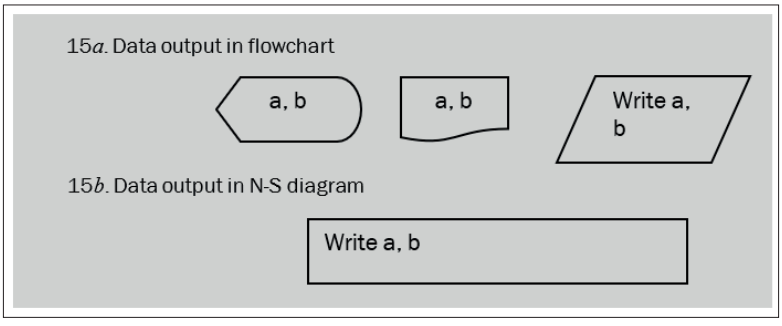
Writing in the form:

Write "name"
Write "age"

It will only show the labels: "name" and "age"

In a flowchart data output can be represented by three symbols: screen output, printer output, and an input/output process (Figure 15a). In N-S diagrams, the "write" instruction is placed inside a box as shown in figure 15b.

Figure 15. *Data Output symbols*



4.1.4 Examples with Sequential Structures

Example 3. Adding Two Numbers

This algorithm reads two numbers, stores them in variables, sums them, and displays the result. It aims to represent algorithmically the process of adding two values. For example:

$3 + 2 = 5$

If this operation is converted into an arithmetic expression with variables, it becomes:

$r = x + y$

By assigning or reading values for the variables x and y, we can compute the result. For example:

$x = 6$
$y = 7$

$r = x + y$
$r = 6 + 7$
$r = 13$

From this example, it follows that the algorithm requires two inputs, which are stored in the variables x and y. Using these, the expression (process) is developed, and the result is the output of interest to the user, therefore, this is the output data.

Input: x, y
Process: r = x + y
Output: r

Now that the general solution to the problem is understood, it can be expressed algorithmically. Table 14 presents the algorithm in pseudocode notation, and Figures 16 and 17 show flowcharts and N-S diagrams, respectively.

Table 14. *Pseudocode for the Algorithm to Add Two Numbers*

```
1    Begin
2        Integer: x, y, r
3        Read x, y
4        r = x + y
5    End algorithm
```

Figure 16. *Flowchart for Adding Two Numbers*

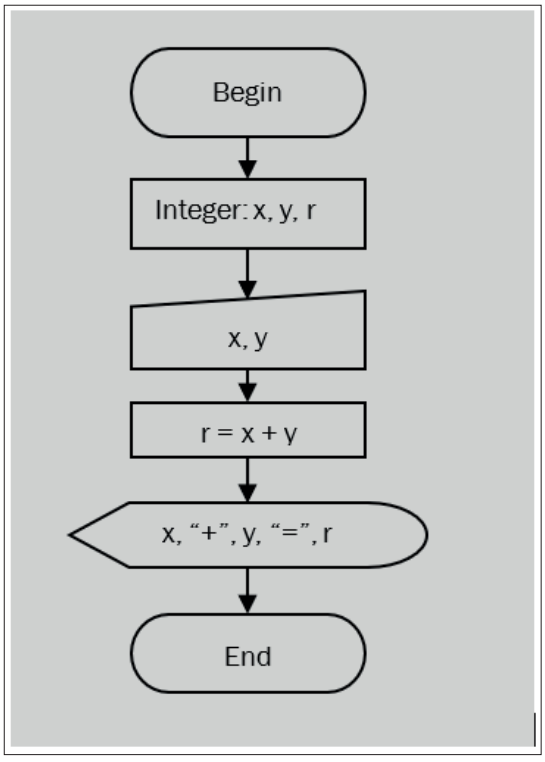


Figure 17. *N-S Diagram for Adding Two Numbers*

Begin
Integer: x, y, r
Read x, y
r = x + y
Write x, " + ", y, " = ", r
End algorithm

In this algorithm, three variables are declared: the first two (x, y) correspond to the two input data, and the third (r) saves the result of the process. The output for this exercise is undoubtedly the result of the operation; however, in many programs, it is also necessary to show some of the input data for the output to be more understandable, in this example, the operands and the result of the operation are shown.

To verify whether this algorithm is correct, a table like the one shown in Table 15 is created, and the algorithm is run line by line, providing the input data and performing the operations. The data is written in the columns of the table labeled with the variables and then checked if the output is correct. (The algorithm verification is explained in Section 3.5).

Table 15. *Verification of the Algorithm for Adding Two Numbers*

Execution	X	y	r	Output
1	3	4	7	3 + 4 = 7
2	5	8	13	5 + 8 = 13
3	2	3	5	2 + 3 = 5

It is important to remember that, when verifying the algorithm, the steps must be strictly followed and not done from memory, as what is in mind and what is written may not be the same, and what we want to verify is what is written.

Example 4. The Square of a Number.

The square of a number is equal to multiplying the number by itself, as shown:

*32 = 3 * 3 = 9*
*52 = 5 * 5 = 25*

Now, to generalize the operation, a variable is used for the number that needs to be squared, resulting in an expression of the form:

*num2 = num * num = ?*

If we take this process into an algorithm and then into a program, the computer will be able to provide the result for any number. However, before proceeding to design the algorithm, the information is organized in terms of input, output, and process.

Input: number
Output: the square of the number
*Process: square = number * number*

This information is used to design the solution of the problem. Table 16 shows the pseudocode, and Figures 18 and 19 show flowcharts and N-S diagrams.

Table 16. *Pseudocode to Calculate the Square of a Number*

```
1   Begin
2       Integer: num, square
3       Read num
4       square = num * num
5       Write square
6   End algorithm
```

In this algorithm, two variables are declared: ***num*** to store the number entered by the user and ***square*** to store the result of the process, that is, the square of the number. Table 17 shows three test runs to verify the correctness of the algorithm: in the first one, the number 3 is entered and the result is 9; in the second one, the number 2 is entered and the result is 4; and in the third one, the number 7 is entered and the result is 49. These results confirm that the algorithm is correct.

Table 17. *Verification of the Algorithm for the Square of a Number*

Execution	num	square	Output
1	3	9	9
2	2	4	4
3	7	49	49

Figure 18. *Flowchart to Calculate the Square of a Number*

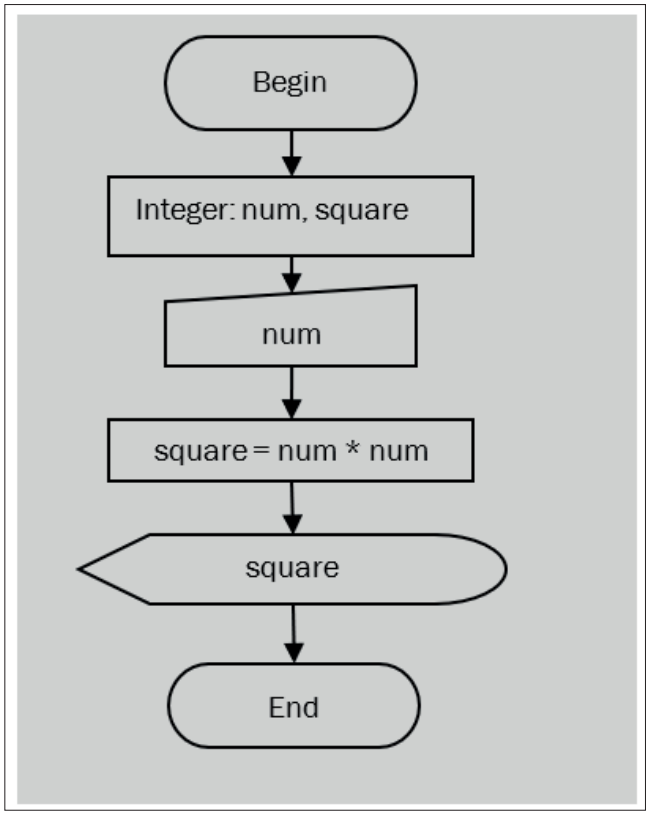


Figure 19. *N-S Diagram to Calculate the Square of a Number*

Begin
Integer: num, square
Read num
square = num * num
Write square
End algorithm

Example 5. Selling Price of a Product

Consider the following case: a liquor store purchases its products by the case and sells them by the unit. Given that a case of any product has a cost c and contains n units, it is desired to calculate the price p for each unit, ensuring a 30% profit.

The first step is to understand the problem, that is, to be able to obtain the results and solve the problem manually.

Example 1: A case of rum is purchased for $240,000,oo (c), containing 24 units (n), and the goal is to achieve a 30% profit. What price (p) should each unit be sold for?

To solve this problem, three calculations are necessary: first, apply the profit percentage to the cost of the case; second, add the cost and the profit to obtain the total price of the case; and third, divide the total value by the number of units (n) to get the unit price (p).

First:

*profit = 240,000,oo * 30/100*
profit = $ 72,000,oo

Second:

total = $ 240,000,oo + $ 72,000,oo
total = $ 312,000,oo

Third:

p = $ 312,000,oo / 24
p = $ 13,000,oo

Finally, the selling price of each unit is $13.000.oo

Example 2: Suppose another product is purchased, with a case cost of $600,000,oo and containing 12 units. Then we have:

First:

*profit = $ 600,000,oo * 30/100*
profit = $ 180,000,oo

Second:

total = $ 600,000,oo + $ 180,000,oo
total = $ 780,000,oo

Third:

p = $ 780,000,oo / 12
p = $ 65,000,oo

The selling price of each unit is $65,000,oo

These two examples help to understand how the problem is solved. Now, it is necessary to generalize the solution process to specify an algorithm. For this, we represent the values with variables and the calculations with formulas applied on these variables, as follows:

Let:

c = cost of the case
n = number of units
p = selling price per unit

From these data: the first two are inputs, while the last one is the output. The calculations to perform are:

*profit = c * 30/100*
total = c + profit
p = total / n

Having reached this point, we can now design an algorithm to solve any case of this problem. In Table 18, the pseudocode is shown; in Figure 20, the flowchart; and in Figure 21, the N-S diagram.

Table 18. *Pseudocode to Calculate the Unit Price of a Product*

```
1  Begin
2      Integer: n
       Real: c, profit, total, p
3      Read c, n
4      profit = c * 30/100
5      total = c + profit
6      p = total / n
7      Write “Price = ”, p
8  End algorithm
```

Figure 20. *Flowchart to Calculate the Unit Price of a Product*

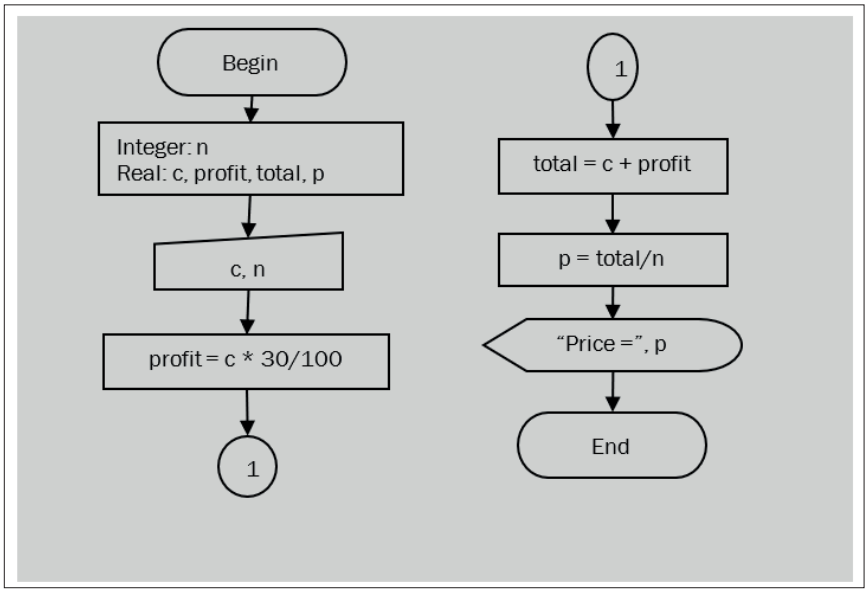


Figure 21. *N-S Diagram to Calculate the Unit Price of a Product*

Begin
Integer: n Real: c, profit, total, p
Read c, n
profit = c * 30/100
total = c + profit
p = total / n
Write "Price = ", p
End algorithm

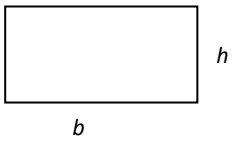
The data from three tests conducted to verify the correctness of this algorithm are shown in Table 19. The first two columns correspond to the input data, and the last one to the output generated by the algorithm.

Table 19. *Verification of the Algorithm to Calculate the Unit Price of a Product*

C	n	Profit	total	P	Output
240,000	24	72,000	312,000	13,000	Price = 13,000
600,000	12	180,000	780,000	65,000	Price = 65,000
300,000	10	90,000	390,000	39,000	Price = 39,000

Example 6. Area and Perimeter of a Rectangle

An algorithm is needed to calculate the area and perimeter of a rectangle of any dimension.



To solve this problem, it is necessary to know the formulas for obtaining both the area and the perimeter of a rectangle.

Let:

b = base
h = height

The formulas to use are:

*Area = b * h*
*Perimeter = 2 * (b + h)*

If the rectangle has a base of 10 cm and a height of 5 cm, we obtain:

*Area = 10 * 5*
Area = 50

*Perimeter = 2 * (10 + 5)*
Perimeter = 30

As seen in the example, to perform the calculations, it is necessary to have the base and height, which correspond to the data that the user must provide and that the algorithm must read. Meanwhile, the area and perimeter are the data that the algorithm must calculate and present to the user; therefore:

Input data: base (b) and height (h)
Output data: area (a) and perimeter (p)
Processes:
*a = b * h*
*p = 2 * (b + h)*

The design of the solution in pseudocode notation is presented in Table 20.

Table 20. *Pseudocode to Calculate the Area and Perimeter of a Rectangle*

```
1    Begin
2        Integer: b, h, a, p
3        Read b, h
4        a = b * h
5        p = 2 (b + h)
6        Write "area:", a
7        Write "perimeter:", p
8    End algorithm
```

To verify that the algorithm is correct, we perform a step-by-step desk check as follows:

Step 2. Declare variables: b, h, a, p corresponding to: base, height, area, and perimeter, respectively.

Step 3. Read the base and height from the keyboard and store them in variables b and h. Assume the values are 5 and 8

Step 4. Calculate the area, 5 * 8 = 40 (a = b * h) and store it in variable a

Step 5. Calculate the perimeter, 2 * (5 + 8) = 26 and store it in variable p

Steps 6 and 7. Display the contents of variables a and p with their respective messages.

The results of the verification with three sets of data are presented in Table 21.

Table 21. *Verification of the Algorithm for Area and Perimeter of a Rectangle*

Execution	b	h	a	p	Output
1	5	8	40	26	area: 40 perimeter: 26
2	3	4	12	14	area: 12 perimeter: 14
3	8	4	32	24	area: 32 perimeter: 24

Example 7. Time Dedicated to a Subject

It is known that a university professor hired on an hourly basis dedicates one hour to preparing a two-hour class, and for every four hours of classes taught, the professor conducts and evaluation that takes them two hours to grade. If the professor is assigned a course of ***n*** hours per semester, how many hours will the professor work in total?

The best way to understand a problem is to take a specific case and develop it. Assume the professor has been hired for a mathematics course with a total number of 64 hours (nh) per semester.

If for every two hours of class, the professor dedicates one hour to preparation, then the preparation time (pt) is:

pt = 64 / 2
pt = 32 hours

In general: pt = nh/2

It is also known that the professor conducts one evaluation for every four hours of class, so we can calculate the number of evaluations (ne) the professor will perform during the semester:

ne = 64 / 4
ne = 16

For any case: ne = nh / 4

Now, if it takes two hours to grade each evaluation, the total grading time (gt) for the semester is:

*gt = 16 * 2*
gt = 32

*Using variables: gt = ne * 2*

Finally, to obtain the total time (tt) that the professor dedicates to the mathematics course, we simply sum the preparation time, the time spent teaching classes, and the grading time:

tt = 32 + 64 + 32
tt = 128

That is: tt = pt + nh + gt

Thus, for a course of 64 class hours, the professor dedicates 128 hours of work. To calculate this value for any course, it is necessary to design an algorithm that takes the number of semester class hours as input and returns the total dedicated hours.

From this, we can deduce that:

Input: number of class hours (nh)
Output: total time dedicated (tt)
Calculations to perform:
pt = nh/2
ne = nh / 4
*gt = ne * 2*
tt = pt + nh + gt

The algorithm in flowchart notation is presented in Figure 22.

To ensure that the algorithm performs the calculations correctly, we test it with three different cases. The results are presented in Table 22.

Figure 22. *Flowchart to Calculate the Time Dedicated to a Subject*

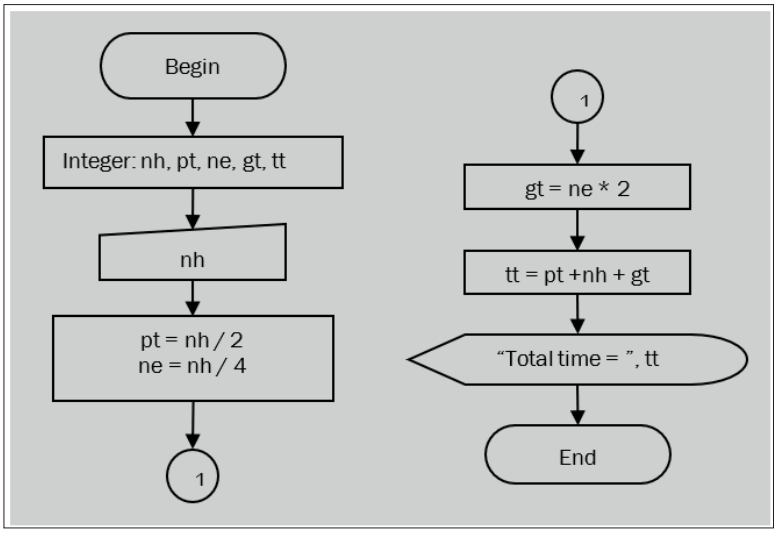


Table 22. *Verification of the Algorithm for Time Dedicated to a Subject*

Exec.	nh	pt	ne	gt	tt	Output
1	64	32	16	32	128	Total time = 128
2	80	40	20	40	160	Total time = 160
3	96	48	24	48	192	Total time = 192

4.1.5 Proposed Exercises

Design algorithms to solve the following problems and represent them using pseudocode, flowcharts, or N-S diagrams.

1. Read three grades and calculate the average
2. Calculate the area of a triangle
3. Calculate the area and perimeter of a circle
4. Calculate the volume and surface area of a cylinder
5. Given the width, length, and height of a box, calculate the volume and the amount of paper (in cm^2) needed to cover it.
6. Read an integer and separate its digits into: thousands, hundreds, tens, and units.
7. Calculate the interest and future value of an investment with simple interest.
8. Calculate the interest and future value of an investment with compound interest.
9. If the university finances student tuition in four equal monthly installments with a 2% interest on the balance, given the tuition amount and the number of installments, a student wants to know: What will be the value of each installment? How much will they pay in total?
10. A salesperson receives a base salary plus 10% commission on their sales. If in any given month they make three sales with values: v1, v2, and v3, how much will they receive in commission? And how much in total?
11. A customer at a supermarket purchases n products at a unit price p. If the

product has a 10% discount during the season, which is applied at checkout, what is the value of the discount? How much will they need to pay?

12. A student wants to know their final grade in Programming. This grade consists of the average of three partial grades. Each partial grade is obtained from a workshop, a theoretical evaluation, and a practical evaluation. Workshops count for 25% of the partial grade, theoretical evaluations for 35%, and practical evaluations for 40%.

13. A traveler wants to know how many dollars they will receive for their capital in pesos.

14. Bill for electricity service. The monthly consumption is determined by the difference in readings.

15. A company's profits are distributed among three partners as follows: Partner A = 40%, Partner B = 25%, Partner C = 35%. Given a sum of money, how much will each receive?

16. A store owner buys an item for x pesos and wants to achieve a 30% profit. What will be the selling price of the item?

17. A store aims for a 30% profit on each item and offers a 15% discount on the selling price during the season for all its products. Given the cost of a product, calculate the selling price, ensuring that the discount can be applied to the selling price while still achieving a 30% profit on the cost.

18. Three individuals decide to invest their money to start a business. Each invests a different amount. Calculate the percentage each contributes relative to the total amount invested.

19. Calculate the salary of an employee who has worked n daytime overtime hours and m nighttime overtime hours, with daytime overtime increasing by 25% over regular hours and nighttime overtime by 35%.

20. A student wants to know the minimum grade they must obtain on the final evaluation in calculus after knowing their two partial grades, knowing that the subject is passed with a 3.0, and the final grade is calculated as follows: 30% for each partial and 40% for the final.

21. Given the amount of a loan, the time, and the amount paid in interest, calculate the interest rate applied.

22. Knowing the time an athlete takes to complete a lap around the stadium (400 m), estimate the time it will take to cover the 12 km required for a competition.

23. Solve a quadratic equation ($aX^2 + bX + c = 0$) considering that a, b, and c are integer values that can be positive or negative.

24. Given the amount a customer pays for a product, calculate what portion corresponds to the cost of the product and how much to VAT. Considering that the VAT percentage can vary over time and from product to product, this data is read from the keyboard.

25. A teacher designs a questionnaire with n questions and estimates that it takes m minutes to grade each question. If the questionnaire is applied to x students, how much time (in hours and minutes) will be needed to grade all the exams?

4.2 DECISION STRUCTURES

In programming, as in real life, it is necessary to evaluate circumstances and act accordingly, as the course of actions cannot always be predetermined. During the execution of a program, many expressions will be subject to certain conditions; therefore, the programmer must identify these situations and specify what to do in each case (Timarán ***et al***, 2009).

Every problem, whether in programming or other fields, includes a set of variables that can take different values. The presence or absence of certain variables, as well as their behavior, require different actions in the solution. This means that programs are not simple sequences of instructions, but complex structures that implement jumps and branches based on the values of the variables and the conditions defined on them.

For example, to perform a division, two variables are needed: dividend and divisor, which can hold any integer or real number. However, division by zero is undefined, so the condition is that the divisor must be different from zero. When implementing this operation in a program, the restriction must be checked, and it must be decided whether to perform the calculation or display an error message. If this decision is not implemented and the variable takes the value of zero during execution, a calculation error will occur, and the execution will be interrupted.

Implementing a decision involves evaluating one or more variables to determine whether they meet one or more conditions and establishing a flow of actions for each possible outcome. Actions may consist of executing one or more expressions, skipping them, or selecting a sequence of instructions from two or more available alternatives.

4.2.1 Condition

In this section, the term "condition" is mentioned repeatedly, making it appropriate to clarify its meaning. A condition is a relational or logical expression that may or may not hold true depending on the values of the variables involved in the expression. When using a relational expression, the condition is a comparison between two variables of the same type or a variable and a constant, while a logical expression consists of one or more relational expressions combined with logical operators: AND, OR, NOT; whose result depends on the truth table of the corresponding operator.

Here are examples of conditions that contain relational expressions:

x = 5
y <= 10
x > y

When writing the program, the condition is specified using expressions like these, but when it is executed, they are evaluated, and the result can be true or false depending on the value of the variables. Suppose that:

x = 8
y = 9

In this case, condition (a) is not met, yielding a false result; condition (b) is met, producing a true result; and condition (c) also returns false.

Here are some examples of conditions formed by logical expressions:

x > 0 ***AND*** *x < 10*
x == 5 ***OR*** *y == 9*
NOT*(x>y)*

Condition (a) returns *true* if both relational expressions are true; condition (b) returns *true* if at least one of the relational expressions is true, and condition (c) negates the result of the relational expression within the parentheses. If the variables have the previously indicated values (x = 8, y = 9) condition (a) is *true*, condition (b) is *true*, and condition (c) is *true*.

4.2.2 Types of Decisions

Decisions can be of three types: simple, double, and multiple.

Simple Decision: This involves deciding whether to execute or skip an instruction or a set of instructions. In this case, it is determined what the program should do if the condition is true, but if it is false, control simply passes to the instruction following the decision structure.

An example of a simple decision arises when calculating the absolute value of a number. The absolute value of a number is the same as the number itself, but if it is negative, it must be multiplied by -1 to change its sign. Note that this operation is only necessary for negative numbers. Simple decisions in an algorithm use the IF statement.

Double Decision: This occurs when there are two alternatives for execution, and depending on the result of evaluating the condition, one or the other is executed. If the condition is true, the first instruction or block of instructions is executed; if false, the second is executed.
An example of this type of decision is evaluating a grade to determine whether a student passes or fails a subject, depending on whether the grade is greater than or equal to 3.0.

Multiple Decision: This involves evaluating the content of a variable and executing a sequence of actions based on its value. In this case, the set of values for the variable must be greater than two, and only equality conditions are evaluated. Each possible value is associated with a sequence of execution, and these are mutually exclusive. This type of decision is implemented using the *SWITCH* statement.

As an example of a multiple decision, consider an algorithm that allows executing any of the basic arithmetic operations: addition, subtraction, multiplication, and division, on a set of numbers. Upon execution, the user selects the operation, and the algorithm performs the corresponding operation.

4.2.3 IF Structure

This instruction evaluates a logical value, a relational expression, or a logical expression and returns a logical value, based on which a simple or double decision is made.

Simple Decision

As a simple decision, its pseudocode syntax is:

IF <condition> THEN
Instructions
END IF

Figures 23 and 24 present its representation in flowchart and N-S notation, respectively.

Figure 23. *Simple Decision in Flowchart Notation*

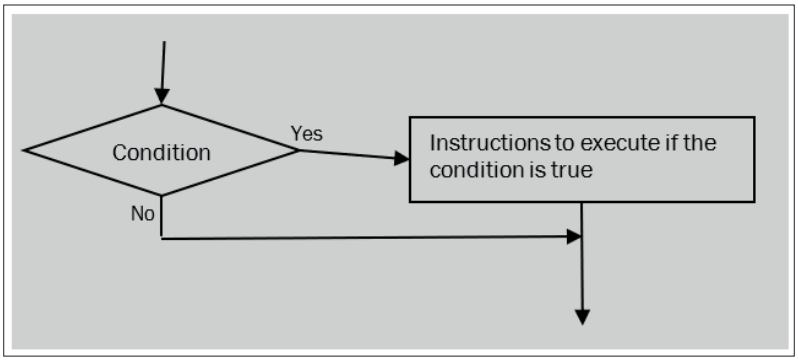
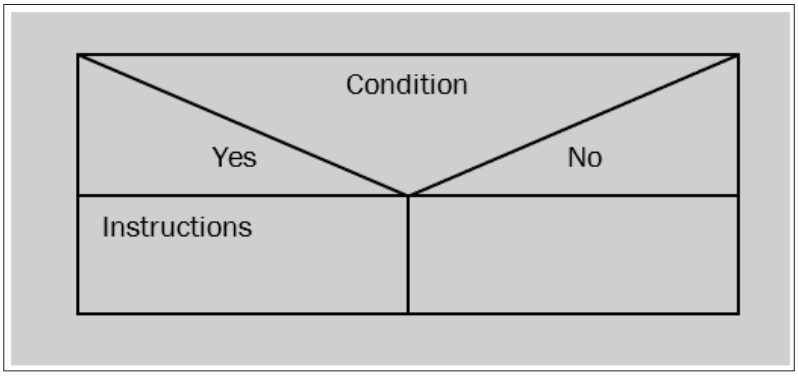


Figure 24. *Simple Decision in N-S Notation*



Example 8. Calculating the Absolute Value of a Number

The absolute value of a number is the same number for positives and the number with its sign changed for negatives; that is, the absolute value is the distance from 0 to the number, and since distances cannot be negative, it will always be positive.

$|5| = 5$
$|-3| = 3$

The pseudocode for this exercise is presented in Table 23.

Table 23. *Pseudocode to Calculate the Absolute Value of a Number*

```
1    Begin
2        Integer: num
3        Read num
4        IF num < 0 THEN
5            num = num * -1
6        END IF
7        Write "Absolute value = ", num
8    End algorithm
```

The decision structure is applied in line 4 to determine if the number is negative. If the condition evaluates to true, line 5 is executed; if it evaluates to false, the *IF* structure ends, and the instruction following *END IF* in line 7 is executed.

If the number 5 is entered when executing this algorithm, the decision condition will not be met, and thus the same number is written. If in a second execution the number -3 is entered, the condition evaluates to true, so the operation within the decision is executed: multiplying the number by -1, and finally, the number 3 is displayed. These results are shown in Table 24.

Table 24. *Verification of the Algorithm to Calculate Absolute Value*

Execution	num	Output
1	5	Absolute value = 5
2	-3	Absolute value = 3

Figures 25 and 26 present the flowchart and the N-S diagram, respectively.

Figure 25. *Flowchart for the Absolute Value of a Number*

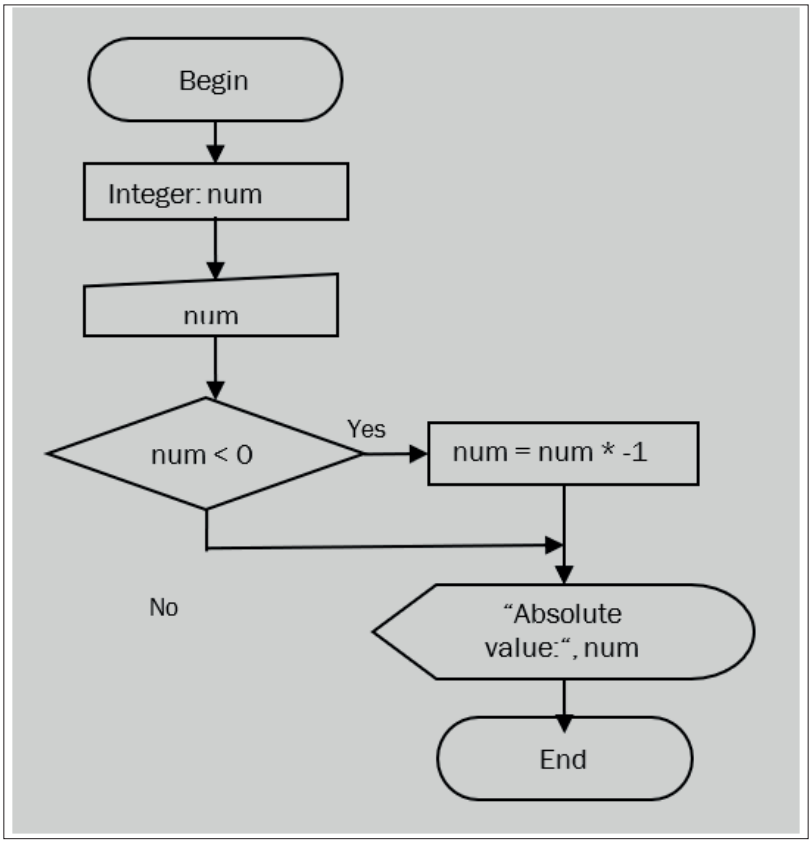
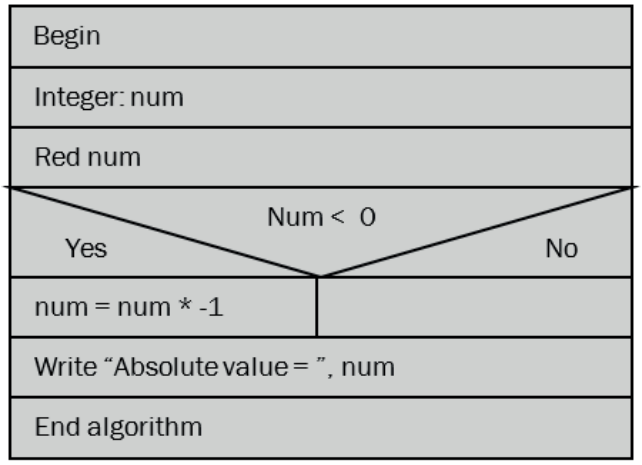


Figure 26. *N-S Diagram to Calculate the Absolute Value of a Number*



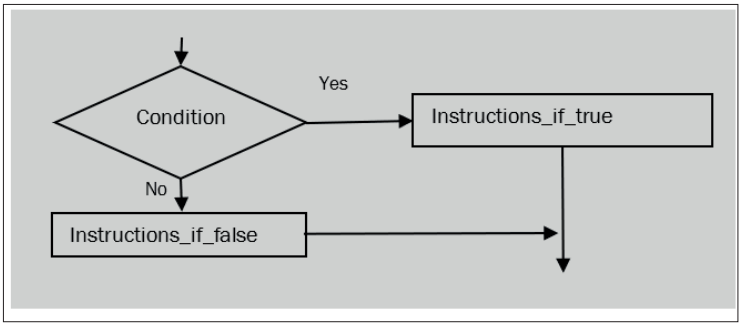
Double Decision

A double decision is implemented when there are two options to continue executing the algorithm, and these depend on a condition; that is, there is one or more instructions for when the condition is true and another set of instructions for when it is false. Simple decisions are used to include jumps in the execution of the algorithm or program, while double decisions allow for branching. Its syntax in pseudocode is:

```
IF <condition> THEN
        Instructions_if_true
ELSE
        Instructions_if_false
END IF
```

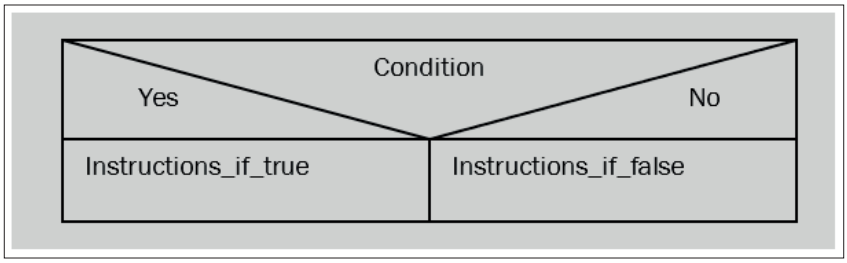
Figures 27 and 28 show its representation in flowchart and N-S notation, respectively.

Figure 27. *Double Decision in Flowchart Notation*



Note that in pseudocode, the instruction ***End If*** is used to indicate the extent of the conditional structure. In a flowchart, the end of the conditional is determined by the joining of the two paths, labeled ***Yes*** and ***No***. In the N-S diagram, the conditional structure has two blocks: the left block contains the instructions to be executed when the condition is met, and the right block contains the instructions to be executed when the condition is not met. The end of the conditional will be marked by the completion of the two blocks and the continuation into a single box.

Figure 28. *Simple Decision in N-S Diagram Notation*



Example 9. Division

As mentioned, before performing a division, it is necessary to verify that the divisor is not zero; otherwise, a computation error will occur, and the program will terminate. Therefore, it is necessary to use the ***If*** statement to decide whether to perform the division or display an error message.

The algorithm for performing a division, represented in pseudocode, is shown in Table 25. Line 4 specifies the condition that must be met to proceed with the division. If the condition is not met, execution jumps to the ***Else*** clause in line 7, and runs line 8, showing an error message.

Table 25. *Pseudocode for Division*

```
1     Begin
2         Integer: dividend, divisor, quotient
3         Read dividend, divisor
4         If divisor != 0 then
5             Quotient = dividend / divisor
6             Write Quotient
7         Else
8             Write “Error, divisor = 0”
9         End if
10    End algorithm
```

To test if the algorithm works correctly, it is executed step by step, a couple of numbers are entered, and the condition is verified along with the path taken depending on it. Three tests are shown in Table 26.

Table 26. *Verification of the Division Algorithm*

Exec.	Dividend	Divisor	Quotient	Output
1	15	3	5	5
2	8	0		Error, divisor = 0
3	12	3	4	4

The solution to this exercise in flowchart notation is shown in Figure 29 and in N-S diagram in Figure 30.

Example 10. Greater and Lesser Number

This exercise consists of reading two different integers and deciding which one is greater and which one is lesser.

To solve this exercise, it is necessary to declare two variables (x, y), input the numbers and store them in these variables, then decide if x is greater than y or the opposite.

Figure 29. *Flowchart for Division*

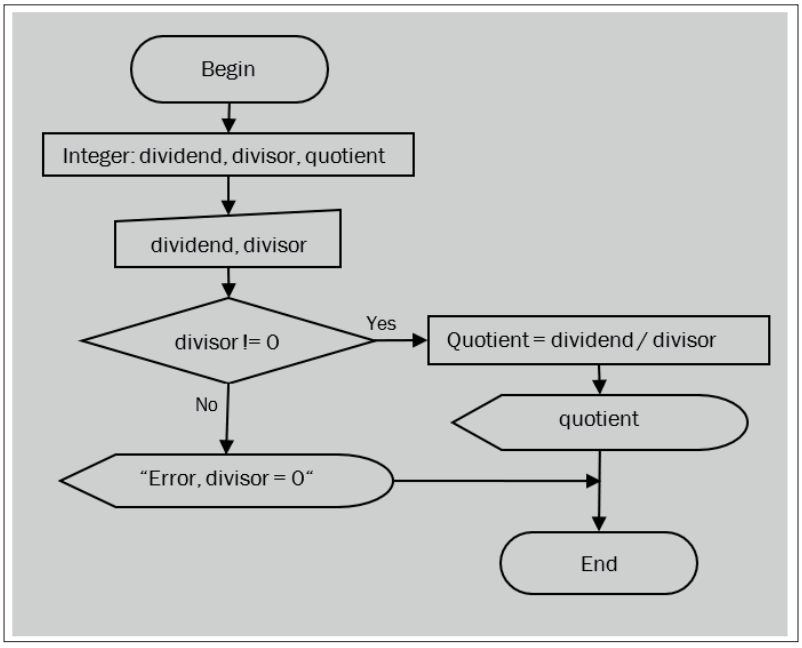
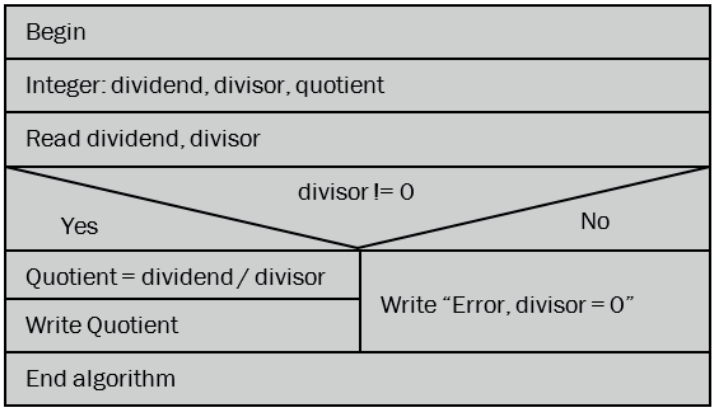


Figure 30. *N-S Diagram for Division*



The algorithm in example 10, in pseudocode notation, is shown in Table 27.

Table 27. *Pseudocode to Identify Greater and Lesser Number*

```
1     Begin
2         Integer: x, y
3         Read x, y
4         If x > y then
5             Write "Greater number: ", x
6             Write "Lesser Number", y
7         Else
8             Write "Greater number: ", y
9             Write "Lesser number", x
10        End if
11    End algorithm
```

Table 28 shows the behavior of the algorithm when tested with two data sets.

Table 28. *Verification of the Greater and Lesser Number Algorithm*

x	y	x > y	Output
23	12	True	Greater number: 23 Lesser number: 12
5	11	False	Greater number: 11 Lesser number: 5

Further examples are provided in section 4.2.6.

4.2.4 SWITCH Structure

Many decisions must be made not only between two alternatives but from a larger set. These cases can be effectively solved using nested double decisions; however, for the sake of clarity in the algorithm and ease for the programmer, it is better to use a multiple decision structure, which is easy to translate into a programming language, as these include some instruction for this purpose.

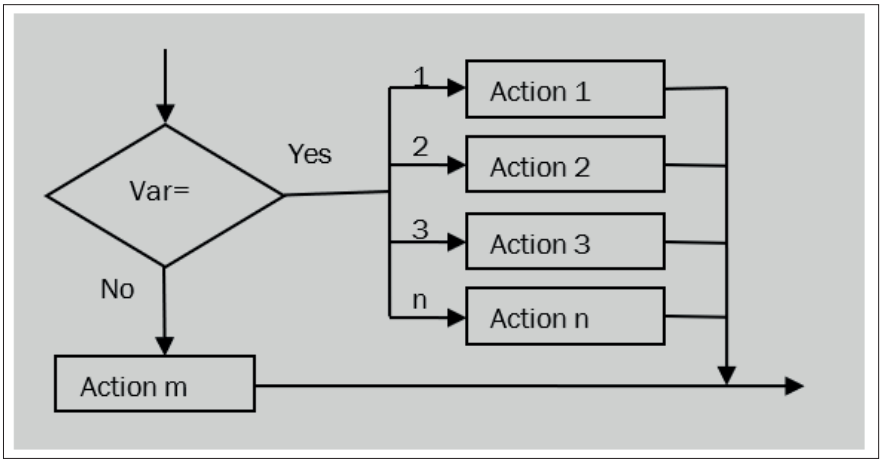
The ***Switch*** statement determines the value of a variable and, depending on this value, follows a course of action. It is important to note that only the condition of equality between the variable and the constant is verified. Based on Joyanes (1996), the following syntax is proposed:

Switch *<variable >* ***do***
    Value_1:
        Action 1
    Value_2:
        Action 2
    Value_3:
        Action 3
    Value_n
        Action n
    Else
        Action m
End switch

Since equality between the variable and constant is evaluated, all alternatives are mutually exclusive; this means that only one set of actions can be executed. If none of the alternatives are met due to no equality, the group of actions associated with the *Else* clause will be executed. This last part is optional; if it is not present and none of the cases are met, the execution of the algorithm will simply continue to the line following the *end switch*.

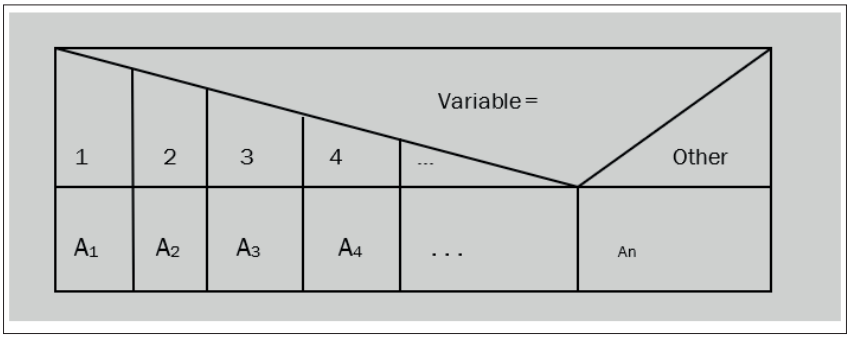
In a flowchart, the multiple decision is represented by a diamond where the variable is placed, and two outputs are shown: one marked with the word ***If,*** subdivided according to the possible values of the variable, and placed along with their corresponding flow; in the other output, the word ***No*** is written along with the actions to be taken if the value of the variable does not match any of the values established, as shown in Figure 31.

Figure 31. *Multiple Decision in Flowchart Notation*



In N-S diagram, it is represented by a box with an inverted triangle where the variable and the equal sign are placed, and the box is divided into as many sub-boxes as there are alternatives for the decision, including a box for the case where the variable takes a value different from those considered in the options, labeled as Other, as shown in Figure 32.

Figure 32. *Multiple Decision in N-S Notation*



Example 11. Roman Numerals

This algorithm reads an Arabic number and displays its Roman equivalent, but only considers the first 10 numbers, so if the entered number is greater than 10, it will display the message: “Invalid number”.

Solving exercises of this type is facilitated by using the multiple decision structure, as there is an equivalent for each value.

The pseudocode is shown in Table 29, which shows the multiple decision structure implemented between lines 4 and 17.

Table 29. *Pseudocode for Roman Numerals*

```
1     Begin
2         Integer: num
3         Read num
4         Switch num do
5             1: Write “I”
6             2: Write “II”
7             3: Write “III”
8             4: Write “IV”
9             5: Write “V”
10            6: Write “VI”
11            7: Write “VII”
12            8: Write “VIII”
13            9: Write “IX”
14            10: Write “X”
15        Else
16            Write “Invalid number”
17        End switch
18    End algorithm
```

In the pseudocode of this exercise, 10 possible values for the variable num are programmed, written after the Switch instruction, followed by a colon, with instructions specified for each value in case the content of the variable matches that value. Finally, the Else clause accounts for any value greater than 10 or less than 1.

The flowchart and N-S diagrams are shown in Figures 33 and 34, respectively. The diagrams show the different execution paths generated from the multiple decision structure.

Figure 33. *Flowchart for Roman Numerals*

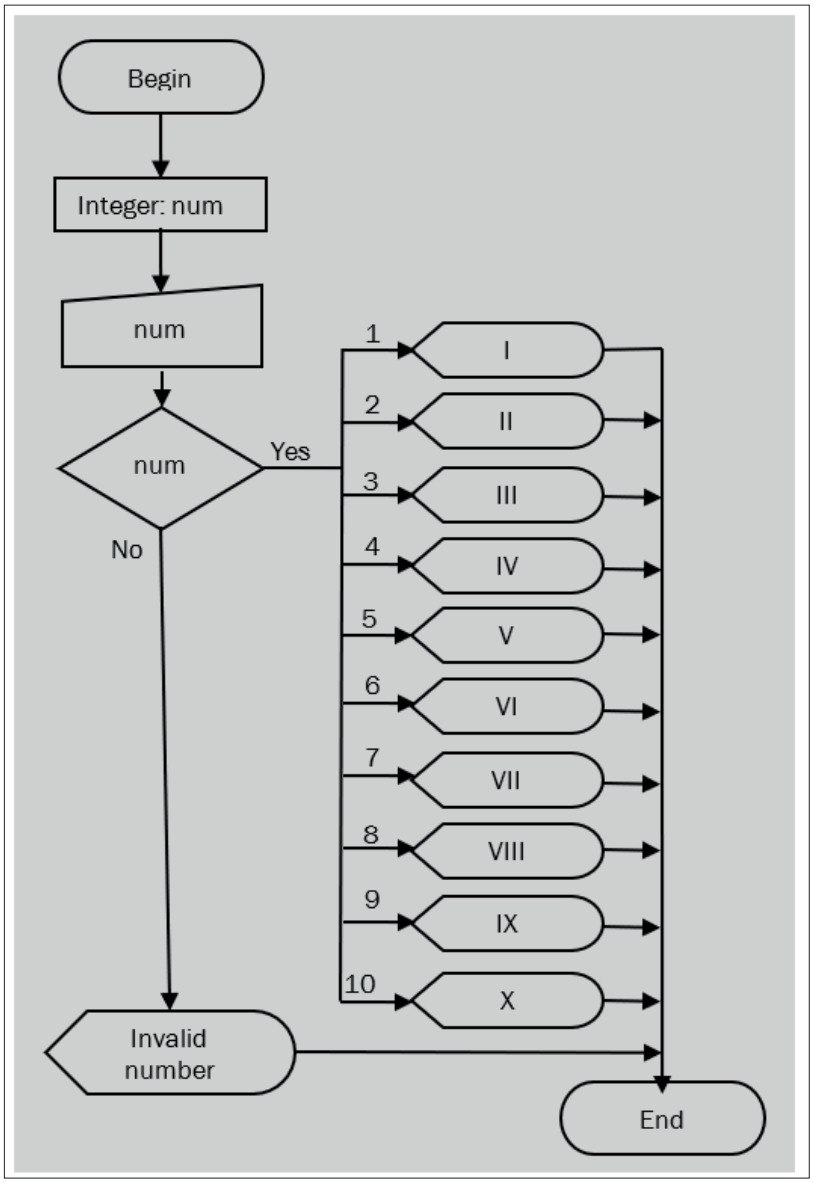


Figure 34. *N-S Diagram for Roman Numerals*

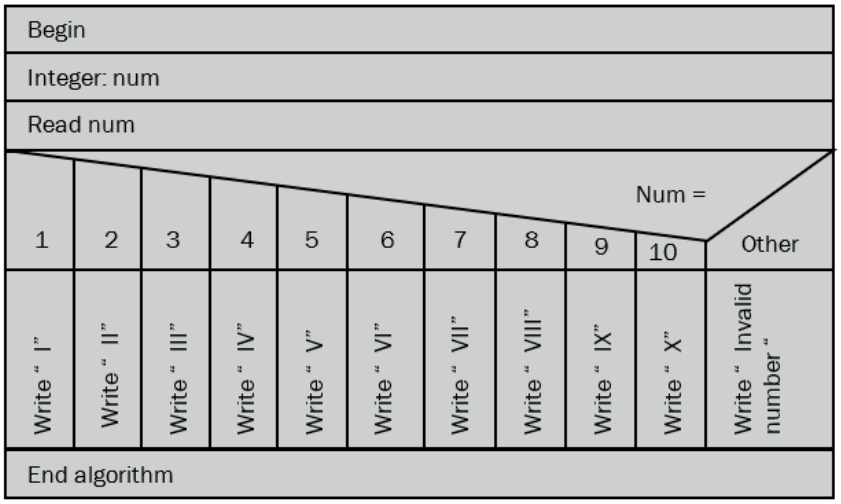


Table 30 shows the results of three executions with different numbers.

Table 30. *Verification of the Roman Numeral Algorithm*

Execution	num	Output
1	3	III
2	9	IX
3	12	Invalid number

Example 12. Day of the Week Name

This algorithm reads a number between one and seven corresponding to the day of the week and shows the name of the day, matching one (1) with Monday, two with Tuesday, and so on, using the ***Switch*** statement. If the entered number is less than one or greater than seven, it displays an error message. The pseudocode is shown in Table 31, the flowchart in Figure 35 and the N-S diagram in Figure 36.

In this exercise, the name of the day is not written in each case of the decision; instead, a variable is declared and assigned the corresponding name (lines 5 to 11). At the end, after the decision structure has closed, the name of the day is displayed (line 15).

Table 31. *Pseudocode for the Day of the Week Algorithm*

```
1   Begin
2       Integer: day
        String: dayName
3       Read day
4       Switch day do
5           1: dayName = “Monday”
6           2: dayName = “Tuesday”
7           3: dayName = “Wednesday”
8           4: dayName = “Thursday”
9           5: dayName = “Friday”
10          6: dayName = “Saturday”
11          7: dayName = “Sunday
12      Else
13          dayName = “Invalid number”
14      End switch
15      Write dayName
16  End algorithm
```

Figure 35. *Flowchart for the Day of the Week Algorithm*

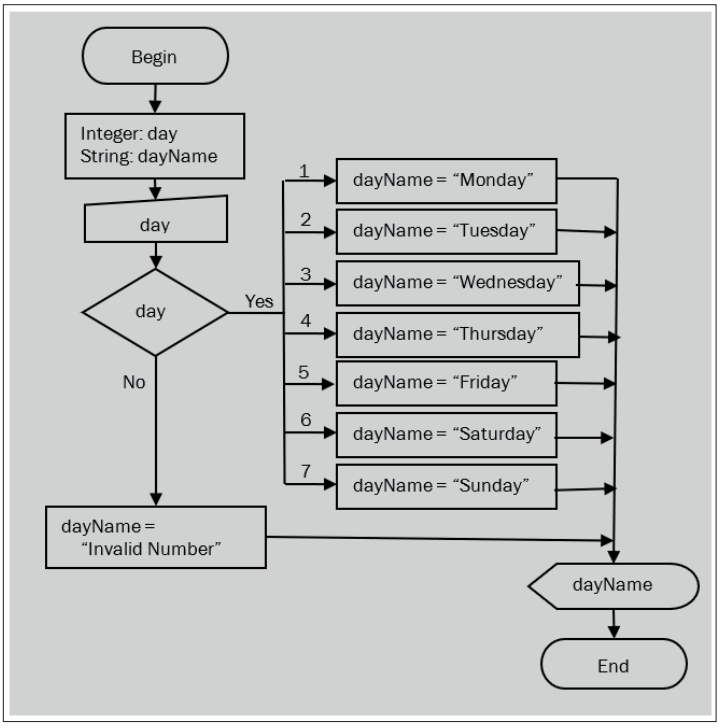


Figure 36. *N-S Diagram for the Day of the Week Algorithm*

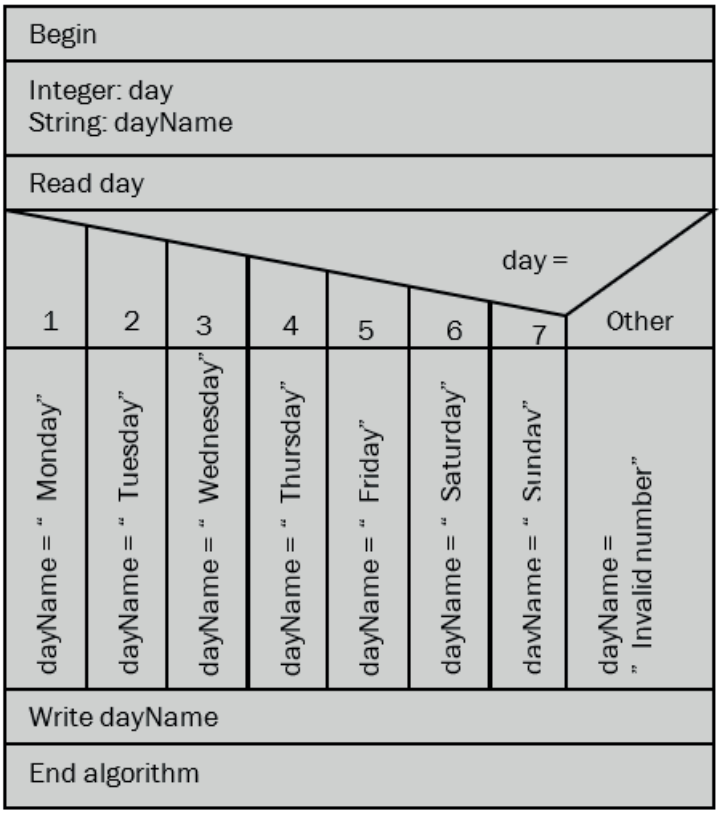


Table 32 shows three cases used to verify the functioning of the algorithm.

Table 32. *Verification of the Day of the Week Algorithm*

Execution	Day	dayName	Output
1	6	Saturday	Saturday
2	4	Thursday	Thursday
3	9	Invalid number	Invalid number

4.2.5 Nested Decisions

Nested decisions are those written within one another; meaning after making one decision, another must be made. In pseudocode, nesting has the form of the structure shown in Table 33.

If condition-1 is true, condition-2 is evaluated, and if this is also true, instructions-1 are executed. Thus, instructions-1 are executed only if condition-1 and condition-2 are true; instructions-2 are executed if condition-1 is true and condition-2 is false.

Table 33. *Nested Decisions*

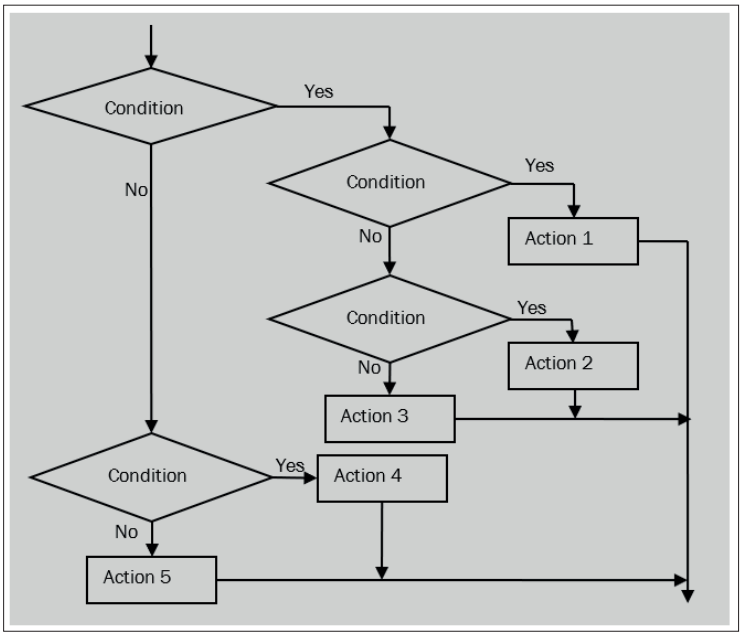
```
If <condition 1> then
        If <condition 2> then
           Instructions 1
         Else
                If <condition 3> then
                       Instructions 2
                 Else
                        Instructions 3
                End if
        End if
Else
        If <condition 4> then
             Instructions 4
        Else
              Instructions 5
        End if
End if
```

When the second decision is written for the case where the first is true, it is said to be nested by true, as occurs with condition-2. If the second decision is evaluated when the first decision is not fulfilled, it is said to be nested by false. Both true and false decisions can be nested as needed.

Nesting can be done with the ***If*** structure similarly with the ***Switch*** structure, and the two can be combined: an ***If*** within a ***Switch*** or otherwise, as many levels as the solution of the problem requires.

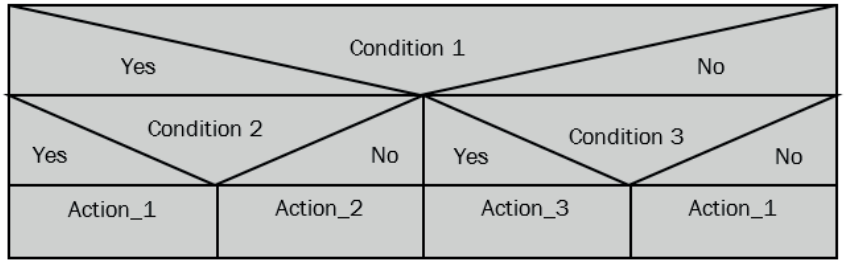
The representation in a flowchart is shown in Figure 37. In this type of diagram, it is easy to see the nesting of decisions and understand the dependence of the operations with respect to the decisions; one only needs to evaluate the condition and follow the corresponding arrow.

Figure 37. *Flowchart for Nested Decisions*



The N-S Diagram may seem complicated, but it is not; just read one box at a time, from top to bottom and left to right (see Figure 38) ***Condition 1*** sets two paths; following the left one leads to ***Condition 2***, which also provides two paths. If *Condition 1* is false, the right path is taken and leads to ***Condition 3***, which also provides two paths to follow, depending on the result of the condition.

Figure 38. *N-S Diagram for Nested Decisions*



Example 13. Comparing Two Numbers

Given two integers, they can be equal or different. If they are different, which one is greater? And which one is smaller?

In this case, two numbers are read from the keyboard and stored in two variables: n1 and n2.

The algorithm requires two decisions, the first being given by the condition:

n1 = n2 ?

If this condition is true, it displays a message. If not, another decision must be made based on the condition:

n1 > n2 ?

If this condition is true, it will show a result; if false, another result will be shown.

Table 34 presents the solution in pseudocode notation; the flowchart and N-S diagrams are shown in Figures 39 and 40.

Table 34. *Pseudocode for Number Comparison Algorithm*

```
12  Begin
3       Integer: n1, n2
4       Read n1, n2
5       If n1 = n2 then
6           Write "Numbers are equal"
7       Else
8           If n1 > n2 then
9               Write n1, "Greater"
10              Write n2, "Smaller"
11          Else
12              Write n2, "Greater"
13              Write n1, "Smaller"
14          End if
15      End if
16  End algorithm
```

Three tests are performed to verify the correctness of the algorithm. In the first execution, the number 3 is entered for both variables; thus, evaluating the first condition (line 4) yields true, and the message "Numbers are equal" is displayed (line 5). In the second execution, the numbers 5 and 2 are entered; this time, the first

condition is not met, and execution continues to line 7, where another condition is met, leading to lines 8 and 9 being executed. In the third execution, the numbers 6 and 8 are entered; when evaluating the first condition, it is not met, jumping to line 7 and evaluating the second condition, which is also false, leading to lines 11 and 12 being executed.

Figure 39. *Flowchart for Number Comparison Algorithm*

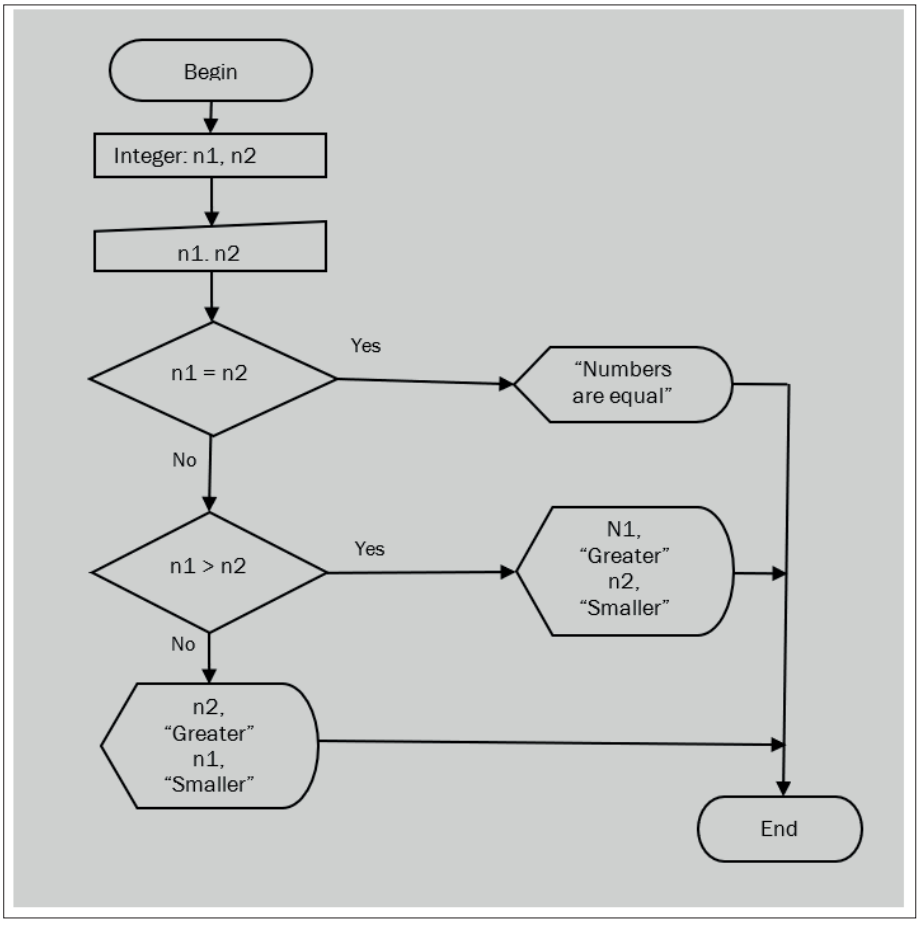
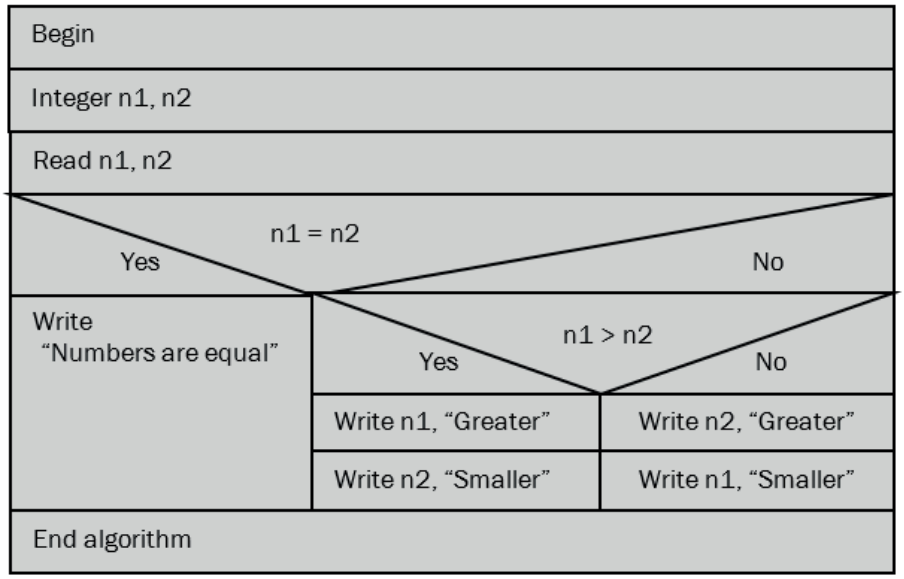


Figure 40. *N-S Diagram for Number Comparison Algorithm*



The results obtained from the algorithm tests are shown in Table 35.

Table 35. *Pseudocode for Number Comparison Algorithm*

Execution	n1	n2	Output
1	3	3	Numbers are equal
2	5	2	5 Greater 2 Smaller
3	6	8	8 Greater 6 Smaller

Example 14. Calculate Salary Increase

The company La Generosa S.A wants to increase salaries for its employees, having established the following conditions: those earning up to $800,000 will receive a 10% increase, those earning more than $800,000 and up to $1,200,000 will receive an 8% increase, and those earning more than that will receive a 5% increase. An algorithm is required to calculate the increase amount and the new salary for each employee.

To understand the problem, consider the following cases:

An employee earns $700,000; since this amount is below $800,000, they will receive a 10% increase, therefore:

Increase = 700,000 * 10 / 100 = 70,000
New salary = 700,000 + increase
New salary = 770,000

Another employee earns $1,000,000; this amount is above $800,000 but below 1,200,000; therefore, they will receive an 8% increase:

Increase = 1,000,000 * 8 / 100 = 80,000
New salary = 1,000,000 + increase
New salary = 1,080.000

A third employee earns $1,500,000; since this amount is above $1,200,000, the increase percentage is 5%:

Increase = 1,500,000 * 5 / 100 = 75,000
New salary = 1,500,000 + increase
New salary = 1,575,000

It follows that:

Inputs: salary (sal)
Outputs: increase amount (inc) and new Salary (newSal)
Calculations: increase = salary * percentage (per)
new salary = salary + increase

The solution design in pseudocode notation is presented in Table 36.

Table 36. *Pseudocode for Salary Increase Algorithm*

```
1    Begin
2        Real: sal, per, inc, newSal
3        Read sal
4        If sal <= 800000 then
5            per = 10
6        Else
7            If sal <= 1200000 then
```

```
8                   per = 8
9               Else
10                  per = 5
11              End if
12          End if
13          inc = Sal * per /100
14          newSal = Sal + inc
15          Write “Increase:”, inc
16          Write “New salary:”, newSal
17      End algorithm
```

The flowchart is presented in Figure 41 and the N-S diagram in Figure 42.

Figure 41. *Flowchart for Salary Increase Algorithm*

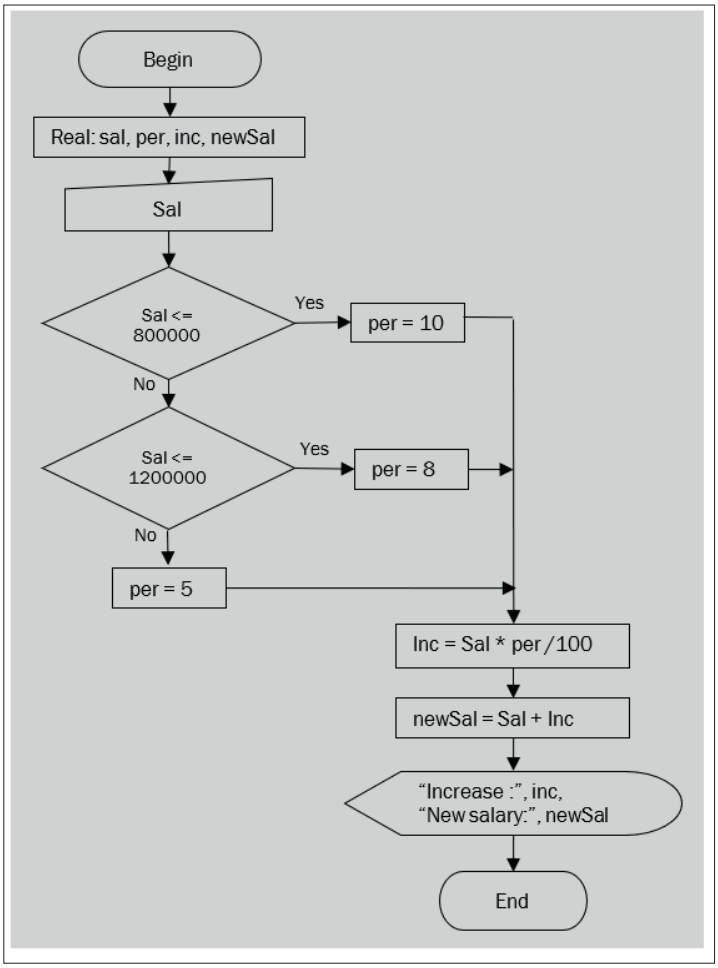
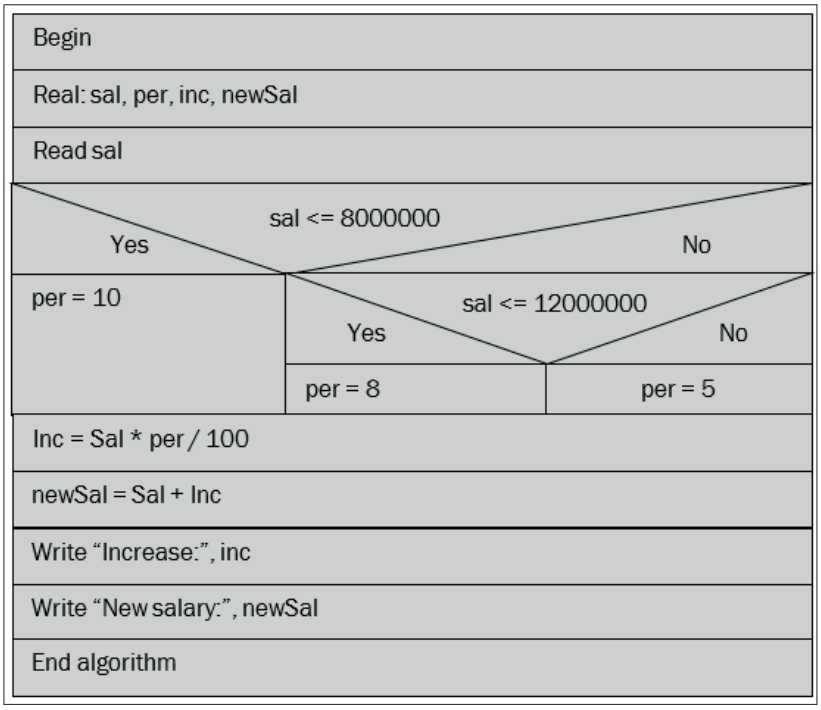


Figure 42. *N-S Diagram for Salary Increase Algorithm*



To verify that the algorithm works correctly, tests are conducted with the previously analyzed data. The results of this test are shown in Table 37.

Table 37. *Verification of the Salary Increase Algorithm*

Exec	Sal	per	inc	newSal	Output
1	700,000	10	70,000	770,000	Increase: 70,000 New salary: 770,000
2	1,000,000	8	80,000	1,080,000	Increase: 80,000 New salary: 1'080.000
3	1,500,000	5	75,000	1,575,000	Increase: 75,000 New salary: 1,575.000

4.2.6 More Examples of Decisions

Example 15. Recruitment

A company wants to hire a professional to fill a vacancy. The requirements to be considered eligible are: being a professional and being between 25 and 35 years old

inclusive, or having a postgraduate degree, in which case age is not taken into account. An algorithm is required to evaluate each candidate's data and inform whether they are suitable or not.

Following the strategy proposed in this document, particular cases are analyzed, data and operations are identified, and then a generic solution is designed.

Pedro applies for the position. He has a bachelor's degree in business administration and is 28 years old. Comparing his data with the problem's conditions, we find that his age falls within the valid range and he has a bachelor's degree; therefore, he is considered suitable.

Next, Lucía applies, who is a public accountant with a specialization in senior management and is 38 years old. Upon reviewing the conditions, we find that her age is outside the range, but since she has a specialization, she is categorized as suitable.

Finally, Carlos, who is 45 years old and has a bachelor's degree in economics, applies. Examining the first condition, we see that his age exceeds the established limit. This situation could be ignored if he had specialized training, but it is verified that his level of education is a bachelor's degree; therefore, he is declared unsuitable.

Now, we need to design an algorithm that automatically determines whether any person meets the requirements.

In this case, three input data points are required: name, age, and academic background. To minimize the possibility of error when entering the third data point, a menu is presented from which the user selects the corresponding option:

Academic Background

1. Technological
2. Professional
3. Specialist
4. Master's
5. Doctorate

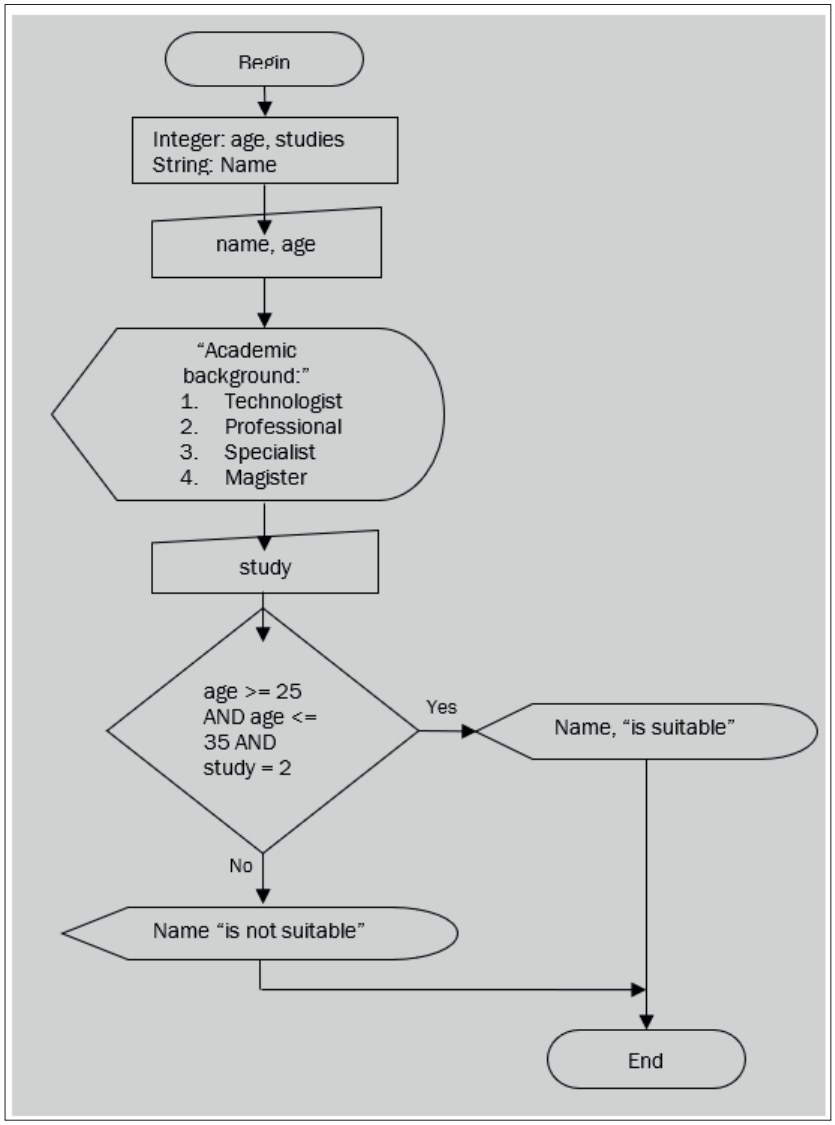
In this way, the condition is set on numerical data. The algorithm to solve this problem is presented as a flowchart in figure 43.

The results of the algorithm verification are shown in Table 38.

Table 38. *Verification of the Recruitment Algorithm*

Exec.	Name	Age	Studies	Output
1	Peter	28	2	Peter is suitable
2	Lucia	38	3	Lucia is suitable
3	Carlos	45	2	Carlos is not suitable

Figure 43. *Flowchart for Employee Selection*



Example 16. Transportation Allowance

An algorithm is required to decide whether an employee is entitled to transportation allowance. It is known that all employees earning a salary less than or equal to two minimum legal wages (SML in Spanish) are entitled to this benefit.

To identify the data of the problem and the solution, two examples are proposed:

Assume a legal minimum wage of $600,000.oo. José earns a monthly salary of $750,000.oo. The problem states that he is entitled to allowance if his salary is less than or equal to twice the legal minimum wage; thus:

750,000,oo <= 2 * 600,000,oo
750,000,oo <= 1,200,000,oo

This relational expression is true; therefore, José is entitled to transportation allowance.

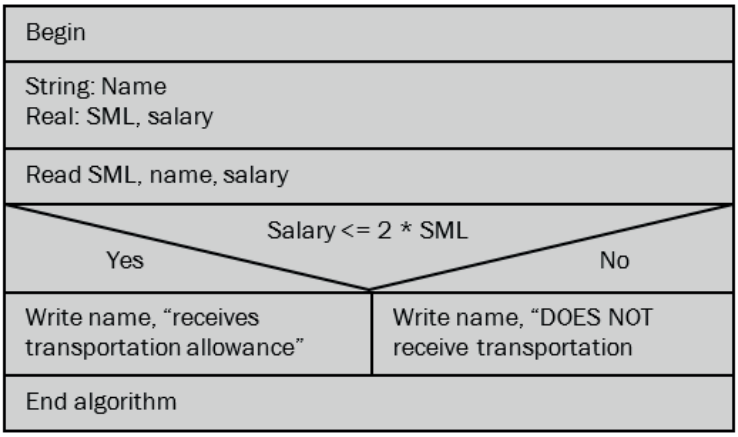
Luis earns $1,300,000.oo monthly. Is he entitled to allowance?

1,300,000,oo <= 2 * 600,000,oo
1,300,000,oo <= 1,200.000,oo

The condition is not met; consequently, Luis does not receive transportation allowance.

The design of the solution to this problem is proposed through the N-S diagram in Figure 44.

Figure 44. *N-S Diagram of the Transportation Allowance Algorithm*



The results of the verification of this algorithm, with the proposed examples, are shown in Table 39.

Table 39. *Verification of the Transportation Allowance Algorithm*

Exec.	SML	Name	Salary	Salary <= 2 * SML	Output
1	515,000	José	750,000	True	José receives transportation allowance
2	515,000	Luis	1,200,000	False	Luis does NOT receive transportation allowance

Example 17. Final Grade

At Buena Nota University, an algorithm is required to calculate the final grade and decide whether the student passes or fails the course. The final grade is obtained from two partial grades and a final exam, where the first partial is worth 30%, the second partial is worth 30%, and the final exam is worth 40%. The minimum passing grade is 3.0.

If the average of the two partial grades is less than 2.0, the student cannot take the final exam and fails the course due to low average; in this case, the final grade is the

average of the partial grades. If the average is equal to or greater than 2.0, the student can take the final exam.

If the final exam score is less than 2.0, the partial grades are disregarded, and the final grade is the score obtained in the final exam. If the score is equal to or greater than 2.0, the final grade is calculated using the above percentages for the partial grades and the final exam.

If the final grade is equal to or greater than 3.0, the student passes the course; if it is below 3.0, they fail; however, they can retake it, provided they score at least 2.0 on the final exam. In this case, the final grade will be the one obtained in the retake.

To better understand this situation, consider the following cases:

Raúl obtained the following grades:

Partial grade 1 = 2.2
Partial grade 2 = 1.6
Average = 1.9
Final grade = 1.9

Following the information of the problem, Raúl cannot take the final exam since the average of the two partial grades is less than 2.0; consequently, the final grade is the average of the partial grades, and he cannot take the final exam or retake. He fails the course.

Karol obtained the following grades:

Partial grade 1 = 3.4
Partial grade 2 = 2.0
Average = 2.7

She can take the final exam

Final exam = 1.5

Since her final exam score is less than 2.0, it becomes the final grade, and she cannot retake.

Carlos's grades were:

Partial grade 1 = 3.5
Partial grade 2 = 2.5
Average = 3.0

He can take the final exam

Final exam = 2.2

Final grade = 3.5 * 30% + 2.5 * 30% + 2.2 * 40% = 2.7

Carlos does not pass the course, but since he has a score greater than 2.0 on the final exam, he can retake

Retake = 3.5
Final grade = 3.5. Passes the course

Ana, on the other hand, obtained the following grades:

Partial grade 1 = 3.7
Partial grade 2 = 4.3

Average = 4.0

She takes the final exam

Final exam = 3.5

Final grade = 3.8

Therefore, Ana passes the course.

Now that the problem is well understood, an algorithm can be designed to solve any case. This algorithm is presented in Table 40.

Table 40. *Pseudocode for Final Grade Algorithm*

```
1   Begin
2       Real: p1, p2, fe, fg, aver, rg
3       Read p1, p2
4       aver = (P1 + P2)/2
5       If aver < 2.0 then
6           fg = aver
7           Write “fails the course due to low average”
8       Else
9           Read fe
10          If fe < 2.0 then
11              fg = fe
12              Write “Fails the course and cannot retake”
13          Else
14              fg = p1 * 0.3 + p2 * 0.3 + fe * 0.4
15              If fg >= 3.0 then
16                  Write “Passed the course”
17              Else
18                  Write “Failed the course but can retake”
19                  Read rg
20                  fg = rg
21                  If rg >= 3.0 then
22                      Write “Passed the retake”
23                  Else
24                      Write “Failed the retake”
25                  End if
26              End if
27          End if
28      End if
29      Write “Final grade:”, fg
30  End algorithm
```

In this algorithm, variables are declared to store the two partial grades, the average, the final exam, the final grade, and the retake grade. To start, the two partial grades are read, the average is calculated, and it is evaluated (line 5). If it is below 2.0, it is assigned to the final grade, a message is displayed, and the algorithm ends.

If the average is above 2.0, the final exam is read and evaluated (line 10). If it is less than 2.0, it is assigned as the final grade, a message is displayed, and it jumps to the end of the algorithm. If the final exam is greater than 2.0, the final grade is calculated, and it is determined whether the student passes or fails. If they fail, they can take the retake, which is the last chance to pass the course.

Table 41 shows three sets of data to test the algorithm and the results obtained.

Table 41. *Verification of the Final Grade Algorithm*

Exec	P1	P2	Aver	fe	fg	rg	Output
1	2.2	1.6	1.9	-	1.9		Failed the course due to low average Final grade: 1.9
2	3.4	2.0	2.7	1.5	1.5		Failed the course Final grade: 1.5
3	3.5	2.5	3.0	2.2	2.7	3.5	Failed the course but can retake Passed the retake Final grade: 3.5
4	3.7	4.3	4.0	3.5	3.8		Passed the course Final grade: 3.8

Example 18. Oldest Sibling

This algorithm reads the names and ages of three siblings and decides which sibling is the oldest.

Let's consider the following cases:

José, Luis, and María are siblings; José is 19 years old, Luis is 15, and María is 23. Who is the oldest? Clearly, it is Maria.

Carlos is 32, Rocío is 28, and Jesús is 25. In this case, Carlos is the oldest.

Martha is 8 years old, Ana is 10, and Camilo is 4. Among the three, Ana is the oldest.

Now, we need to design an algorithm for a computer program to make this decision. We need to declare three string variables and relate them to three numeric variables,

so the ages are compared, and the name is shown as a result. The flowchart for this algorithm is presented in Figure 45.

To verify that the algorithm is correct, we review it step by step, recording the values corresponding to the variables and the output on the screen. Table 42 shows the test data and the results obtained.

Figure 45. *Flowchart of the Oldest Sibling Algorithm*

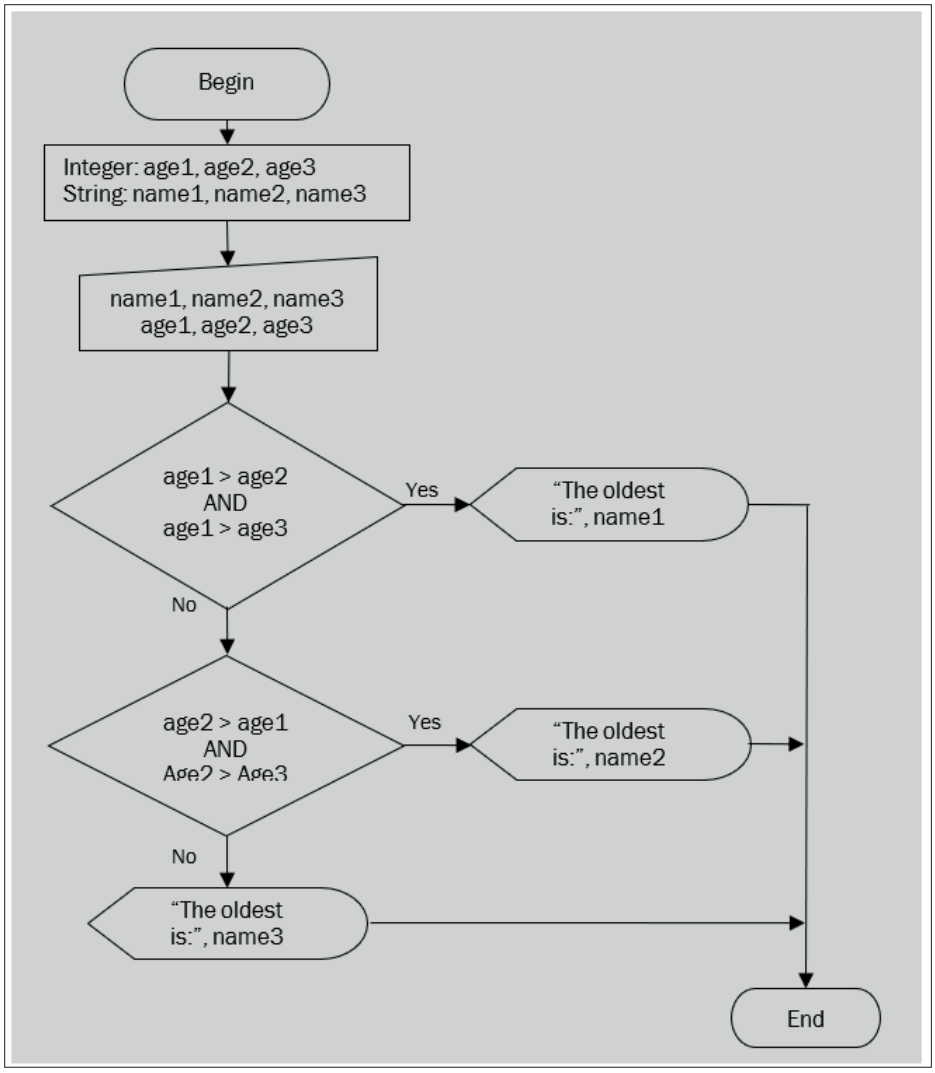


Table 42. *Verification of the Oldest Sibling Algorithm*

Exec.	name1	age1	name2	age2	name3	age3	Output
1	José	19	Luis	15	Maria	23	Maria
2	Carlos	32	Rocío	28	Jesús	25	Carlos
3	Martha	8	Ana	10	Camilo	4	Ana

Example 19. Wholesale Discount

Gran Distribuidor warehouse sells shirts in bulk and offers discounts based on the quantity purchased: for quantities of 1000 units or more, a 10% discount is applied; for quantities between 500 and 999, an 8% discount; between 200 and 499, a 5% discount; and for fewer than 200 units, there is no discount. Given the quantity purchased and the unit price, an algorithm is needed to calculate the discount provided to a customer and the amount to pay.

Before proceeding with the algorithm design, it is useful to understand the problem through a particular case for calculations.

If a shirt costs $80,000 and a customer orders 350 units, they are entitled to a 5% discount:

Units: 350
Unit price: 80,000
Total value = 350 * 80,000 = 28,000,000
Discount percentage: 5%
Discount value = 28,000,000 * 5 / 100 = 1,400,000
Amount payable = 28,000,000 – 1,400,000 = 26,600,000

In this case, the customer will receive a discount of $1,400,000 and will pay a total of $26,600,000.

Now, we can identify the input data, output data, and the processes to be performed:
Input: quantity, unit price
Output: discount value, amount payable
Processes:
Total = quantity * unit price
Discount value = total * discount percentage / 100
Amount payable = total - discount

The solution is presented by N-S diagram in Figure 46.

Figure 46. *N-S Diagram of the Wholesale Discount Algorithm*

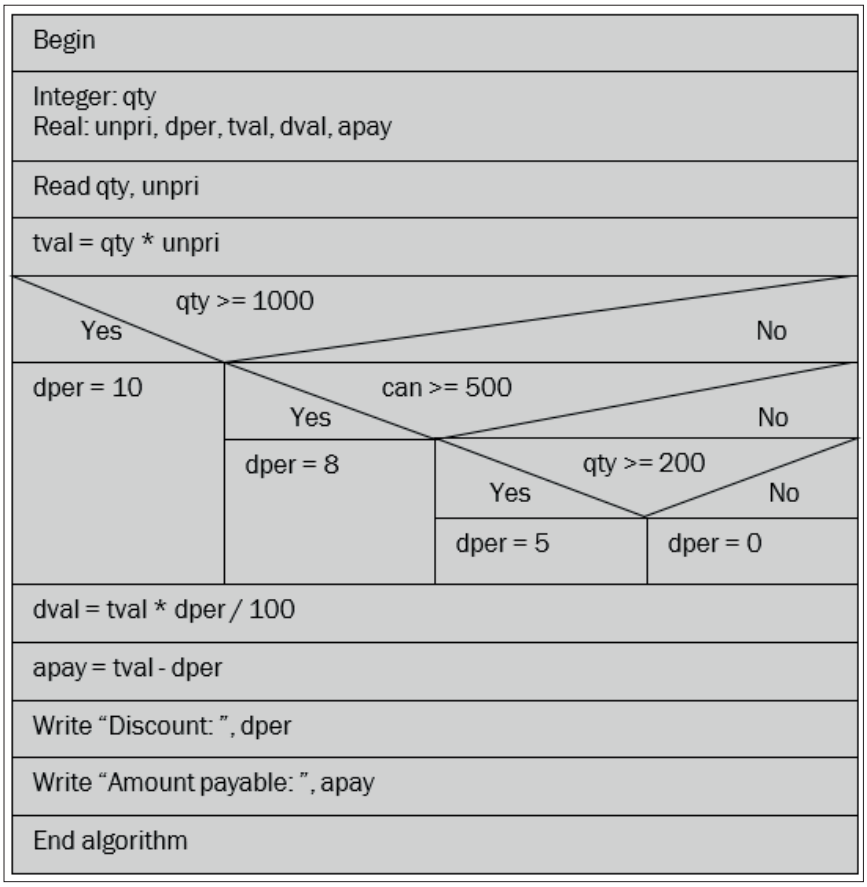


Table 43 shows the test data for this algorithm.

Table 43. *Verification of the Wholesale Discount Algorithm*

Exec.	qty	unpri	tval	dper	dval	apay	Output
1	350	80000	2800000	5	1400000	26600000	Discount 1400000 Pay 26600000
2	600	20000	12000000	8	960000	11040000	Discount 960000 Pay 11040000
3	1100	30000	3300000	10	3300000	29700000	Discount 3300000 Pay 29700000

Example 20. Term Deposit

Buena Paga bank offers different annual interest rates for term deposits depending on the duration. If the deposit is for a period of six months or less, the rate is 8% per year; between seven and 12 months, 10%; between 13 and 18 months, 12%; between 19 and 24 months, 15%; and for periods longer than 24 months, 18%. An algorithm is required to determine how much a customer will receive for a deposit, both in terms of interest and total amount.

Let's consider some specific cases:

Case 1: Pedro Pérez makes a deposit of one million pesos ($1,000,000.oo) for five months. According to the information provided in the problem statement, the bank will pay him an interest rate of 8% per year.

Since the interest rate is expressed annually and the deposit is in months, we first need to convert it to a monthly rate by dividing by 12.

```
Monthly interest rate = 8 / 12
Monthly interest rate = 0.667
```

Now, we have the following data:

```
Capital = 1000000.oo
Time = 5
Monthly interest rate = 0.667%
```

To find out how much he will receive in interest, we apply the percentage obtained to the capital value and multiply by the number of periods (in this case, five months):

```
Interest = 1000000.oo * (0.667 / 100) * 5
Interest = 33350
```

To obtain the future value of the investment, interest is added to the capital:

```
Future value = 1000000 + 33350
Future value = 1033350
```


Case 2: José López wants to deposit five million for a period of one and a half years (18 months). In this case, the bank offers him an interest of 12% per year.
The monthly interest rate converted is:

Monthly interest rate = 12 / 12 = 1

So, the data is:

Capital = 5000000
Monthly interest rate = 1 %
Time = 18

From which we obtain:
Interest = 5000000 * (1 / 100) * 18
Interest = 900000

Future value = capital + interest
Future value = 5000000 + 900000
Future value = 5900000

Based on the two examples, we conclude that:

Input data: capital, time
Output data: interest, future value
Processes:
  Determine the annual interest rate (is a decision)
  Monthly interest = annual interest rate / 12
  Interest = capital * monthly interest rate / 100 * time
  Future value = capital + interest

The algorithm to solve this exercise is presented in pseudocode in Table 44.

This algorithm uses nested if decision structures to determine the percentage corresponding to the annual interest rate, depending on time. Nested decisions are applied rather than a Switch structure since the latter can only evaluate the condition of equality between the content of the variable and a constant value, not for relationships of greater or lesser value.

Table 45 shows the results of verifying the algorithm with the data used in the two examples at the beginning of this exercise, plus a third case with a value of 8000000 over 36 months.

Table 44. *Pseudocode for Term Deposit Algorithm*

```
1   Begin
2       Real: capital, annualIntRate, monthIntRate, interest, futureValue
3       Integer: time
4       Read capital, time
5       If time < 1 then
6           Write “Invalid time”
7           annualIntRate = 0
8       Else
9           If time <= 6 then
10              annualIntRate = 8
11          Else
12              If time <= 12 then
13                  annualIntRate = 10
14              Else
15                  If time <= 18 then
16                      annualIntRate = 12
17                  Else
18                      If time <= 24 then
19                          annualIntRate = 15
20                      Else
21                          annualIntRate = 18
22                      End if
23                  End if
24              End if
25          End if
26      End if
27      monthIntRate = annualIntRate / 12
28      Interest = Capital * monthIntRate / 100 * Time
29      futureValue = capital + interest
30      Write “Interest = ”, interest
31      Write “Future value = ”, futureValue
32  End algorithm
```

Table 45. *Verification of the Term Deposit Algorithm*

Capital	Time	annualIntRate	monthIntRate	interest	Future value
1000000	5	8	0,667	33350	1033350
5000000	18	12	1	900000	5900000
8000000	36	18	1,5	4320000	12320000

Example 21. Publishing Company

A publishing company has three groups of employees: salespeople, designers, and administrative staff. It is required to calculate the new salary for employees, considering the following increases: administrative staff 5%, designers 10%, and salespeople 12%.

As a particular case, we will calculate the new salary for the secretary, who currently earns $600,000.oo. Since her position is classified as administrative, she will receive a 5% increase.

Position: Secretary
Job Type: administrative
Current Salary: 600,000.oo
Increase Percentage: 5%

Thus,

Increase Amount: 30,000.oo
New Salary: 630,000.oo

In this exercise, the current salary and job type (administrative, designer, salesperson) are taken as input data, since the increase percentage depends on the job type and is applied to the current salary. The algorithm to solve this problem is presented in flowchart notation in Figure 47.

In this algorithm, three real-type variables are declared for the current salary, new salary, and increase percentage, and an integer variable is used to capture the option corresponding to the employee's job type. A (multiple) decision is made based on the job type and the percentage to be applied is established, the calculation is made, and the new salary is shown. If the selected job type option is not between 1 and 3, an error message is displayed.

Three sets of data are proposed to verify if the algorithm is correct. The results of the executions are presented in Table 46.

Figure 47. *Flowchart for Publishing Company Algorithm*

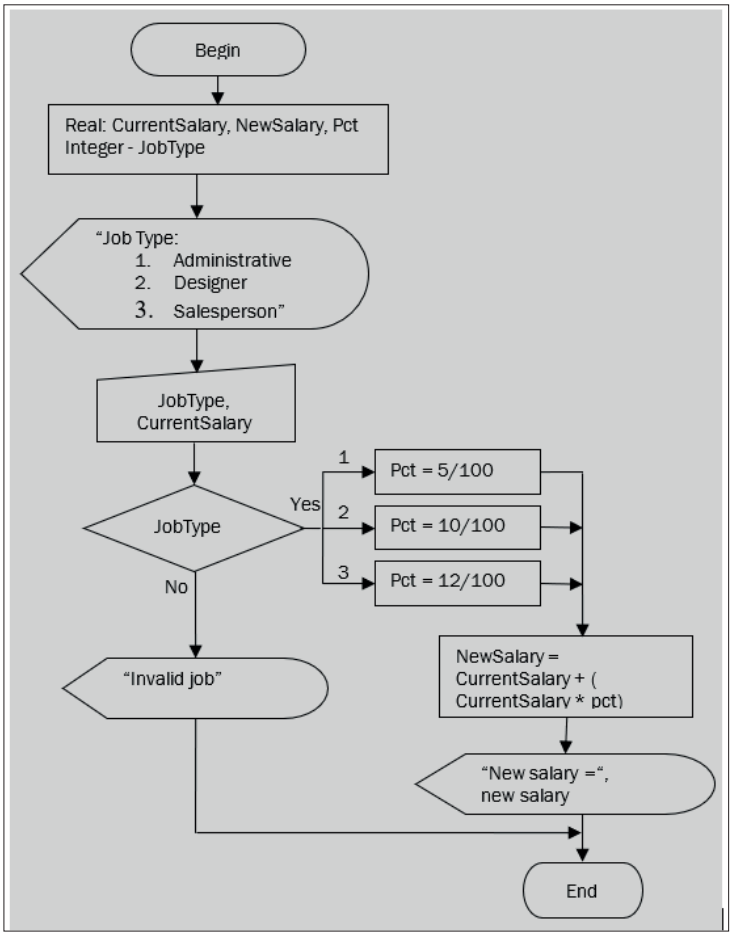


Table 46. *Verification of the Publishing Company Algorithm*

Exec	JobType	CurrentSalary	Pct	NewSalary	Output
1	1	600,000	0.05	630,000	New salary = 630,000
2	2	800,000	0.1	880,000	New salary = 880,000
3	3	700,000	0.12	784,000	New salary = 784,000
4	4	500,000			Invalid job

Example 22. Motorcycle Discount

The motorcycle distributor Rueda Floja offers a promotion as follows: Honda motorcycles have a 5% discount, Yamaha motorcycles have an 8% discount, Suzuki motorcycles have a 10% discount, and motorcycles from other brands have a 2% discount. It is required to calculate the amount to be paid for a motorcycle.

Let's consider some examples of this problem.

Pedro wants to buy a Honda motorcycle priced at six million pesos. How much does he really have to pay for that motorcycle?

In the problem description, it is said that this type of motorcycle has a 5% discount, so we need to calculate the discount amount for the motorcycle Pedro wants. We have the following data:

Motorcycle price = 6,000,000,oo
Discount percentage = 5/100 (5%)
Discount amount = 6,000,000,oo * 5 / 100 = 300,000,oo
Net amount = 6,000,000,oo – 300,000,oo = 5,700,000,oo

Since the motorcycle Pedro wants has a 5% discount, the net amount to be paid is $ 5,700,000,oo.

Juan wants to buy a Yamaha motorcycle with a list price of $8,500,000,oo. Since this brand has an 8% discount, we have:

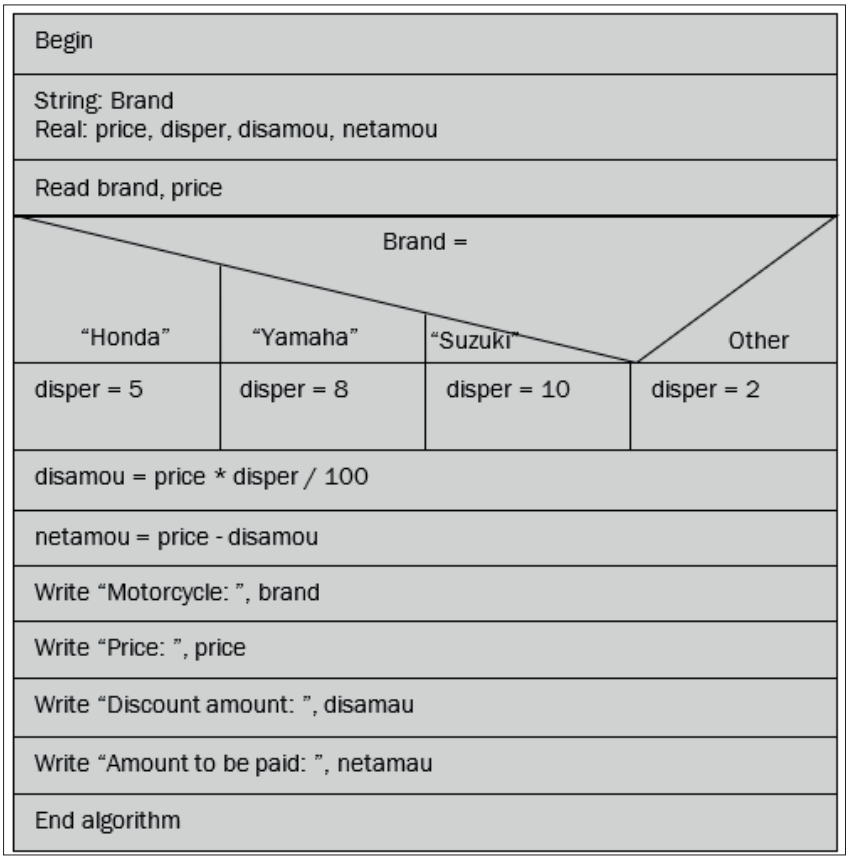
Motorcycle price = 8,500,000,oo
Discount percentage = 8/100 (8%)
Discount amount = 8,500,000,oo * 8 / 100 = 680,000,oo
Net amount = 8,500,000,oo – 680,000,oo = 7,820,000,oo

Therefore, Juan must pay the sum of $7,820,000,oo for the Yamaha motorcycle. Based on the previous examples, we can generalize the procedure as follows:

Discount percentage = depends on the brand (5%, 8%, 10%, or 2%)
Discount amount = motorcycle price * discount percentage
Net amount = motorcycle price – discount amount

Consequently, the data to be entered corresponds to: the list price and the brand of the motorcycle. The algorithm that solves this problem is presented in N-S notation in Figure 48.

Figure 48. *Motorcycle Discount*



To verify that the algorithm is correct, the desk checking is performed considering each of the decision alternatives, as shown in Table 47.

Example 23. Sales Commission

The V&V company pays its salespeople a base salary plus a commission on sales made during the month, provided that these exceed $1,000,000. The commission percentages are:

Sales greater than $1,000,000 and up to $2,000,000 = 3%
Sales greater than $2,000,000 and up to $5,000,000 = 5%
Sales greater than $5,000,000 = 8%
The goal is to determine how much commission the salesperson is entitled to and what their total earnings will be for the month.

Following the approach proposed for this book, the first thing is to consider some examples to understand the problem and how to solve it. In this order of ideas, four examples are proposed:

Table 47. *Verification of the Motorcycle Discount Algorithm*

Exe	Brand	price	disper	disamou	netamou	Output
1	Honda	6,000,000	5	300,000	5,700,000	Motorcycle: Honda Price: 6,000,000 Discount amount: 300,000 Amount payable: 5,700,000
2	Yamaha	8,500,000	8	680,000	7,820,000	Motorcycle: Yamaha Price: 8,500,000 Discount amount: 680,000 Amount payable: 7,820,000
3	Suzuki	3,800,000	10	380,000	3,420,000	Motorcycle: Suzuki Price: 3,800,000 Discount amount: 380,000 Amount payable: 3,420,000
4	Kawasaki	5,000,000	2	100,000	4,900,000	Motorcycle: Kawasaki Price: 5,000,000 Discount amount: 100,000 Amount payable: 4,900,000

Jaime was hired with a base salary of $600,000,oo and during the last month, he sold $1,200,000,oo. Since he sold more than one million, he is entitled to a commission. The next step is to determine which percentage applies. The exercise states

that if sales are between one and two million, the salesperson receives a commission of 3% on the sales value. This is calculated as follows:

Salesperson: Jaime
Base salary: 600,000,oo
Sales value: 1,200,000,oo
Commission percentage: 3%
Commission value = 1,200,000 * 3 / 100 = 36,000
Monthly salary = 600,000 + 36,000 = 636,000

Jaime will receive $36,000 in commission, resulting in a total salary of $636,000,oo

Susana has a base salary of $700,000,oo her sales for the last month amounted to $3,500,000,oo. In this case, according to the problem's requirements, she is entitled to a 5% commission. The results for Susana are:

Salesperson: Susana
Base salary: 700,000,oo
Sales value: 3,500,000,oo
Commission percentage: 5%
Commission value = 3,500,000 * 5 / 100 = 175,000
Monthly salary = 700,000 + 175,000 = 875,000

Susana earns a commission of $175,000, resulting in a total salary of $875,000.oo. Carlos has a base salary of $600,000,oo and his sales last month totaled $950,000,oo. Since he did not reach one million in sales, he will not receive a commission.

Salesperson: Carlos
Base salary: 600,000,oo
Sales value: 950,000,oo
Commission percentage: None
Commission value = None
Monthly salary = 600,000

Carlos only receives his base salary.

Rocío is known as the company's top seller, with a base salary of $800,000,oo. In the last month, she managed to sell seven million pesos.

Salesperson: Rocío
Base salary: 800,000,oo
Sales value: 7,000,000,oo
Commission percentage: 8%
Commission value = 7,000.000 * 8 / 100 = 560,000
Monthly salary = 800,000 + 560,000 = 1,360,000

Rocío receives a commission of $560,000,oo and a total salary of $1,360,000,oo
From the previous examples, the input data and the calculations to be performed are identified. The output data specified by the problem statement includes the salesperson, commission, and total salary. The algorithmic solution is presented in Table 48 in pseudocode notation.

Input Data:
Salesperson's name
Base salary
Sales value
Calculations to Perform:
Commission value = sales value * commission percentage (3, 5, 8) / 100
Monthly salary = base salary + commission value

To verify that the algorithm works correctly, the pseudocode is executed step by step using the data of the previously explained examples, for which the calculations and results are already known. The results are presented in Table 49.

Table 48. *Sales Commission Algorithm*

```
1   Begin
2     String: name
3     Real: basesal, sale, comper, comval, monthsal
4     Read name, basesal, sale
5     If sale > 1000000 then
6         If sale <= 2000000 then
7             comper = 3
8         Else
9             If sale <= 5000000 then
10                comper = 5
11            Else
12                comper = 8
```

```
13                End if
14            End if
15        Else
16            comper = 0
17        End if
18        comval = sale * comper / 100
19        monthsal = basesal + comval
20        Write "Salesperson:", name
21        Write "Commission:", comval
22        Write "Total salary:", monthsal
23    End algorithm
```

Table 49. *Verification of the Sales Commission Algorithm*

Name	basesal	Sale	comper	comval	monthsal	Output
Jaime	600000	1200000	3	36000	636000	Salesperson: Jaime Commission: 36000 Total salary: 636000
Susana	700000	3500000	5	175000	875000	Salesperson: Susana Commission: 175000 Total salary: 875000
Carlos	600000	950,000	0	0	600000	Salesperson: Carlos Commission: 0 Total salary: 600000
Rocío	800000	7000000	8	560000	1360000	Salesperson: Rocío Commission: 560000 Total salary: 1360000

Example 24. Electric Service Billing

An algorithm is required to bill the electric power service. The monthly consumption is determined by the difference in readings. The value per kilowatt (kW[6]) is the same for all users, but a discount is applied based on socioeconomic strata as follows:

Stratum 1: 10%

6 kW is the kilowatt symbol. A kilowatt is a unit of measurement of electricity equivalent to 1000 watts. It is also equivalent to the energy produced or consumed by a power of one kW for one hour.

Stratum 2: 6%
Stratum 3: 5%

Additionally, a 2% discount is applied to all strata if consumption exceeds 200 kW. The program should display the consumption and the amount to be paid for this service. (This exercise is purely educational and has no relation to actual billing practices).

In a real company, records of readings are maintained, so only one reading is necessary and it is compared with the last stored value. However, for this exercise, both readings must be taken to calculate the difference. To better understand this approach, let's consider the following examples:

José is a user of the energy service residing in stratum 1. Last month, his meter registered 12345, and this month it registered 12480. Since he is in stratum 1, he will receive a 10% discount. If the value of kW is $500, the consumption and amount to be paid are calculated as follows:

User: José
Reading 1: 12345
Reading 2: 12480
Consumption: 12480 – 12345 = 135
Value per kW: 500
Value of consumption: 67500
Stratum: 1
Discount percentage: 10%
Discount value = 67500 * 10 / 100 = 6750
Amount payable = 67500 – 6750 = 60750

From this example, we can identify the necessary data and processes to solve the exercise. The input data includes:

User,
Socioeconomic stratum,
Reading 1,
Reading 2 and
Value per kW.

(Universal Encyclopedia Circle)

The discount percentage to be applied is based on the user's socioeconomic stratum. The values to be calculated are:

Consumption = reading 2 – reading 1
Value of consumption = consumption * value per kW
Discount value = value of consumption * discount percentage / 100
Amount payable = value of consumption – value discounted

The next step is to organize the actions algorithmically. The flowchart of the solution to this exercise is presented in Figure 49.
To verify that the solution is correct, the algorithm is executed step by step, using the data from the analysis, and the results generated by the algorithm are compared with the manually obtained results.

Example 25. Calculator

This algorithm reads two numbers and an arithmetic operator, applies the corresponding operation, and displays the operation name and result.

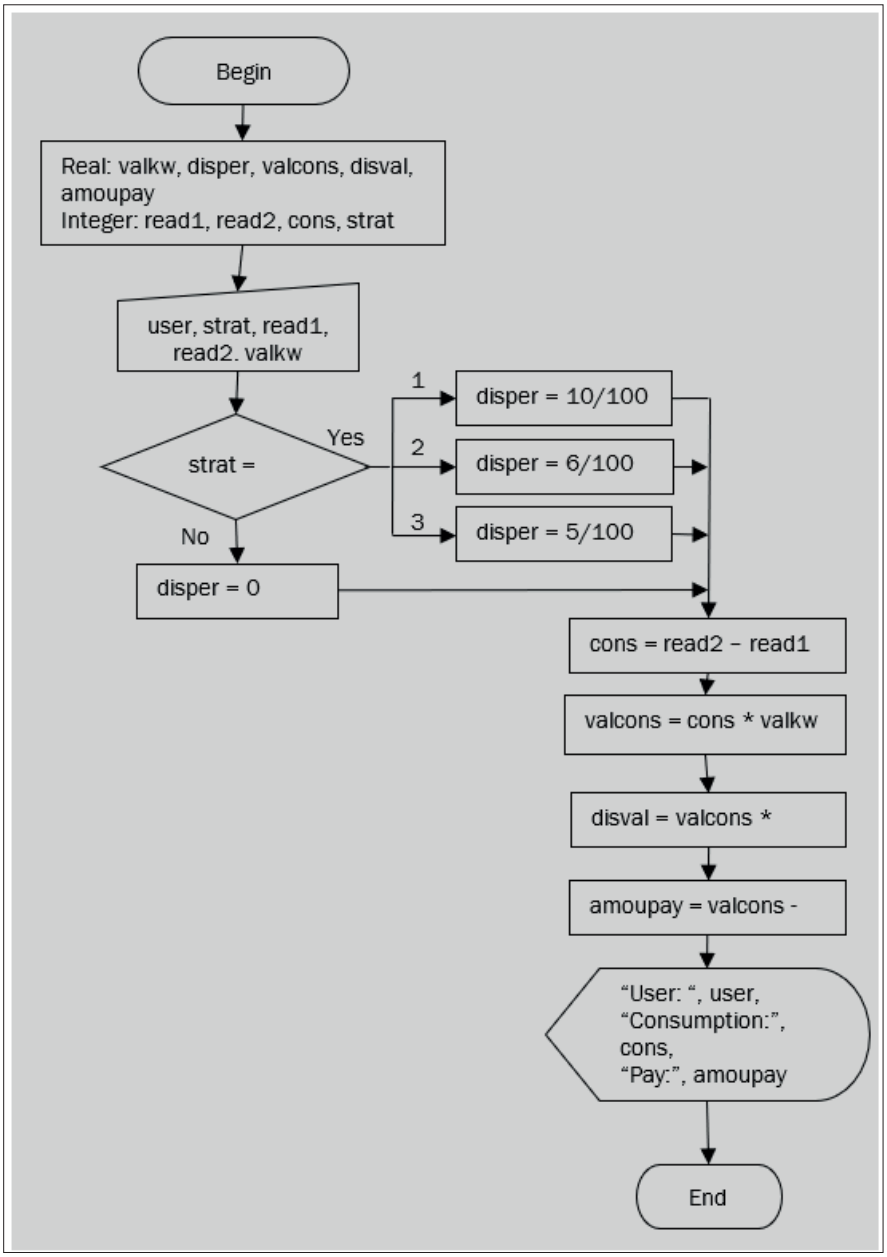
In this case, it is necessary to identify the operator entered by the user and perform the corresponding operation. The arithmetic operators to consider are:

+	addition
-	subtraction
*	multiplication
/	division

For example:

First number: 3
Second number: 4
Operator: +

Figure 49. *Electric Service Billing*



The algorithm should perform the addition and display the result as follows:

Addition: 3 + 4 = 7
Another example:
First number: 6
Second number: 2
Operator: -

The result will be:

Subtraction: 6 – 2 = 4

The algorithm to solve this exercise requires a multiple decision applied to the operator, where each decision alternative performs the corresponding operation and displays the result. The N-S diagram of this algorithm is presented in Figure 50.

The variables declared are: a, b, and r for first number, second number, and result, and ope for the operator. During the verification of the algorithm, all operators are tested, and the results are shown in Table 50.

Figure 50. *N-S Diagram of the Calculator Algorithm*

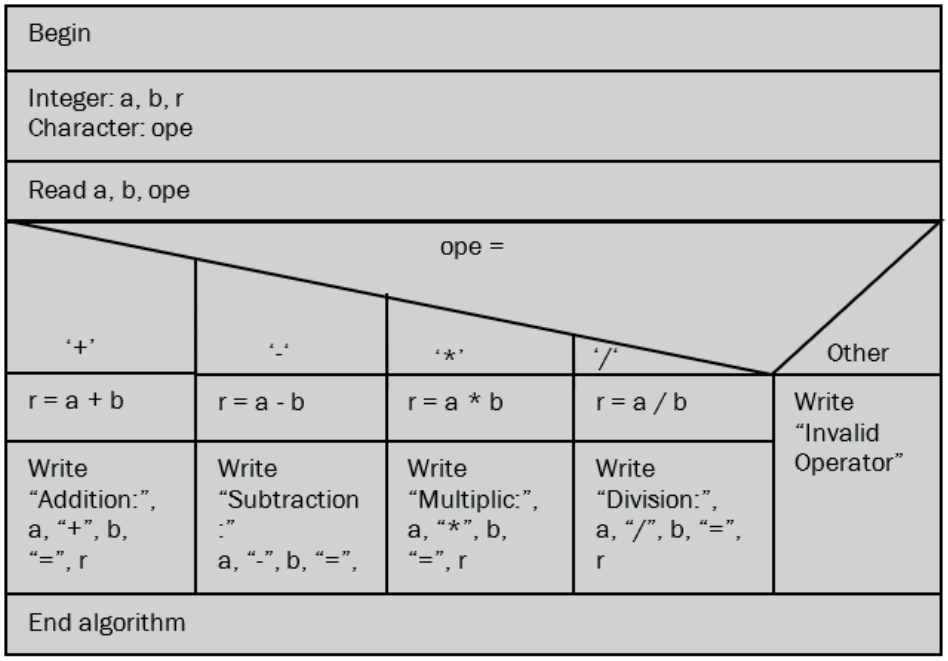


Table 50. *Verification of the Calculator Algorithm*

Exec.	a	b	ope	r	Output
1	3	4	+	7	Addition: 3 + 4 = 7
2	6	2	-	4	Subtraction: 6 – 2 = 4
3	2	5	*	10	Multiplic: 2*5=10
4	27	9	/	3	Division: 27/9 = 3
5	8	7	X		Invalid Operator

4.2.7 Proposed Exercises

Design algorithms to solve the following problems and represent them using pseudocode, flowcharts, and N-S diagrams.

To be selected for the basketball team, in addition to being a good player, a minimum height of 1.70 m is required. Given the height of an applicant, decide if they are eligible.

1. Given two integers, determine if one of them is a multiple of the other.

2. Given two rational numbers a/b and d/e, determine if they are equivalent; if not, identify which one is greater and which is smaller.

3. A company pays a bonus to its employees based on their length of service and marital status: for single employees: if they have up to five years, they receive 2% of their salary; between 6 and 10 years, 5%; and more than 10 years, 10%. For married employees: if they have up to 5 years, they receive 5% of their salary; between 6 and 10 years, 10%; and more than 10 years, 15%.

4. A store offers an 8% discount to customers whose purchase exceeds $1,000,000 and a 5% discount if the purchase is greater than $500,000 but less than or equal to $1,000,000. How much will a person pay for their purchase?

5. A year is a leap year if it is divisible by four and not by 100, or if it is divisible by 400. Determine if a given year n is a leap year.

6. An algorithm is required that reads three numbers and sorts them in ascending order.

7. Given a cylindrical container with radius r and height h, and a box with width a, length b, and height c, determine which one has a greater storage capacity.

8. A supermarket offers a 10% discount for the purchase of 10 or more units of the same item. How much will a customer have to pay for their purchase?

9. An insurance company opened a new agency and established a program to attract clients. If the amount for the insurance contract is less than $5,000,000, a 3% fee is charged; if the amount is between $5,000,000 and $20,000,000, a 2% fee is charged; and if the amount is $20,000,000 or more, a 1.5% fee is charged. What will be the value of the policy?

10. ABC University has a scholarship program for students with good academic performance. If the average of the four subjects taken each semester is greater than or equal to 4.8, the student does not have to pay tuition for the next semester; if the average is greater than or equal to 4.5 and less than 4.8, the student will receive a 50% discount; for averages greater than or equal to 4.0 and less than 4.5, the fee remains the same; and for averages below 4.0, it increases by 10% compared to the previous semester. Given the final grades, determine the tuition fee for the next semester.

11. A computer distributor offers discounts based on the quantity purchased. For purchases of less than five units, there is no discount; for purchases of five to nine units, a 5% discount is applied; and for purchases of 10 or more units, a 10% discount is applied. What will be the value of the discount? How much will the customer pay?

12. A restaurant offers home delivery under the following conditions: if the order exceeds $20,000, there is no additional delivery charge; if it is greater than $10,000 and up to $20,000, a $2,000 charge will be applied; and if it is less than $10,000, a $4,000 charge will be applied. What amount should the customer pay?

13. A distributor has three salespeople who receive a base salary plus a 5% commission on sales, provided that the sales amount is greater than or equal to $1,000,000. Additionally, the salesperson with the highest sales for the month will receive an additional 2% commission on their sales, regardless of the amount. What will be the salary of each salesperson?

14. In a job interview, the following criteria are taken into account: formal education, age, and marital status. The scores are as follows: for ages 18-24 years, 10 points; for ages 25-30, 20 points; for ages 31-40, 15 points; and for those over 40, 8 points. For high school education, 5 points; for technical education, 8 points; for college degrees, 10 points; and for postgraduate degrees, 15 points. Marital status: single 20 points, married 15 points, common-law 12 points, separated 18 points. Calculate the total score for an interviewee.

15. Given a year and a month, determine how many days that month has, considering that February changes depending on whether the year is a leap year.

16. Given an integer between 1 and 9999, express it in words

17. A jewelry store sets the price of its products based on weight and material. An algorithm is required that reads a reference value x and calculates the price per gram of material used and the total value of the product. The cost per processed gram is 3x for silver, 5x for platinum, and 8x for gold.

18. A company provides a special maternity/paternity bonus to its employees based on the number of children: for employees with no children, there is no bonus; for one child, the bonus is 5% of the salary; for two children, it is 8%; for three children, it is 10%; for four children, it is 12%; and for more than four, it is 15%.

19. An appliance store has the following sales policy: for cash sales, a 10% discount is applied, while for credit sales, a percentage is added to the product value depending on the number of monthly installments, which is divided by the number of installments. For 2 months, there is no financing interest; for 3 months, the value increases by 5%; for 4 months, it increases by 10%; for 5 months, it increases by 15%; and for 6 months, it increases by 20%. Determine the product value, the amount payable by the customer, the value of the discount or increase, and the value of each installment, if applicable.

20. An algorithm is needed to calculate the area of major geometric figures: triangle, rectangle, circle, rhombus, and trapezoid. The areas are calculated using the following formulas: area of a triangle = b*h/2, area of a rectangle = b*h, area of a circle = Pi * radius^2, area of a rhombus = D*d/2, area of a trapezoid = ½h(a+b)

21. An algorithm is required to bill phone calls. The cost per minute depends on the type of call: local = $50, regional = $100, national long-distance = $500, international long-distance = $700, and mobile = $200. Regional and national long-distance calls have a 5% discount if their duration is 5 minutes or more.

22. A car rental company requires a program to calculate the cost of each rental. The cost is determined based on the type of vehicle and the number of days the user uses it. The daily rates per category are: car = $20000, camper = $30000, minibus = $50000, and truck = $40000.

23. José Martí University grants a scholarship covering 100% of tuition fees to the student with the highest average in each group and 50% to the second-highest average. Knowing the names and averages of the four best students in a group, determine who will receive the scholarship.

24. A sports sponsorship company supports athletes who meet the following conditions in their training times over the last month: the longest time does not exceed 40 minutes, the shortest time is less than 30 minutes, and the average time is less than 35 minutes. An algorithm is required to decide if an athlete meets the conditions for sponsorship.

25. An engineering student needs three books: Calculus, Physics, and Programming, with prices v1, v2, and v3 respectively. The student has a budget of x of financial support for this purpose. An algorithm is required to help the student decide whether they can buy all three books, two books, or one book. The student wishes to prioritize the more expensive books since any remaining money must be returned, and the less expensive books will be easier to acquire with their own resources.

4.3 ITERATION STRUCTURES

These structures are also known as loops or repetition structures and are used to program one or more actions to be executed repeatedly. They are very important in programming since many activities need to be performed more than once.

Consider the case of an application for generating an invoice. A customer can buy one product, but they can also buy two, three, or more. The same operations performed to bill the first product must be executed for the second, third, and so on. Iterative structures allow instructions to be written once and reused as many times as necessary.

All repetition structures have at least two parts: definition and body. The definition includes a condition that indicates how many times the loop body should repeat; if the number of repetitions is not known in advance, the condition will specify the circumstances under which the loop body repeats, such as using a switch variable. The loop body consists of the instruction or set of instructions that include the repetition structure and will be executed multiple times.

4.3.1 Controlling Iterations

To establish the iteration condition and have control over it, it is almost always necessary to use a variable, whether it is a counter or a switch. This variable is referred to as the loop control variable (Joyanes and Zahonero, 2002).

A loop control variable has three stages:

Initialization. For a variable to be used in the definition of an iterative structure, its initial value must be known. Since these are always counters or switches, their value will not be read during program execution, and the programmer must assign it before including the variable in a condition. If the variable is not initialized, it may happen that the loop does not execute at all or runs infinitely.

Evaluation. The evaluation of the variable can occur either before or after each iteration, depending on the structure used, to decide whether to execute the loop body once more or exit and continue with the other instructions.

Update. If the variable is initialized to a value that allows the loop to execute, it is necessary to update it during each iteration so that at some point in the execution, the loop can exit. If the loop is controlled by a variable and this variable is not updated, an infinite loop will occur.

When specifying iterations controlled by a variable, it is important to ask the following questions: What is the initial value of the variable? What value should the variable have for the loop body to repeat? How does the variable change? The answers to these questions will require defining the three stages referred to and avoiding logical errors.

4.3.2 WHILE Structure

This loop consists of a set of instructions that repeat as long as a condition is met. As in decision structures, the condition is evaluated and returns a logical value that can be true or false. In the case of the while loop, the instructions contained in the repetition structure will only execute if evaluating the condition yields a true value; that is, if the condition is met; otherwise, the instruction following End while will execute.

Unlike other loops, the while structure begins by evaluating the conditional expression; if the result is true, the loop body instructions execute; upon reaching the End while line, the condition is re-evaluated. If it holds true, the instructions execute again, and this process continues until the condition is no longer met, at which point control of the program passes to the line following End while. If the condition is not satisfied during the first pass through the loop, the instructions within the loop will not execute at all.

In pseudocode, the loop is written as follows:

While *<condition>* ***do***
    Repeating instructions . . .
End while

In a flowchart, the while loop can be represented by a decision and a connector, as seen in Figure 51, or by a hexagon with two inputs and two outputs, as shown in Figure 52. The first form is how it has traditionally been used; in this case, the loop is not explicit in the diagram and is only identified when the algorithm is executed step by step. In the second form, using a symbol that denotes iteration, the loop is evident. The DFD program commonly used for creating flowcharts utilizes the second form, which is likely why the iteration symbol is becoming increasingly common.

In a Nassi-Shneiderman diagram, it is represented by a box for the loop definition and internal boxes for the instructions that repeat, as shown in Figure 53.

Once the representation of this iterative structure is understood, it is necessary to learn how to use it in algorithm design. Some examples are presented for this purpose.

Figure 51. *Representation of the While Loop in a Flowchart (version 1)*

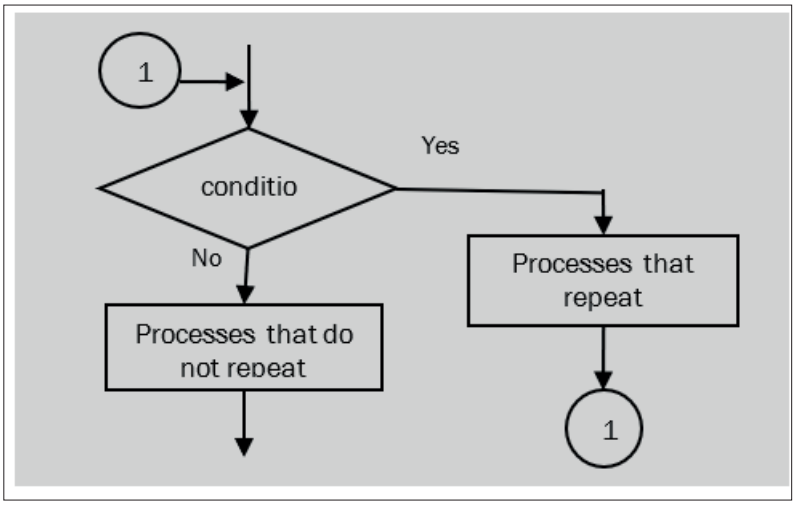


Figure 52. *Representation of the While Loop in a Flowchart (version 2)*

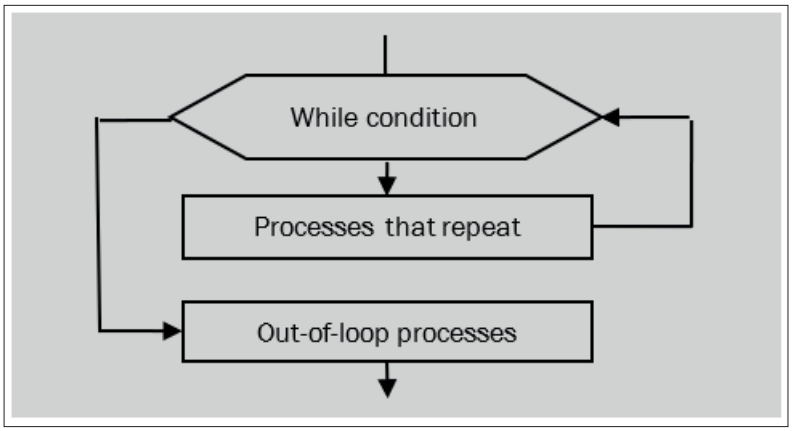
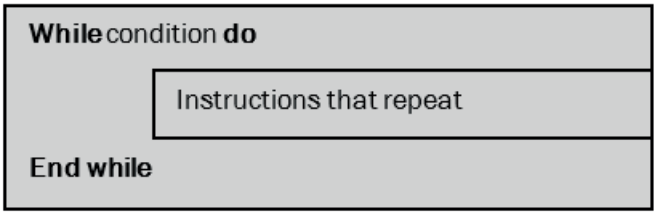


Figure 53. *Representation of the While Loop in a N-S Diagram*



Example 26. Generating Numbers

This algorithm uses iterations to generate and display numbers from 1 to 10.

The first step is to declare an integer variable for loop control, which will also be used to display the numbers. The variable is initialized to 1, then the loop is defined. Within this loop, the number is displayed, and the variable (counter) is incremented. This way, the three fundamental operations for controlling a loop using a counter variable are carried out: it is initialized, evaluated in the loop definition, and updated progressively towards 10, which will end the loop. The pseudocode is presented in Table 51, the flowchart in Figure 54, and the Nassi-Shneiderman diagram in Figure 55.

Table 51. *Pseudocode of the Algorithm for Generating Numbers*

```
1    Begin
2        Integer: num = 1
3        While num <= 10 do
4            Write num
5            num = num + 1
6        End while
7    End algorithm
```

In the pseudocode, the loop is located between lines 3 and 6; line 3 defines the loop, and line 6 ends it. Lines 4 and 5 correspond to the loop body; that is, the instructions that repeat. Since the ***while*** loop requires the variable to be initialized, this operation is performed in line 2. In this particular case, each time execution reaches line 6, it will return to line 3 to evaluate the loop condition. If the condition is true, lines 4 and 5 will execute; if false, execution jumps to line 7.

Figure 54. *Flowchart of the Algorithm for Generating Numbers*

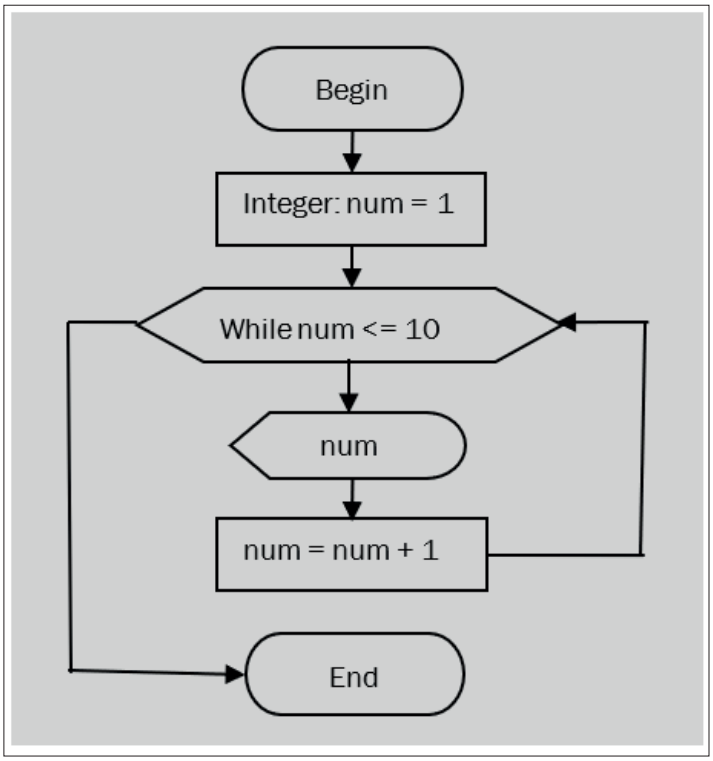
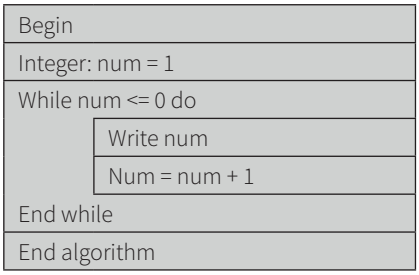


Figure 55. *N-S Diagram of the Algorithm for Generating Numbers*



The flowchart shows more clearly the behavior of the iterative structure. When declaring the variable it is assigned an initial value (1). In the loop definition, the condition (num <= 10) is set; if this condition is met, execution continues down the flow, the variable is shown and increased by one, and then it returns to the loop definition to evaluate the condition, If true, the instructions are executed again; if false, it exits to the left and goes to the end of the algorithm. The loop ends when num = 11.

In the Nassi-Shneiderman diagram, the loop is also easily identifiable since the entire structure is contained within a single box, and the loop body appears internally. Only when the loop has finished does execution move to the next box.

To verify the operation of this algorithm, a desk checking is performed, the results of which are shown in Table 52.

Table 52. *Verification of the Algorithm for Generating Numbers*

Iteration	Num	Result
	1	
1	2	1
2	3	2
3	4	3
4	5	4
5	6	5
6	7	6
7	8	7
8	9	8
9	10	9
10	11	10

Example 27. Divisors of a Number

An algorithm is required to display the divisors of a number in descending order.

Before starting to solve the exercise, it is necessary to clarify what is meant by a divisor of a number.

Given two integers a and b, b is said to be a divisor of a if, when performing an integer division a/b, the remainder is 0. For example, if we take the numbers 27 and 3, we can confirm that 3 is a divisor of 27 since when we divide 27/3 = 9, and the remainder is 0. For simplicity, the remainder of an integer division can be directly obtained using the Mod operator (arithmetic operators are covered in Section 2.3.1)

Returning to the exercise, it requires displaying all divisors of a number starting from the highest; therefore, the solution involves taking the numbers from the input number down to one and checking each value to see if it meets the condition of being a divisor of the number. To do this, it is necessary to use a control variable that starts at the number and decreases by one until it reaches 1.

For example, if we take the number 6, we must examine all the numbers between 6 and 1 and select those that are divisors of six, as shown below:

Number	Counter	Number Mod Counter	Divisor
6	6	0	Yes
	5	1	No
	4	2	No
	3	0	Yes
	2	0	Yes
	1	0	Yes

From the last column, we can see that the divisors of six, in descending order, are: 6, 3, 2, 1.

In a second example, if we take the number 8, we generate the list of numbers from eight down to one and check each one to see if it is a divisor of eight.

Number	Counter	Number Mod Counter	Divider
8	8	0	Yes
	7	1	No
	6	2	No

	5	3	No
	4	0	Yes
	3	2	No
	2	0	Yes
	1	0	Yes

Thus, the divisors of 8, in descending order, are: 8, 4, 2, 1.

As shown in the previous examples, the key to finding the divisors is to generate the list of numbers, and for that, it is appropriate to use an iterative structure and a counter variable, as shown in the pseudocode in Table 53.

Table 53. *Pseudocode for the Divisor Algorithm*

```
1     Begin
2         Integer: num, cou
3         Read num
4         cou = num
5         While cou >= 1 Do
6             If num Mod cou = 0 then
7                 Write cou
8             End if
9             cou = cou – 1
10        End while
11    End algorithm
```

The importance of using a loop to generate a list of numbers has already been mentioned, but not all numbers are shown since only those that meet the condition of being divisors of the input number are required. The control variable of the loop is initialized to the input number, the loop runs while this variable is greater than 1 (line 5), and the variable is decremented by one (line 9). Table 54 shows the results of the verification with the numbers 6 and 8.

Table 54. *Verification of the Divisor Algorithm*

Execution	num	cou	Output
1	6	6	6
		5	
		4	

		3	3
		2	2
		1	1
		0	
2	8	8	8
		7	
		6	
		5	
		4	4
		3	
		2	2
		1	1
		0	

Flowchart and N-S diagram representations are presented in Figures 56 and 57.

Figure 56. *Flowchart of the Divisor Algorithm*

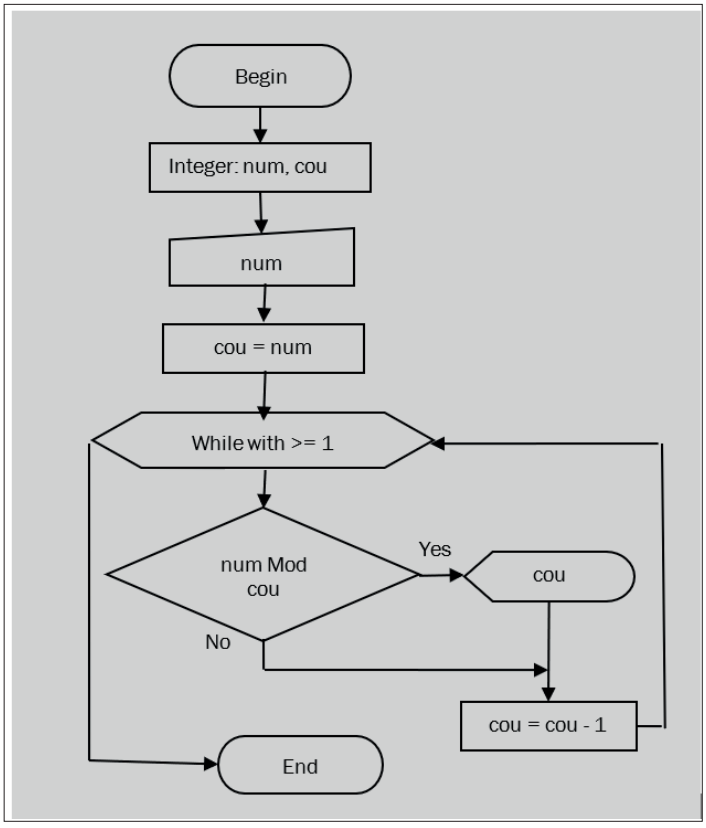
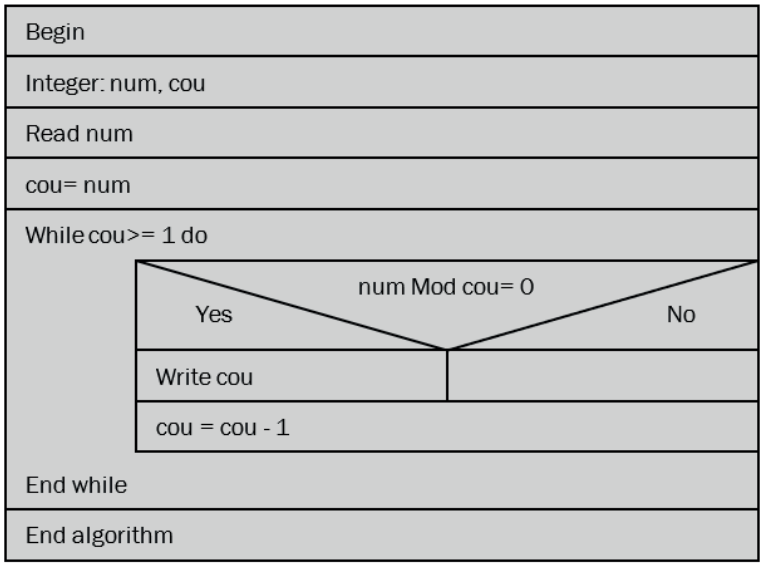


Figure 57. *N-S Diagram of the Divisor Algorithm*



4.3.3 DO WHILE Structure

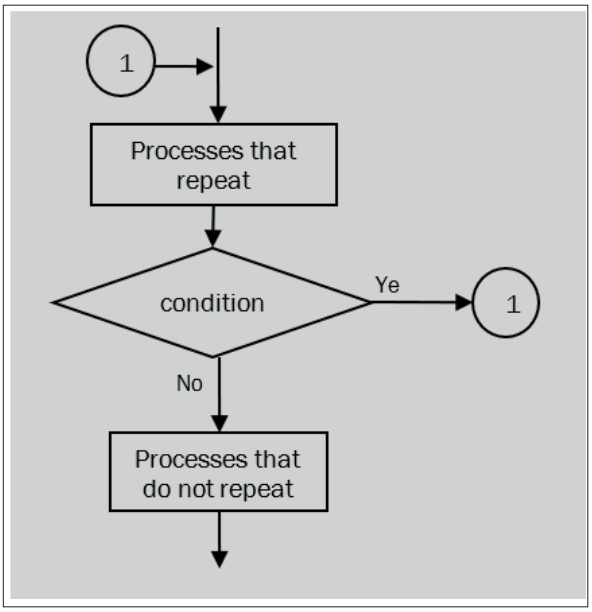
This structure is proposed as an alternative to the ***Repeat Until*** loop, as the latter is no longer present in modern programming languages. The ***Do While*** loop allows a set of instructions to be executed repeatedly, with the particularity that it evaluates the condition controlling the loop after each iteration. This means that the first evaluation of the condition occurs after executing the loop's instructions, ensuring that the body of the loop is executed at least once.

When the word ***Do*** is found in a program, the following lines continue to execute. When the ***While*** statement is found, the associated condition is evaluated, which must return a logical value (true or false). If the value is true, control returns to the ***Do*** instruction, and the subsequent instructions are executed again. If the condition is false, execution continues at the instruction following ***While***. In pseudocode, it is written as follows:

```
Do
    Instruction 1
    Instruction 2
    . . .
    Instruction n
While <condition>
```

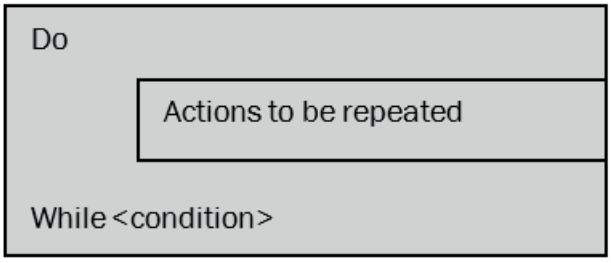
In flowcharts, there is no specific symbol to represent this type of structure; therefore, it is implemented using a decision symbol and a connector to return to a previous process and repeat the sequence while the condition is true, as shown in Figure 58.

Figure 58. *Do While Loop in Flowchart*



In N-S diagrams, a box is used to represent the loop, and internal boxes are used for the actions that make up the body of the loop, as presented in Figure 59.

Figure 59. *Do While Loop in N-S Diagram*



Example 28. Sum of Positive Integers

This algorithm reads integer numbers and sums them as long as they are positive. When a negative number or zero is entered, the loop ends, the total sum is displayed, and the algorithm terminates. The pseudocode is shown in Table 55.

Table 55. *Pseudocode for the Sum of Integers Algorithm*

```
1    Begin
2        Integer: num = 0, sum = 0
3        Do
4            sum = sum + num
5            Read num
6        While num > 0
7        Write Sum
8    End algorithm
```

In this algorithm, two variables are declared and initialized since the first instruction to be executed is the sum, and it is necessary to control the content of the variables. In line 5, a number is read, but to perform the sum, it is necessary to go back, which is subject to the condition of being a positive integer. If the condition is met, execution returns to line 3, performs the sum in line 4, and reads a number again, continuing this way until zero or a negative number is entered. Only after the loop ends is the result of the sum displayed, and execution stops. Figure 60 presents the flowchart, and Figure 61 shows the N-S diagram.

Figure 60. *Flowchart of the Sum of Integers Algorithm*

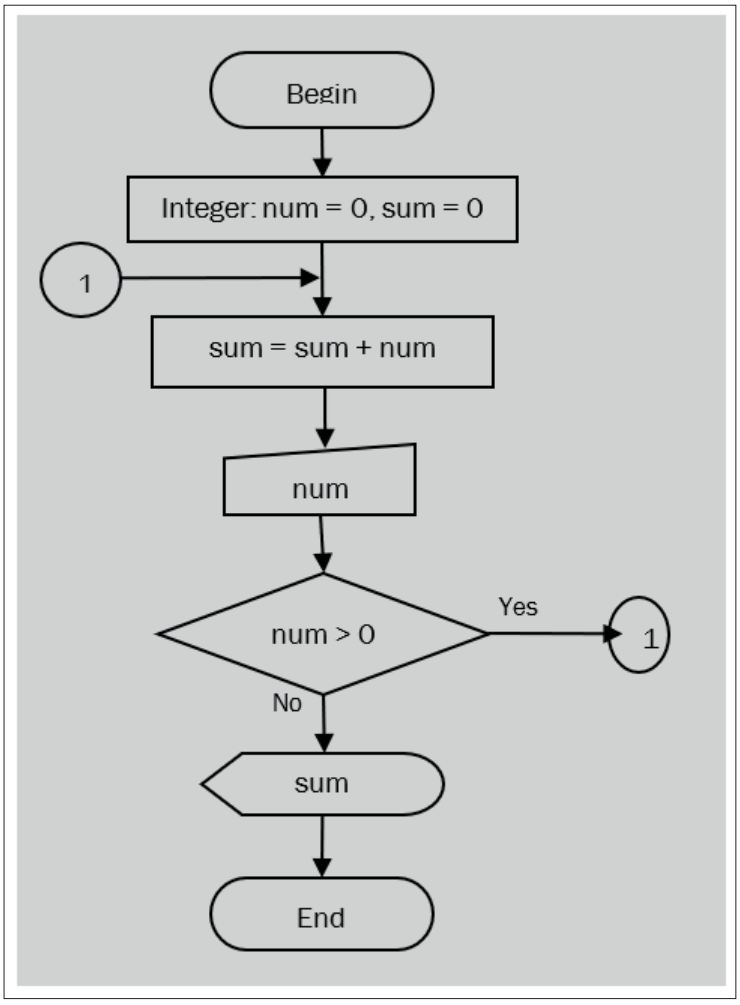


Figure 61. *N-S Diagram of the Sum of Integers Algorithm*

Begin	
Integer: num = 0, sum = 0	
Do	
	sum = sum + num
	Read num
While num > 0	
End algorithm	

Table 56 shows the results of verifying the algorithm.

Table 56. *Verification of the Sum of Integers Algorithm*

Iteration	Num	Sum	num > 0	Output
	0	0		
1	4	4	T	
2	8	12	T	
3	1	13	T	
4	5	18	T	
5	0		F	
				18

Example 29. Binary Number

Given a number in base 10, this algorithm calculates its equivalent in base 2; that is, it converts a decimal to binary.

One way to convert a base 10 (decimal) value to base 2 (binary) is to divide the value by 2 and take the remainder, which can be 0 or 1, then take the quotient and divide it by 2 again. This operation is performed repeatedly until the result is 0. However, this solution has a slight difficulty to resolve: the remainders form the binary number but must be read in reverse order; that is, the first number obtained occupies the least significant position while the last should occupy the first position in the binary number.

Consider the number 20

20 / 2 = 10 Remainder = 0
10 / 2 = 5 Remainder = 0
5 / 2 = 2 Remainder = 1
2 / 2 = 1 Remainder = 0
1 / 2 = 0 Remainder = 1

If the numbers are taken in the order they are generated, we obtain: 00101, which equals 5 in decimal, far from the original value of 20.

The numbers generated are taken in reverse order; the last remainder is the first digit of the binary number. Therefore, the binary number generated from this conversion is: 10100. This does equal 20.

Now, the question is: how to ensure that each new digit obtained is placed to the left of the previous ones?

To solve this issue, a variable is used to keep track of the position that the digit should occupy in the new number. The variable starts at 1 and is multiplied by 10 in each iteration to indicate an increasingly significant position. Thus, we have:

Input: number in base 10 (decimal)
Output: number in base 2
Process:
  digit = decimal Mod 2
  binary = binary + digit * position
  decimal = decimal / 2
  position = position * 10

The solution to this problem is presented in Table 57.

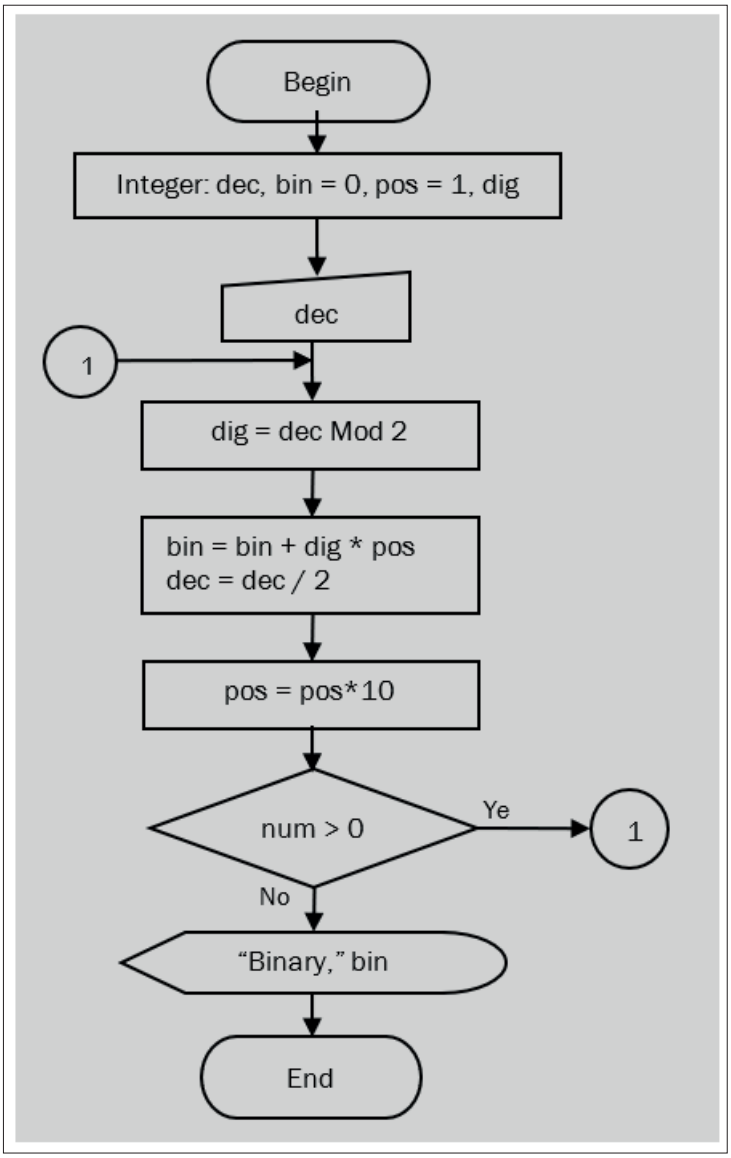
Table 57. *Pseudocode for the Binary Number Algorithm*

```
1     Begin
2         Integer: dec, bin = 0, pos = 1, dig
3         Read dec
4         Do
5             dig = dec Mod 2
6             bin = bin + dig * pos
7             dec = dec / 2
8             pos = pos*10
9         While dec > 0
10        Write "Binary:", bin
11    End algorithm
```

The algorithm shows that the operations in lines 5 to 8 repeat while there is a number to divide (***dec > 0***). In line 5, a digit (0 or 1) is obtained corresponding to the remainder of dividing the decimal number by two. That digit is multiplied by the power of 10 to convert it into 10, 100, 1000, or any other power when it is a one, and then added to the digits obtained previously (line 6). The decimal number is halved each time, tending toward zero (line 7), while the power indicating the position of the next one is multiplied by 10 in each iteration (line 8).

The flowcharts and N-S diagrams for this algorithm are presented in Figures 62 and 63, respectively.

Figure 62. *Flowchart of the Binary Number Algorithm*



To test this algorithm, the binary equivalents for the numbers 20 and 66 is calculated, yielding results of 10100 and 1000010, respectively, which are correct. The data generated from the tests are presented in Table 58.

Figure 63. *N-S Diagram of the Binary Number Algorithm*

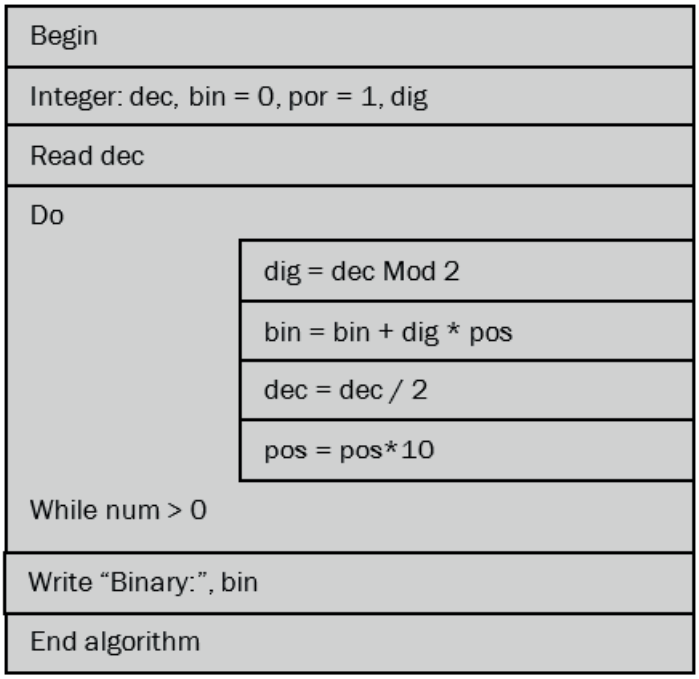


Table 58. *Verification of the Binary Number Algorithm*

Iteration	dig	dec	Bin	pos	Output
			0	1	
		20	0		
1	0	10	0	10	
2	0	5	0	100	
3	1	2	100	1000	
4	0	1	100	10,000	
5	1	0	10100	100000	Binary: 10100
		66	0	1	
1	0	33	0	10	
2	1	16	10	100	
3	0	8	10	1000	
4	0	4	10	10,000	
5	0	2	10	100000	
6	0	1	10	1000000	
7	1	0	1000010		Binary: 1000010

4.3.4 FOR Structure

This structure, like the previous ones, allows for repeated execution of a single instruction or a group of instructions. However, unlike other repetition instructions, it manages the initial value, the increment or decrement value, and the final value of the control variable as part of the loop definition.

When a ***for*** instruction is encountered during the execution of an algorithm, the control variable (counter) takes the initial value, it is verified that this value does not exceed the final value, and then the loop's instructions are executed. Finding the ***end for*** statement, the increment occurs and it is verified again that the control variable has not surpassed the allowed limit, and the instructions within the loop are executed repeatedly until the final established value is exceeded.

The ***for*** loop terminates when the control variable (counter) surpasses the final value; that is, equality is allowed, and instructions execute when the counter equals the final value.

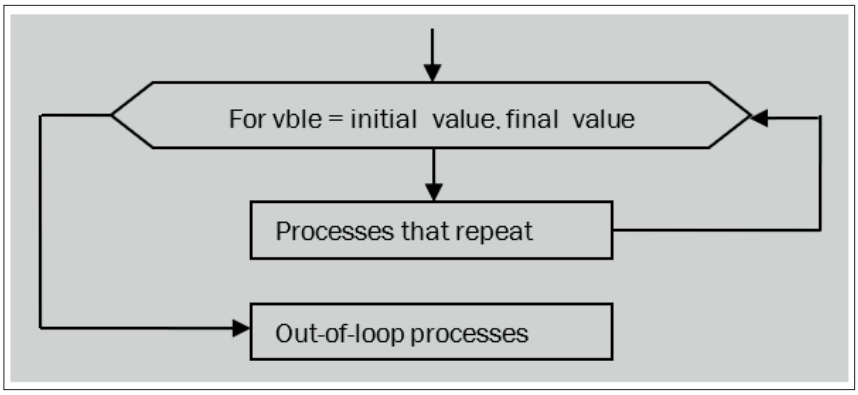
Some programming languages define the syntax for this loop by including a condition that must be met, similar to the ***While*** loop, instead of a final value for the variable, as proposed in pseudocode.

This loop can be presented in three ways: the first is the most common, where an increment of 1 occurs in each iteration, in which case it is not necessary to explicitly write this value. In pseudocode, it is expressed as follows:

```
For variable = initial_value to final_value do
    Instructions
End for
```

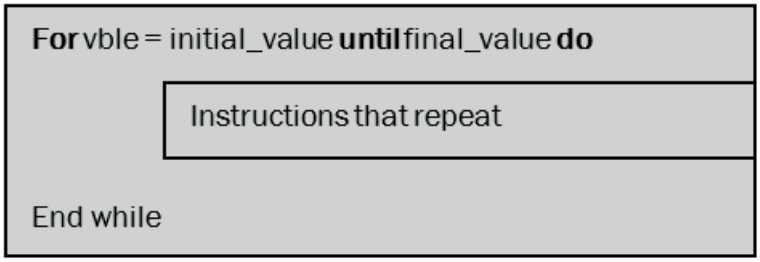
In flowchart representation, it appears as shown in Figure 64, and in the N-S diagram as shown in Figure 65.

Figure 64. *For Loop in Flowchart (Version 1)*



In the flowchart, it is not necessary to write all the words; these are replaced by commas (,) except the first one.

Figure 65. *For Loop in N-S Diagram (Version 1)*



The second case of using the ***for*** loop arises when the increment is different from 1, in which case the word ***increment*** followed by the value to be added in each iteration will be written.

In pseudocode, it is written as:

For *vble = initial_value* ***to*** *final_value* ***increment*** *value* ***do***
    Instructions that repeat
End for

F Flowchart and N-S diagram representations appear as in Figures 66 and 67.

Figure 66. *For Loop in Flowchart (Version 2)*

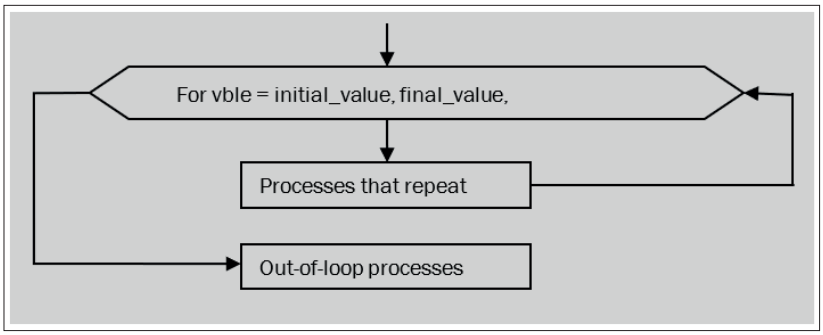
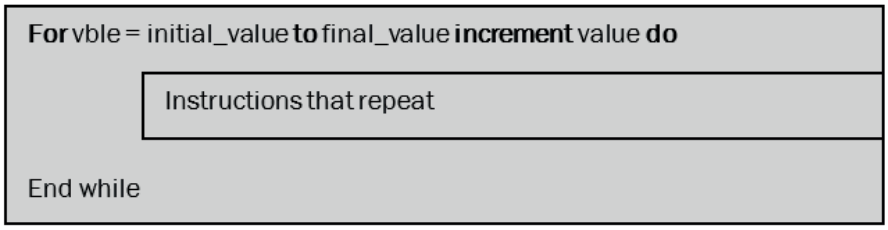


Figure 67. *For Loop in N-S Diagram (Version 2)*



The third case occurs when the for loop does not increment from an initial value to a higher value, but rather decreases from a higher initial value to a lower value. To do this, it is enough to write decrement instead of increment.

In pseudocode, it looks like this:

For counter = initial_value to final_value decrement value do
    Instructions
End for

In this case, for the first iteration to execute, the initial value must be greater than the final value; otherwise, it will simply skip to the instruction following the *end* for.

It is important to note that when the control variable must decrease in each iteration, *decrement* and the value must always be written, even if it is -1.

In flowchart representation, it will be enough to precede the value to be decremented with a minus sign (-).

Example 30. Iterative Sum

Given a positive integer n, the sum of the numbers from 1 to n is calculated and shown.

In this case, an iterative structure is required to generate the numbers from one to ***n***, and each value in this interval is stored in an accumulator-type variable.

For example: If the number 3 is entered, the sum will be:

$$1 + 2 + 3 = 6$$

If the number 6 is entered, the sum will be:

$$1 + 2 + 3 + 4 + 5 + 6 = 21$$

The exercise can be organized as follows:

Input: number
Output: sum
Process:

$$\text{sum} = \sum_{i=1}^{i=n} I$$

The solution to this exercise is presented in Table 59 in pseudocode notation, with flowchart and N-S diagrams shown in Figures 68 and 69.

Table 59. *Pseudocode for the Sum Algorithm*

```
1   Begin
2       Integer: num, i, sum = 0
3       Read num
4       For i = 1 to num do
5           sum = sum + i
6       End for
7       Write "Sum:", sum
8   End algorithm
```

Figure 68. *Flowchart of the Sum Algorithm*

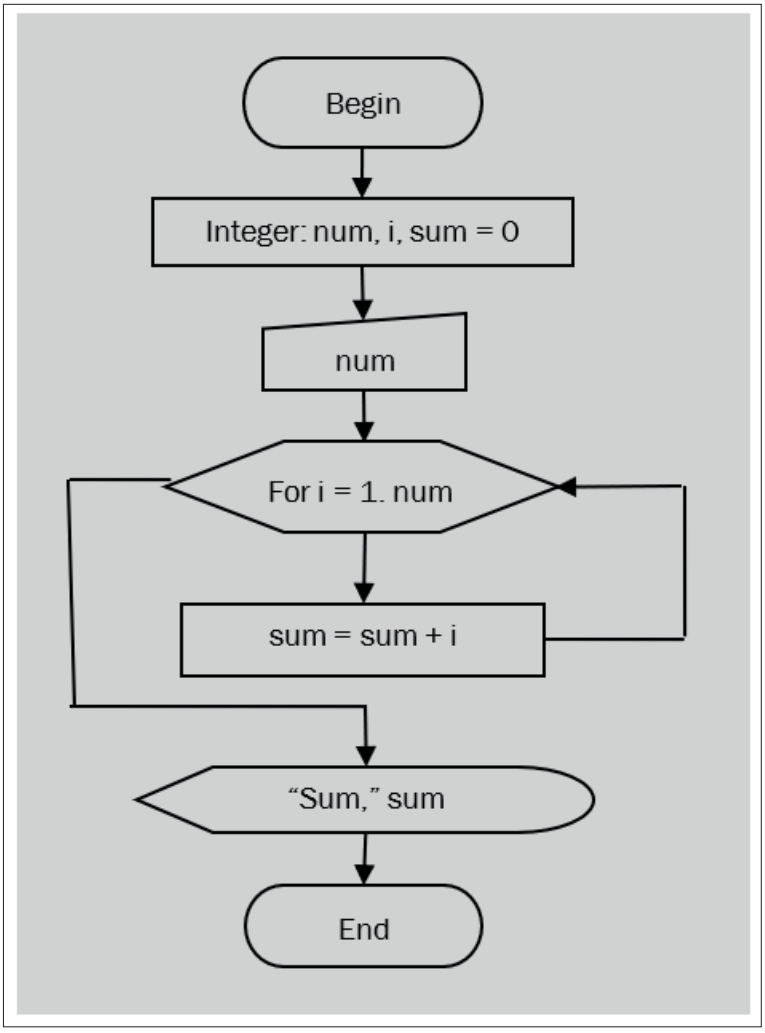
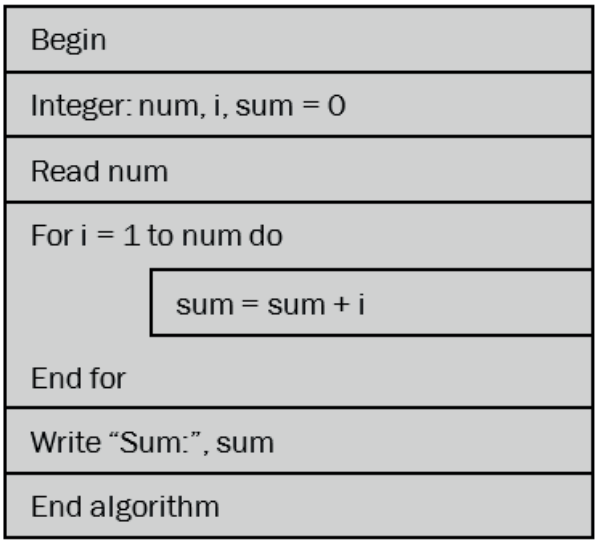


Figure 69. *N-S Diagram of the Sum Algorithm*



Since the ***for*** loop automatically increments the variable, the only instruction within the loop is to sum the generated numbers.

Table 60 shows the behavior of the variables when executing the algorithm to calculate the sum of the numbers from 1 to 6.

Table 60. *Verification of the Sum Algorithm*

Iteration	Num	i	sum	Output
			0	
	6		0	
1		1	1	
2		2	3	
3		3	6	
4		4	10	
5		5	15	
6		6	21	Sum: 21

Example 31. Multiplication Table

Commonly known as the multiplication table of a number, this is a list of the first 10 multiples. For example, the table for 2:

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20

To display it this way, the iterative for structure can be used, which varies the multiplier from 1 to 10, calculating the product and displaying the data in each iteration. The pseudocode is shown in Table 61, with flowchart and N-S diagrams in Figures 70 and 71.

Table 61. *Pseudocode for the Multiplication Table Algorithm*

```
1    Begin
2        Integer: m, n, r
3        Read m
4        For n = 1 to 10 do
5            r = m * n
6            Write m, '*', n, '=', r
7        End for
8    End algorithm
```

Figure 70. *Flowchart of the Multiplication Table Algorithm*

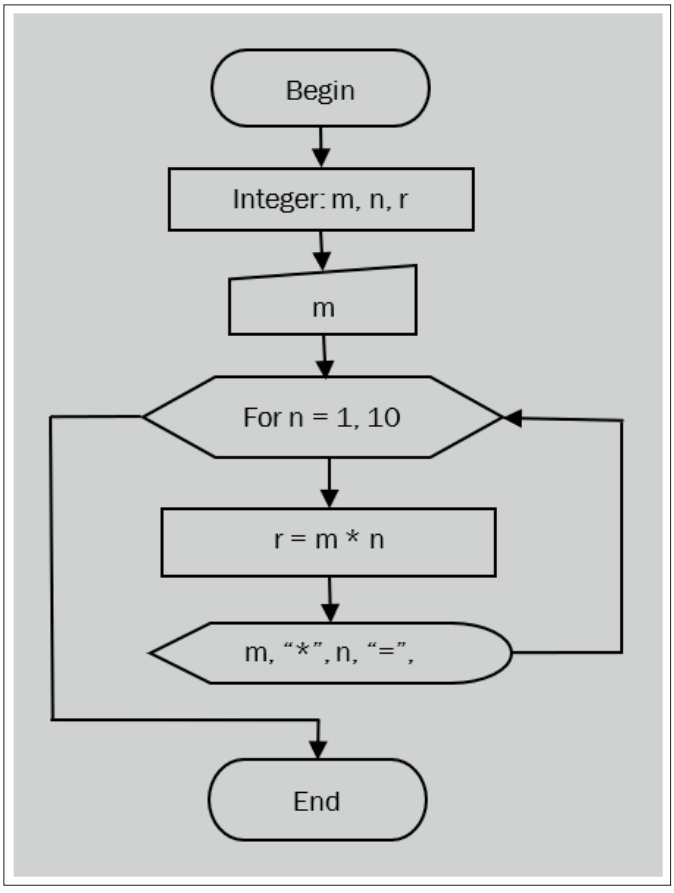


Figure 71. *N-S Diagram of the Multiplication Table Algorithm*

Begin
Integer: m, n, r
Read m
For n = 1 to 10 do
r = m * n
Write m, "*", n "=", r
End for
End algorithm

To verify the algorithm's functionality, the multiplication table for 4 is generated, with the results presented in Table 62.

Table 62. *Verification of the Multiplication Table Algorithm*

Iteration	m	n	r	Output
	4			
1		1	4	4 * 1 = 4
2		2	8	4 * 2 = 8
3		3	12	4 * 3 = 12
4		4	16	4 * 4 = 16
5		5	20	4 * 5 = 20
6		6	26	4 * 6 = 24
7		7	28	4 * 7 = 28
8		8	32	4 * 8 = 32
9		9	36	4 * 9 = 36
10		10	40	4 * 10 = 40

4.3.5 Nested Iterative Structures

Iterative structures, like selective ones, can be nested; that is, one can be placed inside another.

The nesting of loops consists of an outer loop and one or more inner loops, where each time the outer loop repeats, the inner loops reset and execute all their defined iterations (Joyanes, 2000).

A clear example to illustrate the concept of nested loops is a digital clock. Time is measured in hours, minutes, and seconds; hours consist of minutes, and minutes consist of seconds. Therefore, if you write a loop for the hours, another for the minutes, and another for the seconds, the minute loop in each iteration must wait for the 60 iterations of the second loop (0..59) to complete, and the hour loop must wait for the minute loop (0..59) to finish.

Example 32. Digital Clock

To implement a digital clock, it is necessary to declare three variables to control: hours, minutes, and seconds. Each of these variables controls a loop, as follows: the seconds increment from 0 to 59, the minutes also from 0 to 59, and the hours from 1 to 12 or from 1 to 24.

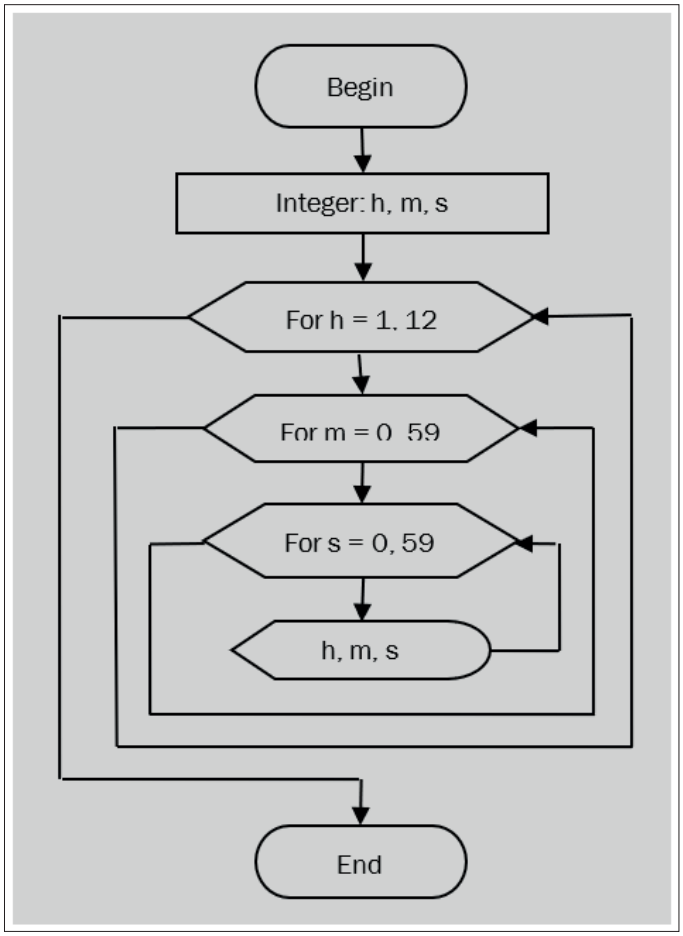
The variable that controls the hours increments by one each time the variable that controls the minutes has completed its cycle from 0 to 59. Similarly, the variable of the minutes increments by one each time the variable of the seconds has completed its cycle from 0 to 59. For this to happen, the hour loop must contain the minute loop, and the minute loop must contain the second loop, as shown in Table 63.

Table 63. *Pseudocode for the Digital Clock Algorithm*

```
1     Begin
2         Integer: h, m, s
3         For h = 1 to 12 do
4             For m = 0 to 59 do
5                 For s = 0 to 59 do
6                     Write h,":", m, ":", s
7                 End for
8             End for
9         End for
10    End algorithm
```

This same algorithm expressed in a flowchart is presented in Figure 72 and in a N-S diagram in Figure 73.

Figure 72. *Flowchart of the Digital Clock Algorithm*



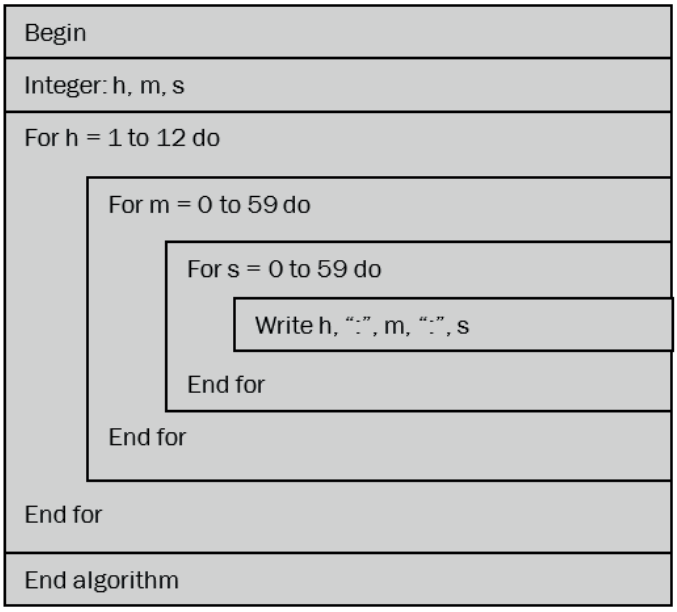
When executing this algorithm, the output would look like this:

```
1:0:0
1:0:1
1:0:2
…
1:0:58
1:0:59
1:1:0
1:1:1
1:1:2
```

```
...
1:59:58 AM
1:59:59 AM
2:0:0
2:0:1
2:0:2
    ...
```

This algorithm is designed to end execution when it reaches 12:59:59. To make the clock run indefinitely, it would be necessary to place the three loops inside another loop, so that when it reaches 12:59:59, the variable h resets and the count starts again. The outer loop should be an infinite loop of the form: *while true do.*

Figure 73. *N-S Diagram for the Digital Clock Algorithm*



Example 33. Nine Multiplication Tables

In a previous example, a loop was used to generate the multiplication table of a single number; in this case, nested loops are used to generate the tables from two to 10. The first loop is associated with the multiplicand and refers to each of the tables to be generated, so it takes values between 2 and 10. The inner loop is related to the

multiplier, responsible for generating the multiples in each of the tables, thus taking values between 1 and 10. The pseudocode is presented in Table 64.

Table 64. *Pseudocode for Nine Multiplication Tables*

```
1    Begin
2        Integer: m, n, r
3        For m = 2 to 10 do
4            Write “Multiplication table for”, m
5            For n = 1 to 10 do
6                r = m * n
7                Write m, “*”, n, “=”, r
8            End for
9        End for
10   End algorithm
```

When executing this algorithm, the output would look like this:

```
Multiplication table for 2
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
…
Multiplication table for 3
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
…
And so on until

 10 * 10 = 100
```

The flowcharts and N-S diagrams for this algorithm are presented in Figures 74 and 75.

Figure 74. *Flowchart for Nine Multiplication Tables*

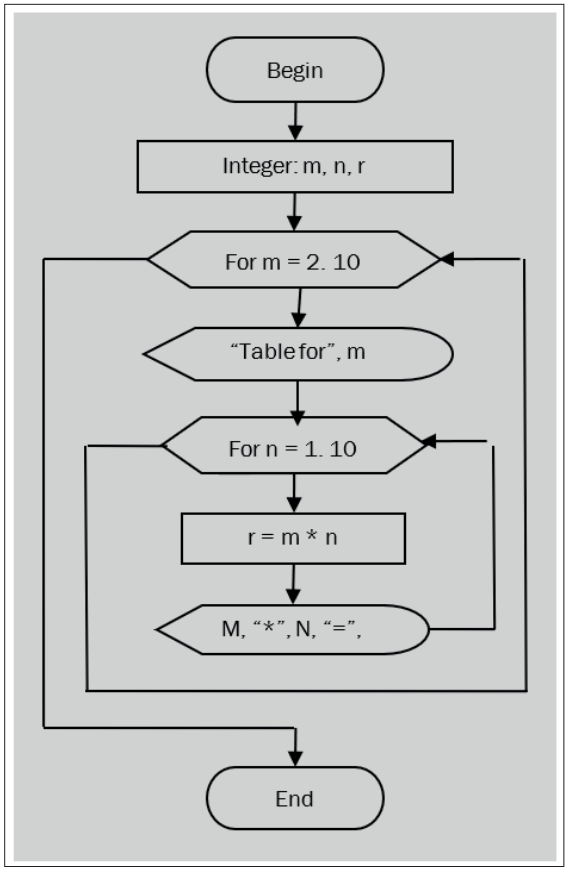
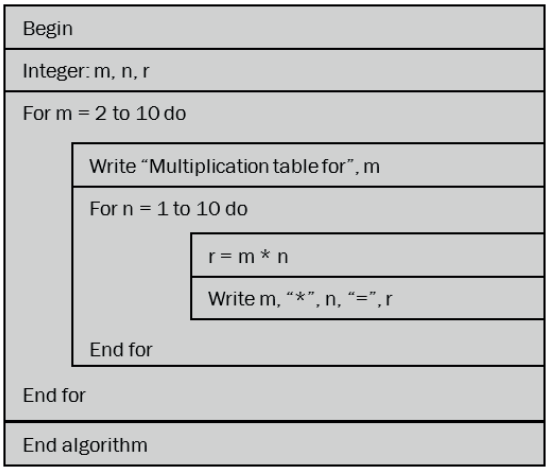


Figure 75. *N-S Diagram for Nine Multiplication Tables*



4.3.6 More Examples of Iteration

Example 34. Minimum, Maximum, and Average of n Numbers

This exercise consists of reading *n* numbers and then reporting the minimum and maximum numbers from the list, as well as calculating the average value.

Suppose the user inputs the following list of data:

2
14
55
6
12
34
17

We have:
Number of inputs: 7
Maximum number: 55
Minimum number: 2
Sum: 140
Average = 140 / 7 = 20

To solve this problem, the first step is to determine how to know when to stop reading data, as the number of inputs is not specified, and there are no conditions given in the problem statement to indicate when to end the loop. This type of problem is common when the amount of data to process is not known in advance.

There are at least three simple ways to implement loops when the number of iterations is unknown: The first approach is to define a variable to store this data (n) and ask the user to input it, for example, by asking "***how many numbers would you like to enter?"*** The inputted data is then used as a final value to control the loop; the second approach is to use a switch or flag, which involves defining a variable with two possible values: yes and no or 0 and 1. For example, asking the user "***Another number (Y/N) ?"***, if the user inputs "***Y,"*** data reading continues; otherwise, the loop ends. The third approach is to use a sentinel value that ends the loop when a condition is met. For example, the loop ends when the number 0 is entered, presenting the user with a message like: "***Enter a number (0 to end):*** ".

To solve this exercise, the first proposal is chosen. The following information is available:

Input data: number of inputs, numbers
Output data: average, maximum, and minimum

Processes:
sum = sum + number
Average = sum / number of inputs

To identify the minimum and maximum numbers, we can choose between two alternatives: the first is to initialize the minimum variable with a very large number and the maximum variable with 0 so that they get updated in the first iteration; the second is to not initialize the variables and assign the first input value to both the maximum and minimum variables. In this case, we opted for the second approach. The algorithm is presented in Table 65.

In this algorithm, the variables: ***numin, num, max, min,*** and ***avg*** are working variables; ***count*** is the counter that controls the loop, and ***sum*** is the accumulator that totals the numbers entered to later calculate the average.

It is important to understand which instructions should be inside the loop and which should be outside, either before or after. There is no golden rule that applies to all cases, but it can be helpful to remember that everything inside the loop repeats; that is, you should ask yourself how many times the instruction needs to be executed. If the answer is only once, there is no reason for it to be inside the loop. In this example, reading the number of inputs should only execute once since this quantity will establish the limit of repetitions, and calculating the average should also only be done once; therefore, these actions are placed outside the loop, while adding the entered number must occur as many times as numbers are entered, hence this instruction appears within the loop.

Table 65. *Pseudocode for Minimum, Maximum, and Average*

```
1   Begin
2     Integer: numin, num, sum=0, min, max, count=0
3     Real: avg
4     Read numin
5     While count < can do
6         Read num
7         sum = sum + num
8         If count = 0 then
9             min = num
10            max = num
11        Else
12            If num < min then
13                max = num
14            End if
15            If num > max then
16                max = num
17            End if
18        End if
19        count = count + 1
20    End while
21    avg = sum / numin
22    Write “Minimum number:”, min
23    Write “Average:”, avg
24    Write “Maximum number:”, max
25  End algorithm
```

The loop will execute as long as ***count*** is less than the value of the variable ***numin***. Since ***count*** starts at 0, the loop runs for any value of ***numin*** that is greater than or equal to 1. However, if the value entered for ***numin*** is 0, the loop would not run, leading to a division by zero error when calculating the average. The last instruction in the loop is ***count*** = ***count + 1***, which is essential since the counter must increment with each iteration to reach the value of the ***numin*** variable. It is important to remember that every loop must have a limited number of iterations; therefore, when designing the algorithm, you must anticipate how the loop will end.

Table 66 presents the behavior of the variables and the results of an implementation.

Table 66. *Verification of the Algorithm for Minimum, Maximum, and Average*

Iteration	Numin	num	min	Max	Count	sum	avg	Output
					0	0		
	7							
1		2		2	1	2		
2		14	2	14	2	16		
3		55	2	55	3	71		
4		6	2	55	4	77		
5		12	2	55	5	89		
6		34	2	55	6	123		
7		17	2	55	7	140		
							20	Minimum number: 2 Average = 20 Maximum number: 55

Example 35. Grading Process

In a group of ***n*** students, three partial evaluations were conducted. The goal is to calculate each student's final grade and determine the number of students who passed, the number who failed, and the average grade for the group, considering that the percentage weight for each evaluation was agreed upon between the teacher and the group, and the minimum passing grade is 3.0.

Using the strategy proposed to understand the problem, let's assume a specific case, such as this one:

Second semester group: 20 students
Course: Programming
Percentage for first evaluation: 30%
Percentage for second evaluation: 30%
Percentage for third evaluation: 40%

Now, let's find the solution for one student:

Student Pedro Pérez receives the following grades:
Evaluation 1: 3.5

Evaluation 2: 4.0
Evaluation 3: 2.8
What's Pedro's final grade?

Applying the previously mentioned percentages we have:

Final grade = 3.5 * 30% + 4.0 * 30% + 2.8 * 40%
Final grade = 1.05 + 1.2 + 1.12
Final grade = 3.4

The next step is to decide whether the student passes or fails the course:

Final grade >= 3.0? (3.4 >= 3.0?) Yes

Pedro Pérez passed the course; consequently, he is counted as one of those who passed. To calculate the course average, his grade needs to be added to an accumulator variable, which will ultimately be divided by the total number of students (20).

This same procedure must be carried out for each student in the group.

Next, we identify and classify the problem data:

Input data: student name, grade1, grade2, grade3, percentage1, percentage2, percentage3.

Output data: final grade, average, number of students passing, number of students failing

Process:
final grade = grade1 * percentage1 + grade2 * percentage2 + grade3 * percentage3
sum = sum + final grade
Average = sum / number of students

In the analysis of the previous exercise, it was mentioned that there are three ways to program a loop when the number of iterations is not known in advance, and the first approach was implemented. In this example, the second approach will be used, which consists of asking the user if they want to continue entering data. The loop ends when the user responds negatively.

This algorithm uses three counters and an accumulator, in addition to working variables. The variables used to record the number of students, the number of students passing, and the number failing the course are counters, while the variable used to total the final grades is an accumulator. The loop is not controlled by a predefined number of students, but rather by the contents of the ans variable, which must take 'Y' or 'N' as values.

The design of the solution is presented in Figure 76 in flowchart notation, and the results from a test run with four students are shown in Table 67.

Figure 76. *Flowchart for Grading Process*

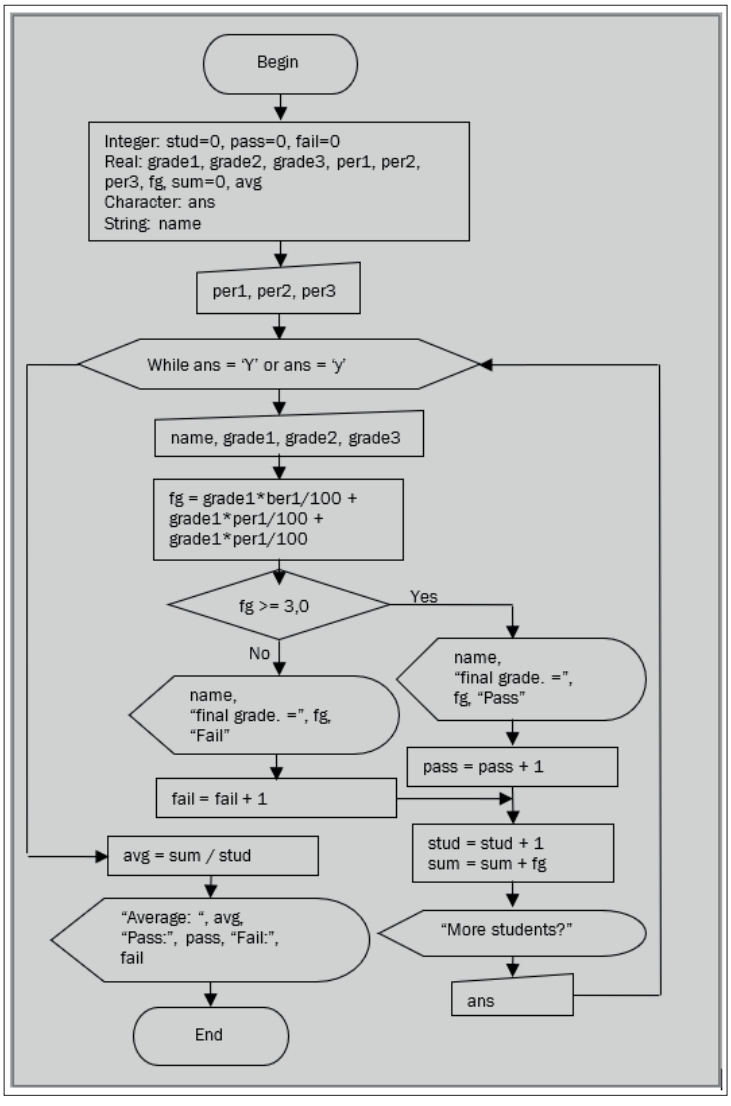


Table 67. *Verification of the Grading Process Algorithm*

per1	per2	Per3	name	Grade 1	Grade 2	Grade 3	Fg	Pass	Fail	stud	sum	ans	avg
								0	0	0	0	S	
30	30	40											
			A	3.0	3.5	4.0	3.5	1		1	3.5	S	
			B	2.0	2.5	2.0	2.5		1	2	6	S	
			C	4.0	3.2	2.0	3.0	2		3	9	S	
			D	2.2	3.5	2.5	2.7		2	4	11.7	n	2.9

The output on the screen is as follows:

```
Name: A
Final grade: 3.5
Passed

Name: B
Final grade: 2.5
Failed

Name: C
Final grade: 3.0
Passed

Name: D
Final grade: 2.7
Failed

Average: 2.9
Passed: 2
Failed: 2
```

Example 36. Fibonacci series

This well-known series, proposed by the Italian mathematician of the same name, corresponds to a mathematical model explaining the reproduction of rabbits and was first published in 1202 in a work entitled *Liber Abaci.*

Fibonacci posed the problem as follows: suppose a pair of rabbits produces two offspring each month, and each new rabbit begins reproducing after two months.

Thus, when acquiring a pair of rabbits, in the first and second months there will be one pair, but by the third month they will have reproduced, resulting in two pairs. In the fourth month, only the first pair reproduces, increasing the count to three pairs; in the fifth month, the second pair begins reproducing, leading to five pairs, and so on. If no rabbits die, the number of rabbits each month is given by the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Assuming the first two numbers are constants—specifically, that the first two numbers of the sequence are 0 and 1—the subsequent numbers can be obtained by summing the two previous ones as follows:

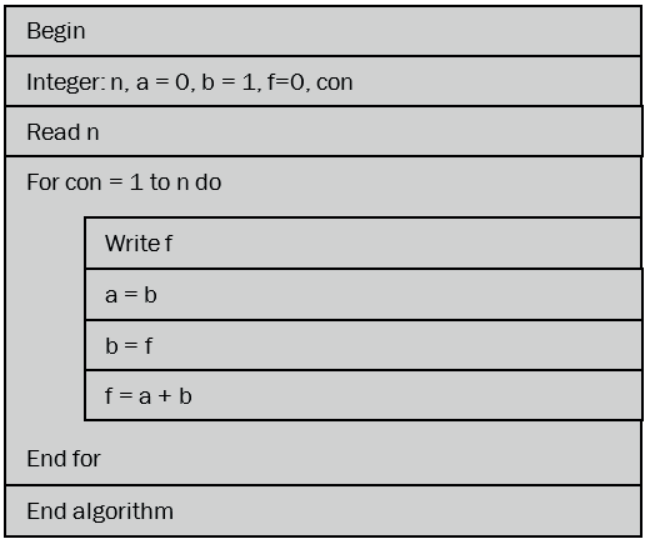
Term	Value	Obtained from
1	0	Constant
2	1	Constant
3	1	0 + 1
4	2	1 + 1
5	3	2 + 1
6	5	3 + 2
7	8	5 + 3
8	13	8 + 5
9	21	13 + 8
10	34	21 + 13

This topic is further discussed in Section 8.7, where a recursive solution is proposed.

In this example, the goal is to design an algorithm that generates the first n terms of this series.

To solve this exercise using an iterative structure, it is necessary, first, to determine how many numbers to generate—i.e., to know the value of n; second, since each number is obtained by summing the two previous terms, three variables are used: one for the generated value and two more to keep track of the last two data points. Additionally, a variable is defined to control the loop. The solution is presented through an N-S diagram in Figure 77.

Figure 77. *N-S Diagram for Fibonacci Series*



To observe the behavior of the variables, Table 68 presents the results of the algorithm verification, showing how the values change from one variable to another.

Table 68. *Verification of the Fibonacci Series Algorithm*

Iteration	n	Con	A	B	f	Output
	10	0	0	1	0	
1		1	1	0	1	0
2		2	0	1	1	1
3		3	1	1	2	1
4		4	1	2	3	2
5		5	2	3	5	3
6		6	3	5	8	5
7		7	5	8	13	8
8		8	8	13	21	13
9		9	13	21	34	21
10		10	21	34	55	34

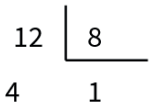
Example 37. Greatest Common Divisor

Given two integers, we need to find their greatest common divisor (GCD).

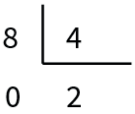
The GCD of two numbers is the largest number that divides them both; it can even be one of the numbers since every number is a divisor of itself. The GCD can be obtained iteratively or recursively. In this section, we solve it using the iterative approach, while Section 8.7 presents the recursive solution using the Euclidean algorithm.
When addressing this problem, the first idea that might come to the reader's mind is to break down the numbers into their divisors and take the common ones, as taught in school. That is a valid solution; however, it is difficult to implement algorithmically.

Using the Euclidean algorithm, the GCD is obtained as follows: divide the first number by the second; if the division is exact (remainder = 0), the GCD is the second number. If the division is not exact, divide the divisor by the remainder. If the modulus is zero, the GCD is the number that was placed as the divisor; otherwise, repeat this operation until an exact division is achieved.

For example, let's find the GCD of the numbers 12 and 8



Since the division is not exact, we perform a second division, taking the previous divisor as the dividend and the remainder as the divisor.



This second operation is an exact division (remainder = 0), so the divisor is the solution to the problem; that is, the GCD of 12 and 8 is 4.

Based on the previous example, we can organize the data of the exercise as follows:

Input data: a, b (two integers)
Output Data: GCD
Process: c = a Mod b

The solution is presented in pseudocode in Table 69

Table 69. *Pseudocode for the Greatest Common Divisor Algorithm*

```
1    Begin
2        Integer: num1, num2, a, b, c
3        Read num1, num2
4        a = num1
5        b = num2
6        Do
7            c = a Mod b
8            a = b
9            b = c
10       While c != 0
11       Write "GCD:", a
12   End algorithm
```

The verification of this algorithm with the numbers 12 and 8, and 6 and 20 generates the data presented in Table 70.

Table 70. *Verification of the Greatest Common Divisor Algorithm*

Execution	Iteration	a	b	c	Output
1		12	8		
1	1	12	8	4	
1	2	8	4	0	
1		4	0		GCD: 4
2		6	20		
2	1	6	20	6	
2	2	20	6	2	
2	3	6	2	0	
		2	0		GCD: 2

Example 38. Reverse Digits

Given an integer, we want to reverse the order of its digits. For example, taking the number 123, when reversed, it becomes 321.

To change the order of the digits, it is necessary to separate them starting from the last digit. This is done by dividing by 10 and taking the remainder using the modulo operator.

```
123 | 10
 3  | 12
```

This operation is equivalent to the expression: 123 Mod 10 = 3

In this way, we obtain the last digit of the original number, which is 3, and it will be the first in the reversed number. Next, we obtain the following digit, which is 2, by dividing the quotient by 10, and so on.

```
12 | 10
 2 |  1
```

As we extract the digits from the original number, we arrange them from left to right to form the new number. To do this, it is necessary to multiply the current number being formed by 10 and add the last digit obtained, as follows:

3 * 10 + 2 = 32

Then we obtain the third and last digit, which is 1, and perform the operation:

```
1 | 10
1 |  0
```

32 * 10 + 1 = 321

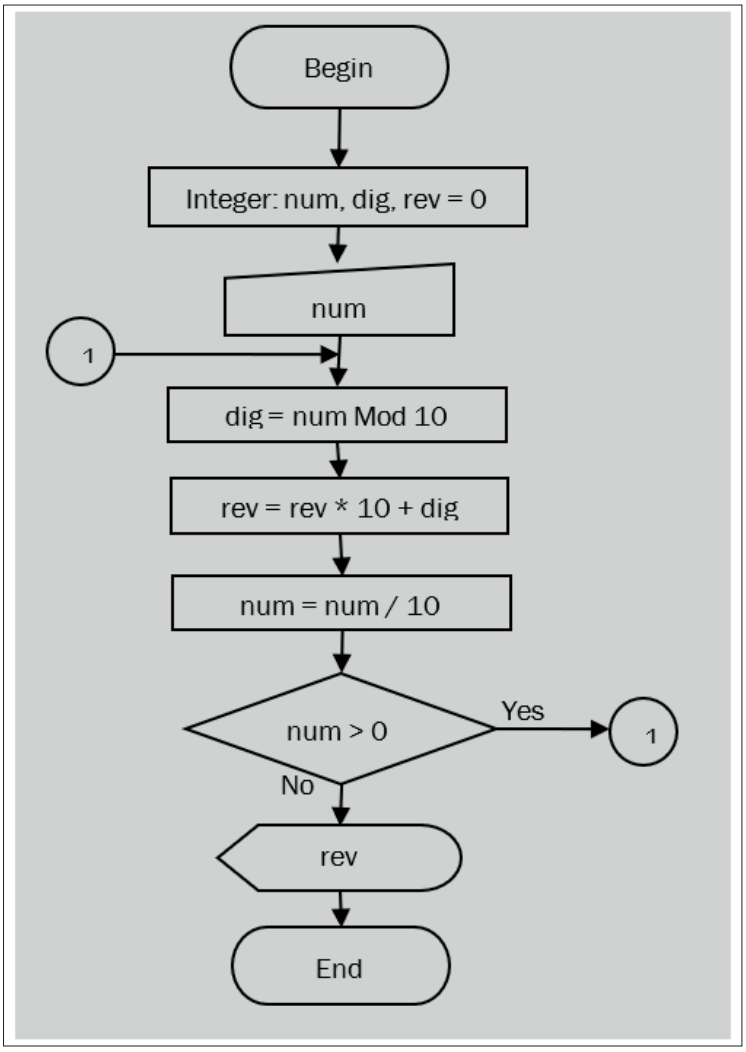
The operations are repeated until the quotient of the division is 0, at which point all digits will have been reversed. In this order of ideas, we have the following data:

Input data: number
Output data: reversed number

Processes:
Digit = number Mod 10
Number = number / 10
Reversed number = reversed number * 10 + digit

The flowchart is presented in Figure 78.

Figure 78. *Flowchart to Reverse the Digits of a Number*



In this algorithm, the iterative structure *do while* is used. Since there is no specific symbol for this structure in flowchart notation, it is modeled using a decision and a connector that controls the execution back to the first process that is repeated.

The data obtained from the step-by-step verification of the algorithm is shown in Table 71.

Table 71. *Verification of the Reverse Digits Algorithm*

Iteration	num	dig	Rev	Output
	123			
1	123	3	3	
2	12	2	32	
3	1	1	321	
	0			321

Example 39. Perfect Number

A number is considered perfect if the sum of its divisors, excluding itself, equals the number. We need an algorithm to determine if a number n is perfect.

To better understand the concept of a perfect number, let's consider two cases: 6 and 8. The divisors of 6 are: 1, 2, 3, and 6; while the divisors of 8 are: 1, 2, 4, and 8 (every number is a divisor of itself). If we sum only the divisors less than each number, we have:

$$\sum_{D_6=1}^{D_6<6} D_6 = 1 + 2 + 3 = 6 \qquad \sum_{D_8=1}^{D_8<8} D_8 = 1 + 2 + 4 = 7$$

Let Dn denote the divisor of n. Thus, we have: the sum of the divisors of 6, less than 6, is equal to 6, from which it is concluded that 6 is a perfect number; regarding the second number, the sum of the divisors of 8, less than 8, is equal to 7, consequently, 8 is not a perfect number.

Now, to determine if a number is perfect, we require three basic processes: identifying the divisors, summing them, and checking if the sum equals the number. To identify the divisors less than n, we need to iterate from 1 to n/2 and verify for each number whether it is a divisor of n; this is done using an iterative structure. To sum

them, an accumulator variable is updated within the loop; and finally, a decision structure located outside the loop will determine whether the number is perfect or not. The solution to this exercise is presented in Figure 79.

Figure 79. *N-S Diagram for Perfect Number*

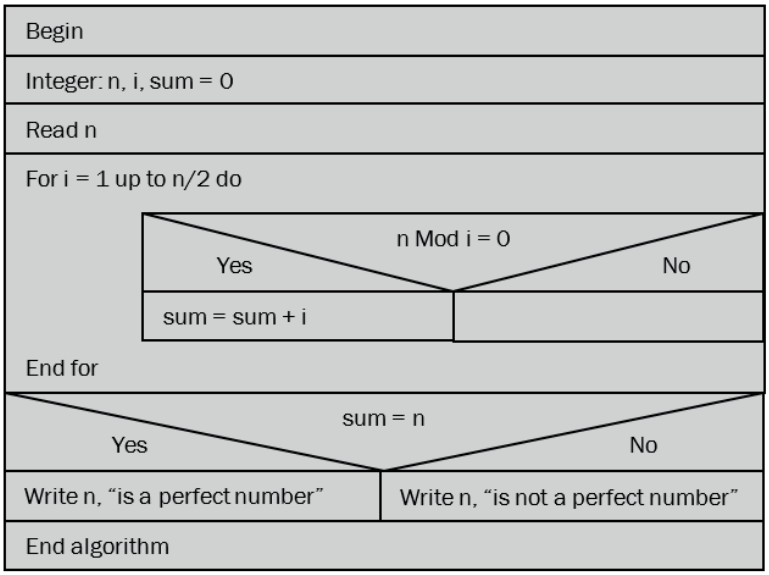


Table 72 shows the results of the verification of this algorithm with numbers 6 and 8.

Table 72. *Verification of the Perfect Number Algorithm*

Execution	n	i	sum	Output
1	6		0	
1		1	1	
1		2	3	
1		3	6	6 is a perfect number
2	8		0	
2		1	1	
2		2	3	
2		3	3	
		4	7	8 is not a perfect number

Example 40. Iterative Exponentiation

Given two integers: ***b*** (the base) and ***e*** (the exponent), we need to calculate the result of the exponentiation.
Exponentiation is a mathematical operation whose result is the product of multiplying the base by itself as many times as indicated by the exponent. Among its properties are: if the exponent is 0, the result is 1, and if the exponent is 1, the result is the same as the base. For this exercise, to simplify the solution, we limit the exponent to positive integers. For example:

$2^3 = 2 * 2 * 2 = 8$
$3^5 = 3 * 3 * 3 * 3 * 3 = 243$

To perform exponentiation, we need to implement an iterative structure and carry out successive multiplications of the base within it. The exponent indicates the number of iterations to perform. The data is as follows:

Input data: base, exponent
Output data: result
Process: product = product * base

Table 73 presents the pseudocode for this exercise.

Table 73. *Pseudocode for Iterative Exponentiation*

```
1    Begin
2        Integer: b, e, p = 1, with
3        Read b, e
4        For with = 1 to e do
5            p = p * b
6        End for
7        Write p
8    End algorithm
```

Any number multiplied by 0 results in 0; therefore, the variable ***p*** (product), which accumulates the results of multiplication, is initialized to 1. If it were initialized to 0, the final result would also be 0. The results of verifying this algorithm are presented in Table 74.

Table 74. *Verification of the Iterative Exponentiation Algorithm*

Execution	Iteration	b	e	With	P	Output
1					1	
1	1	2	4	1	2	
1	2			2	4	
1	3			3	8	
1	4			4	16	16
2		3	5		1	
2	1			1	3	
2	2			2	9	
2	3			3	27	
2	4			4	81	
	5			5	243	243

Example 41. Prime Number

A prime number is an integer that has only two divisors: one and itself. We need an algorithm that, given a number n, decides whether it is prime or not.

To determine if a number satisfies this property, we check if it has any divisors other than one and itself; if it does, it is not prime. Otherwise, it is considered prime. However, it is not necessary to check all numbers; to determine if a number is prime, it suffices to search for divisors between 2 and the square root of the number7.

Examples include numbers 9 and 19. The square root of 9 is 3, so we look for divisors between 2 and 3, finding that 3 is divisor, thus 9 is not a prime number. For 19, we have that the integer square root is 4, so we look for divisors between 2 and 4, finding that the numbers 2, 3, and 4 are not divisors of 19, concluding that 19 is a prime number. We could verify this for the numbers up to 18, but the result would be the same.

To determine whether a number is prime, we have the following data and operations:

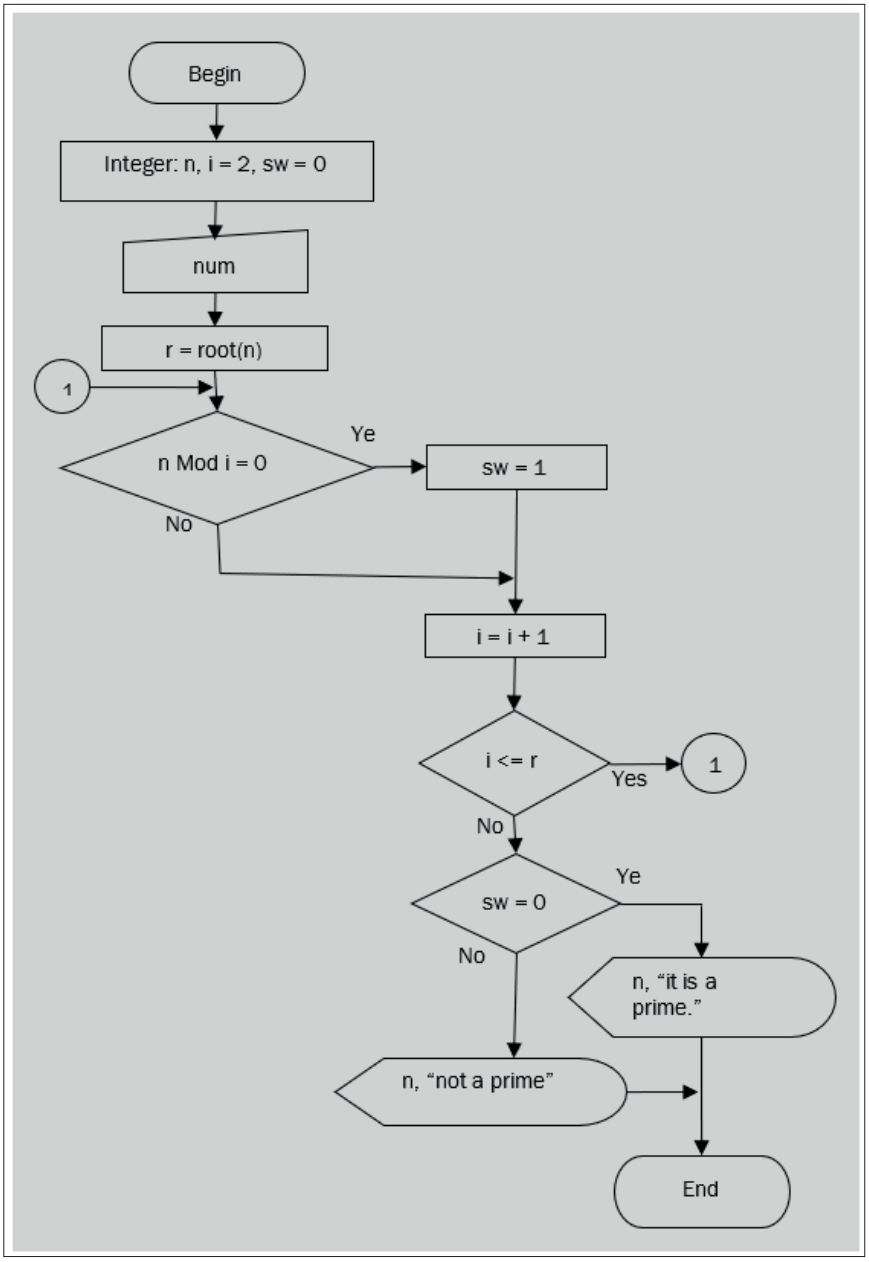
Input data: number

7 The purpose of this exercise is to apply an iterative structure in the solution; therefore, we proceed to look for divisors between all numbers between 2 and the integer square root of the number. There is another faster method to check if a number is prime, proposed by Eratosthenes (Greek mathematician of the 3rd century B.C.), who proposes that to check if a number is prime, one only needs to divide by 2, 3, 5, and 7 (Great Encyclopedia Espasa, 2005).

Output data: message “prime number” or “not a prime number”
Process: number Mod i where i: 2 ... root(number)

The algorithmic solution is presented in Figure 80 in flowchart notation.

Figure 80. *Flowchart for Prime Number*



Four variables have been declared in this algorithm: ***n, r, i,*** and ***sw***; n for the number, ***r*** to compute the square root of ***n, i*** to Iterate between 2 and ***r*** and ***sw***. The last variable is a switch or flag, which serves to indicate if any divisors of ***n*** were found during the iterations for ***i***. This variable is initialized to 0; if it remains 0 at the end of the cycle, it means no divisors were found and thus the number is prime. If the variable takes the value 1, it indicates that there is at least one divisor, and hence the number is not prime.

Table 75 shows the results of the step-by-step implementation for numbers 9 and 19.

Table 75. *Verification of the Prime Number Algorithm*

Execution	Iteration	n	r	i	sw	Output
1		9	3		0	
	1			2		
	2			3	1	9 is not prime
2		19	4		0	
	1			2		
	2			3		
	3			4		19 is prime

Example 42. Points on a Line

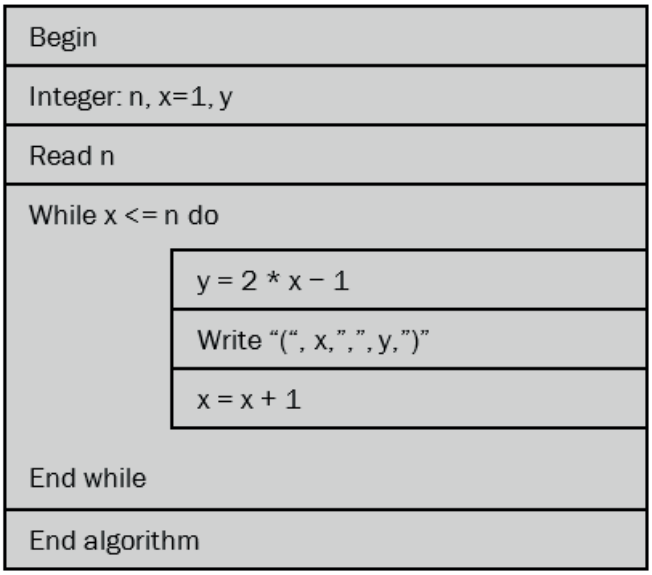
Given the linear equation y = 2x – 1, we need an algorithm to calculate n points through which the line passes from x = 1.

If we take n = 4, the points would be:

X = 1 →y = 2(1) – 1 = 1 →(1.1)
X = 2 →y = 2(2) – 1 = 3 →(2.3)
X = 3 →y = 2(3) – 1 = 5 →(3.5)
X = 4 →y = 2(4) – 1 = 7 →(4.7)

In this case, the input data corresponds to ***n***, the output data corresponds to the points (***x,y***), as many as indicated by ***n***, and the process is limited to developing the equation by replacing ***x*** with the corresponding value. An iterative structure is used, as shown in the N-S diagram in Figure 81.

Figure 81. *N-S Diagram for Points on a Line*



The data generated to verify the correctness of this algorithm is presented in Table 76.

Table 76. *Verification of the Points on a Line Algorithm*

Execution	N	x	y	Output
1	4	1	1	(1.1)
1		2	3	(2.3)
1		3	5	(3.5)
		4	7	(4.7)

Example 43. Square Root

Calculate the integer square root of a number.

It is known that the square root of a number is another number that, when squared, equals the first number.

$$\sqrt{4} = 2 \rightarrow 4 = 2^2$$

$$\sqrt{25} = 5 \rightarrow 25 = 5^2$$

Some numbers, like the previous ones, have an exact integer square root, while others result in a real number, like the square root of 10, and for negative numbers, the root is a complex or imaginary number.

$$\sqrt{10} = 3.16227766$$

The integer square root of a number, in cases where there is no exact root, is obtained by truncating the decimal part; for example, the integer square root of 10 is 3.

In this exercise, the input data is the number, the output data is the root, and the process is to calculate the squares of numbers from 1 up to the square root of the number. The integer root is found when the next number, when squared, gives a value greater than the input number. The solution to this exercise is presented in pseudocode in Table 77.

Table 77. *Pseudocode for the Square Root Algorithm*

```
1   Begin
2       Integer: n, i = 0, c
3       Read n
4       Do
5           i = i + 1
6           c = (i+1) * (i+1)
7       While (c <= n)
8       Write "The square root of", n, "is", i
9   End algorithm
```

Table 78 presents the results of the step-by-step execution of this algorithm.

Table 78. *Verification of the Square Root Algorithm*

Execution	N	I	C	Output
1	25	0		
1		1	4	
1		2	9	
1		3	16	
1		4	25	
1		5	36	The square root of 25 is 5

2	10	0		
2		1	4	
2		2	6	
2		3	16	The square root of 10 is 3

4.4 Proposed Exercises

Design algorithms to solve the problems proposed below. It is suggested to alternate between pseudocode notation, flowcharts, and N-S diagrams, as wells as the three iterative structures studied in this chapter.

1. A student has seven partial grades in the Introduction to Programming course, an algorithm is required to calculate the average of such grades.

2. Design an algorithm to read integers until 0 is entered. Calculate the square of negative numbers and the cube of positive numbers.

3. Design an algorithm to input numbers as many as the user wants. At the end of the cycle, report how many even and odd numbers were recorded, as well as the sum of the even and odd numbers.

4. The principal of Buena Nota School wants to know the average age of the students in each grade. The school offers education from preschool to fifth grade and has a group of students in each grade. Design an algorithm that reads the age and grade of each student in the school and generates the corresponding report.

5. A coach has proposed to an athlete to complete a five-kilometer route for 10 days, to determine if he/she is eligible for the five-kilometer test. To qualify, the athlete must meet the following conditions:

   - That in none of the tests takes more than 20 minutes.
   - That in at least one of the tests he/she takes less than 15 minutes.
   - That his/her average time is less than or equal to 18 minutes.

   Design an algorithm to record the data and decide if the athlete is eligible for the competition.

6. An insurance company employs n salespeople. Each salesperson receives a base salary and an additional 10% in commission on their sales. An algorithm

is required to calculate the amount to be paid to each employee, as well as the total amounts to be paid in terms of salaries and commissions.

7. Develop an algorithm to process the final Biology grades for a group of n students. The aim is to determine the group's average grade, classify the students into the following categories based on their grades: excellent, good, average, and deficient, and count how many students fall into each category. The grading scale is as follows:

   Grade >= 4.8: Excellent
   4.0 <= grade <= 4.7: Good
   3.0 <= grade <= 3.9: Average
   Grade <= 2.9: Deficient

8. Design an algorithm that generates the n-th number of the Fibonacci sequence.

9. A survey was conducted to n participants, requesting their opinions on the topic of compulsory military service for women. The response options were: in favor, against, and no response. An algorithm is required to calculate the percentage of respondents who selected for each option.

10. An algorithm is required that, through a menu, performs the functions of a calculator: addition, subtraction, multiplication, division, exponentiation, and percentage calculation. The menu will include a shutdown option to terminate the execution of the algorithm.

11. An algorithm is required to generate an invoice for a sale with n items. A 5% discount will be applied to quantities exceeding ten units of the same item.

12. Design an algorithm that reads an integer and sums the digits that compose it.

13. The Systems Engineering program requires an algorithm to determine the percentages of students who work and those who are exclusively dedicated to their studies, broken down by gender.
14. Buena Ropa store has monthly sales records and requires an algorithm to determine: in which month the highest sales occurred, in which month the lowest sales occurred, and the average monthly sales.

15. Design an algorithm to calculate the factorial of an n number.

16. An algorithm is required to find the prime numbers between 1 and n

17. Given a group of 20 students who took the Algorithms course, it is necessary to determine the group's average grade, the highest and lowest grades, how many students passed the course, and how many failed.

18. Given the equation $y = 2x^2+3x-4$ calculate the points through which the parabola passes in the interval -5, 5.

19. Design an algorithm to find the first n perfect numbers.

20. Design an algorithm to validate a grade. The grade must be a real value between 0 and 5.0. If a value outside this range is entered, it should be re-entered until a valid grade is provided.

21. Design an algorithm to validate a date in the format dd/mm/yyyy. A date is considered valid if it falls between 01/01/1900 and 31/12/2100, with the month between 1 and 12 and the day between 1 and 31, considering that some months have 28 (or 29), 30, or 31 days. If the date is not valid, an error message should be displayed, and the data should be re-entered again. The algorithm terminates when a valid date is entered.


Mathemitic Alghnimym Formas

5. ARRAYS

One must know oneself.
If this does not serve
to discover truth,
it at least serves
as a rule of life,
and there is nothing better.
Pascal.

When considering an algorithm as the design of a solution to a problem using a computer program, it is necessary to consider at least four aspects:

- Data Processing: The operations for transforming input information into output information.
- Data Storage: How data is stored and accessed, considering primary and secondary memory storage. Arrays are a way to manage data in main memory.
- System Architecture: This pertains to how the various components of the system are organized and interact with each other. Part of this topic will be addressed in the following chapter.
- User Interface: How the user interacts with the system, what elements are available, and how their requirements are met. This topic is not addressed directly in this book.

Arrays are structures that allow for the grouping of data to facilitate its management in computer memory and to perform operations that require access to sets of data simultaneously, such as searching and sorting. To better understand this concept, consider the following cases:

First case: An algorithm is required to calculate the final grade of a student based on three partial grades. In this situation, as in all the examples presented in the previous chapters, it is sufficient to declare three variables, one for each grade.

Second case: An algorithm is required to calculate the final grade of a group of 40 students based on three partial grades and to present the list in alphabetical order. Arguably, 160 variables are declared for the grades and 40 for the names; however, this is practically unmanageable. For cases like this, it is necessary to group data of

the same type under a common identifier, so that an array of names and an array of grades can be managed.

5.1 CONCEPT OF AN ARRAY

An array is a collection of elements of the same type[8] stored in consecutive memory addresses, which are referenced by a common identifier or name. The elements are distinguished from one another by an index that represents the position of each element within the structure.

In this regard, Brassard and Bratley (1997: 167) consider the term array as a synonym of array, which they conceptualize as a data structure with a fixed number of elements of the same type stored in contiguous positions in the computer's memory. Thus, knowing the size of the elements and the address of the first one allows for easy calculation of the position of any item when necessary, considering that they are organized from left to right.

An array is also defined as a composite data type composed of a collection of simple or composite data. This implies that an element of an array can be: a number, a character, a string, an array, or a record.

The use of arrays is essential when it is required to store and operate many data items. For example: a group of students, a payroll, an invoice (which includes several products), an inventory, or a library; all these concepts refer to sets of data of the same type, on which operations such as searching, sorting, and summing can be performed.

Among the characteristics of arrays are:

- They are static structures; that is, once defined they cannot be resized.
- They store homogeneous data, except in untyped languages.
- Data is stored in memory occupying consecutive positions, so that only the reference to the first element is necessary.
- They share a common identifier (variable name) that represents all the elements.

8 The definition of array as a collection of data of the same type (homogeneous) is widely adopted in programming languages with type definitions, where it is necessary to declare variables and arrays before using them. However, in untyped languages, arrays can group elements of different types.

- Individual elements are identified by the index or position they occupy within the array.

5.2 TYPES OF ARRAYS

Arrays are classified according to their contents; consequently, there are numerical arrays, character arrays, string arrays, record arrays, and arrays of arrays.

When an array is composed of simple data, such as numbers or characters, it is referred to as a linear array or one-dimensional (1D) array, and they are known as vectors; these have only one index. When the elements of the array are also arrays, it is referred to as a multidimensional array, which has an index for each dimension and is called an array. Figures 82, 83, and 84 illustrate the graphical representations of one-dimensional, two-dimensional, and three-dimensional arrays.

Figure 82. *Graphical Representation of a Vector*

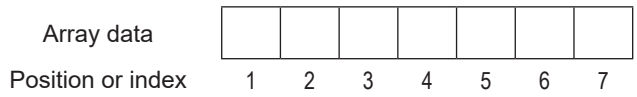


Figure 83. *Graphical Representation of a Two-Dimensional Array*

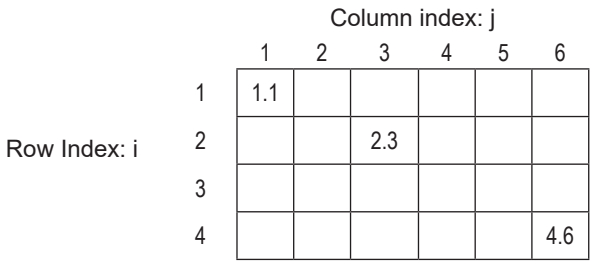
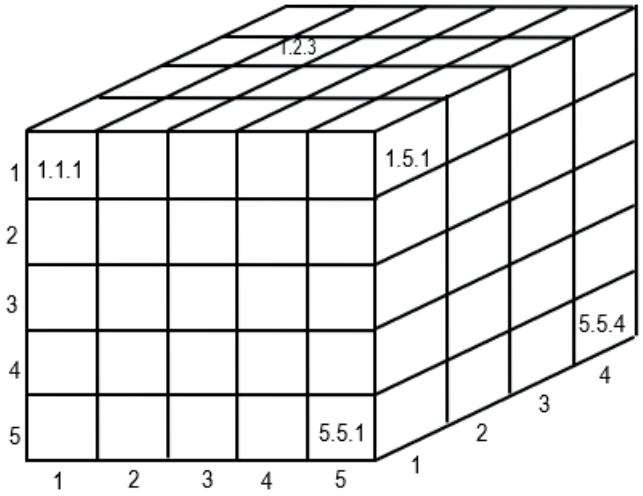


Figure 84. *Graphical Representation of a Three-Dimensional Array*



As can be seen in the graphs, a vector is a list whose elements are identified by a single index, while a two-dimensional array requires two indexes: one for rows and another for columns. A three-dimensional array requires three indexes, as its structure is analogous to a three-dimensional object, similar to a box with length, width, and height divided into equally three-dimensional compartments. Theoretically, an array of two, three, four, or more dimensions can be defined; however, in practice, it is common to use vectors and two-dimensional matrices, but not three-dimensional or higher. The remainder of this chapter focuses on the handling of vectors and two-dimensional matrices.

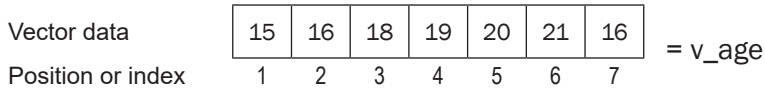
5.3 HANDLING VECTORS

A vector is a static data structure corresponding to a collection of data of the same type that are grouped under the same name and distinguished from one another by the position they occupy within the structure.

According to Timarán et al. (2009), a vector is a finite and ordered collection of elements. It is finite because every vector has a limited number of elements, and it is ordered because it is possible to determine which is the first, second, and n-th element.

For example, if seven students' ages are collected, they can be stored in a vector structure called v_age consisting of seven elements. Graphically, this vector would appear as shown in Figure 85.

Figure 85. *Graphical Representation of an Age Vector*



5.3.1 Declaration

Declaring a vector should be understood as the process of reserving space in memory to store a specified number of data elements, which will be distinguished by an identifier and an index, as previously described.

The simplest and most functional way to declare a vector is as follows:

Data_type: identifier[size]

Examples:

Integer: v_age[7]

A vector of seven integer elements type is being declared, as shown in Figure 85. This means is reserved in memory to store seven integer data elements in consecutive positions, which will be recognized by the identifier ***v_age.***

String: v_name[30]

This declaration reserves memory to store 30 names, which will be recognized by the identifier: ***v_name***.

5.3.2 Accessing Elements

To access an element of a vector and perform operations such as assignment, reading, printing, and forming expression, it is necessary to write the identifier (name) of the array and, in brackets, the position being referenced, in the following way:

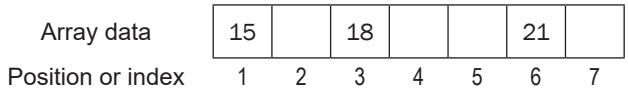
vector[pos]

Example of declaration and access to the elements of a vector:

Integer: v_age[7]
v_age[1] = 15
v_age[3] = 18
v_age[6] = 21

A seven-position vector is declared, and data is assigned to positions 1, 3, and 6. The graphical representation is shown in Figure 86.

Figure 86. *Vector with Data*



To read a number and store it in the fourth position, it should be written:

Read v_age[4]

To print the data stored in the sixth position, it should be written:

Print v_age[6]

Similarly, expressions can be constructed using the elements of the vector. For example, if you want to know the difference between the ages stored at positions 1 and 3, you should write:

Integer: dif
dif = v_age[1] – v_age[3]

In summary, when the vector identifier is written along with the position of an element, it behaves as if it were a variable on which different operations can be performed.

5.3.3 Traversing a Vector

Many operations applied to vectors involve all elements. In these cases, it is necessary to access consecutively from the first to the last position, a process known as traversing a vector.

To traverse a vector, an iterative structure is used, controlled by a variable that starts at 1 to reference the first position and increments by one until reaching the size of the array, as follows:

```
Data_Type: vector[n][9]
int: i
for i = 1 to n do
    operation on vector[i]
end for
```

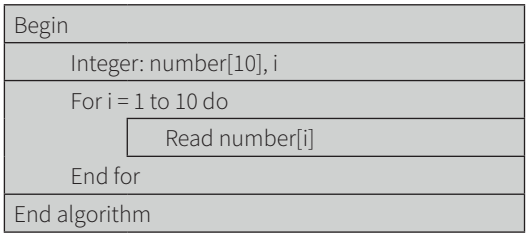
The variable i represents the position of each element; thus, any operation defined on vector[i] inside the loop will be executed on each of the vector's elements.

Example 44. Storing Numbers in a Vector

Design an algorithm that reads 10 numbers and stores them in a vector.

To solve this exercise, it is necessary to use the operations that have been explained in the previous pages: declaration, traversal, and access to the elements of a vector. The N-S diagram of this algorithm is presented in Figure 87.

Figure 87. *N-S Diagram for Storing Numbers in a Vector*



In this algorithm, the use of an iterative structure is observed to traverse the vector and read a value for each position, using the loop control variable (i) as the index to identify the elements of the array.

Example 45: Input and Output of Data from a Vector

Design an algorithm to read 10 numbers, store them in a vector, and then display them in reverse order from how they were entered.

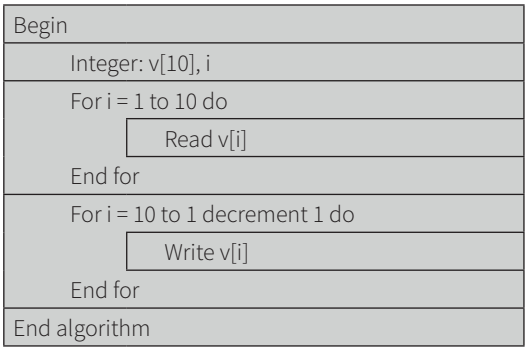
9 n is the vector size.

One of the advantages of using arrays is that they allow to store sets of data in memory and easily access them. The data in memory can be listed in any order, depending on the direction in which the vector is traversed.

Traversal is done using the loop control variable as the index of the vector. If the variable starts at 1 and is incremented, it will traverse forward; whereas if it starts at 10 and is decremented, it will traverse backward.

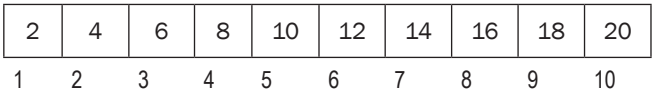
In this algorithm, a vector is declared, and two loops are used: the first traverses from position 1 to 10 to input data for each position, while the second traverses from the last position to the first, printing the data from each position. The N-S diagram of this algorithm is presented in Figure 88.

Figure 88. *N-S Diagram for Input and Output Data from a Vector*



If the following data are entered when verifying the algorithm: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, the vector would look like in Figure 89:

Figure 89. *Graphical Representation of a Vector*



When traversing the vector and displaying the data starting from the tenth position, the data would be in the following order: 20, 18, 16, 14, 12, 10, 8, 6, 4, 2.

Example 46: Calculate the Average of Numbers in a Vector

Read 10 integers and store them in a vector. Calculate and display the average, then generate a list with numbers smaller than average and another list with the numbers greater than the average.

Reading numbers and calculating the average can be done using only loops, but when it is required to display the numbers less and greater than the average separately, it is clear that the numbers must have been stored to access and compare them with the average. Here, the need for a vector becomes evident; otherwise, it would be necessary to declare 10 variables and implement 20 decisions to compare the numbers. This would be bad programming practice, especially considering that just as there are 10 numbers, there could be 100.

To better understand the exercise, consider the vector shown in Figure 90.

Figure 90. *Graphical Representation of the Numeric Vector*

2	44	16	8	10	32	24	160	13	20
1	2	3	4	5	6	7	8	9	10

Once the data is stored in the vector, it is necessary to calculate the average, which requires determining the total sum. In this case, the sum is:

Sum = 329
Average = 32.9

To create a list of numbers greater than the average, each element in the vector must be compared to the value obtained as the average:

Numbers greater than the average: 44, 160

In the same way, the second list is obtained:

Numbers less than the average: 2, 16, 8, 10, 32, 24, 13, 20

Now, we proceed to the design of the algorithm based on this information:

Input data: number (10 numbers are read inside a loop and stored in a vector)

Storage structure: 10-element vector

Output data: average, numbers less than the average, and numbers greater than the average

Process: sum = sum + number (for each position in the vector)
average = sum / 10

In this algorithm, three loops are implemented, one to read the data and store it in each position of the vector, and two more to generate the two lists independently.

The algorithm is presented in Table 79 in pseudocode notation.

Table 79. *Verification of the Square Root Algorithm*

```
1     Begin
2         Integer: v[10], i, sum = 0
3         Real: average
4         For i = 1 to 10 do
5             Read v[i]
6             sum = sum + v[i]
7         End for
8         average = sum / 10
9         Write “Average:”, average
10        Write “Greater than the average”
11        For i = 1 to 10 do
12            If v[i] > average then
13                Type v[i]
14            End if
15        End for
16        Write “Less than the average”
17        For i = 1 to 10 do
18            If v[i] < average then
19                Type v[i]
20            End if
21        End for
22    End algorithm
```

5.4 HANDLING MATRICES

An array is a set of elements arranged along ***m*** rows and ***n*** columns, also known as a two-dimensional array or a complete table (Gran Enciclopedia Espasa, 2005: 7594).

In programming, an array is defined as a static data structure that, under a single identifier, stores a collection of data of the same type. It is a two-dimensional array organized in the form of rows and columns, and therefore, uses two indexes to identify the elements. Graphically, an array has the form of a two-dimensional table, as shown in Figure 91.

Figure 91. *Graphical Representation of an Array*

		Column index: j					
		1	2	3	4	5	6
Row Index: i	1	1.1	1.2	1.3	1.4	1.5	1.6
	2	2.1	2.2	2.3	2.4	2.5	2.6
	3	3.1	3.2	3.3	3.4	3.5	3.6
	4	4.1	4.2	4.3	4.4	4.5	4.6

5.4.1 Declaration

To declare an array, the syntax is the same as for vectors, with the difference that two sizes are used to indicate the number of rows and columns. The general form to declare an array is:

DataType: identifier[m][n]

Where ***m*** is the number of rows and ***n*** is the number of columns. The total number of available positions in the array is the product of the number of rows times the number of columns: ***m * n***.
Examples:

Integer: mat[4][6]
Character: letters[10][12]

The first statement declares the four-row, six-column ***mat*** array, as shown in Figure 89, capable of holding 24 integers. In the second, the 10-row, 12-column letter array, which reserves space to store 120 characters.

5.4.2 Access to Elements

To refer to an element in an array, you must specify the array identifier or name, and in brackets, indicate the row index and column index, as follows:

Identifier [i][j]

Where i refers to the row and j to the column. Writing the identifier and the two indexes refers to a specific element; therefore, it can be used as a simple variable to which data is assigned and from which it is taken to form expressions or send data to an output device.

Example:

Integer: m[3][4]
m[1][1] = 2
m[2][2] = 5
m[3][4] = 15

The first statement declares the array ***m***, the second assigns the number 2 to the first position of ***m***, and the third assigns the number 5 to the element located at the intersection of row 2 and column 2. Similarly, the fourth store number 15 in position 3.4. Graphically, the result of these expressions would be like Figure 92.

Figure 92. *Graphical Representation of an Array with Data*

	1	2	3	4
1	2			
2		5		
3				15

5.4.3 Traversing of an Array

Some operations to each element of the array are applied, such as reading and storing data, printing all data, searching for an element, or summing a numeric array. In these cases, the act of consecutively visiting all the elements of the structure is referred to as traversal. Two nested loops are required to traverse an array: one to traverse the rows and another to traverse the columns.

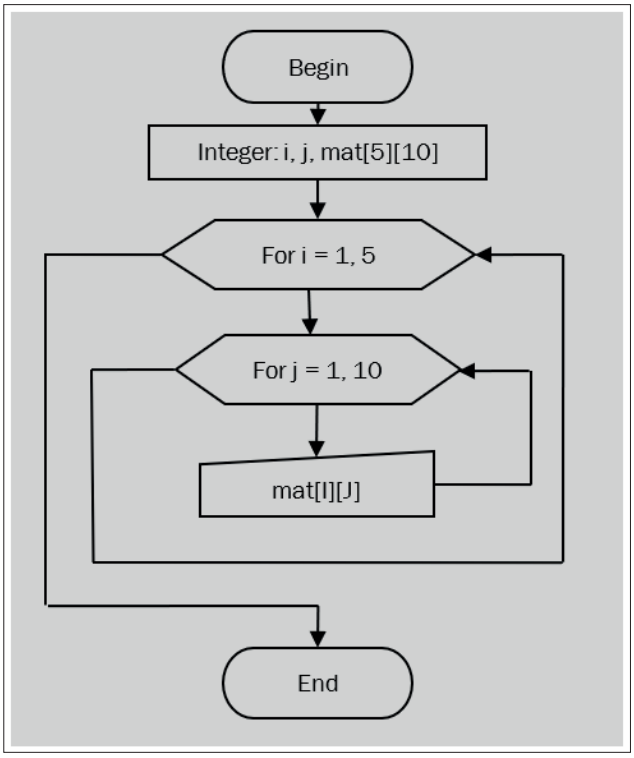
In general terms, to traverse an array of size ***m * n***, the loops are defined as follows:

For i = 1 to m do
For j = 1 to n do
Operation on the data
End for
End for

Example 47. Filling in an Array

To put into practice what has been studied regarding the declaration, access, and traversal of an array, this example declares an array of 5 * 10, traverses it position by position, reads, and stores a number in each of them. The flowchart of this algorithm is presented in Figure 93.

Figure 93. *Flowchart for Filling an Array*



In this example, an array of 5 rows and 10 columns is declared for a total of 50 positions. A loop with the variable ***i*** is used to traverse all the rows, and while in each of

them, a loop with the variable *j* is used to traverse each of the columns; that is, each of the positions in the i-th row and in each position, a value read from the keyboard is stored.

Example 48. Printing the Content of an Array

Declare a 4*6 array, fill each position with the sum of its indexes, and then display its content. For this, it is necessary to implement two independent traversals, one for each operation, the first to fill the array and the second to show its content. Once the first traversal is completed, the array will be like the one shown in Figure 94. The N-S diagram of this algorithm is presented in Figure 95.

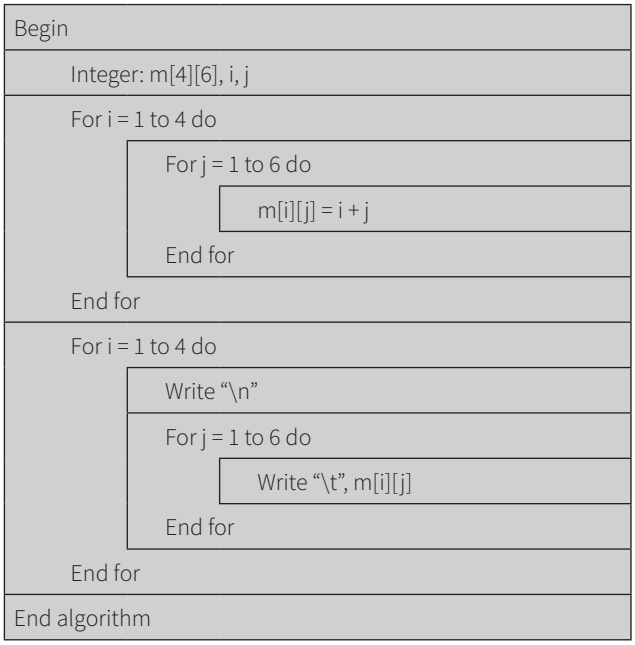
Figure 94. *4*6 Array*

m[i][j]	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10

This algorithm implements two array traversals using four nested loops in pairs. In the first traversal, the variables *i* + *j* are summed, and the result is stored in the corresponding position of the array. In the second traversal, at each row change, "\n" is printed; this is a common code in programming that means a line break, i.e., to move to the next line. At each column change, the "\t" code is printed, which is also common in programming and means to insert a tabulation, used in this case to separate the numbers from each other, and then the content of the cell is printed. Thus, when executing this algorithm, an output such as the following is obtained:

```
2   3   4   5   6   7
3   4   5   6   7   8
4   5   6   7   8   9
5   6   7   8   9   10
```

Figure 95. *N-S Diagram to Printing the Contents of an Array*



5.5 MORE EXAMPLES WITH ARRAYS

Example 49. Finding the Largest and Smallest Values in a Vector

Design an algorithm to store 12 numbers in a vector and then identify the smallest number, the largest number, and the position of each within the vector.

The first part of this exercise, concerning the declaration of the vector and storing the numbers, is performed similarly to previous exercises: by using an iterative structure. The second part: identifying the largest and smallest numbers, requires a solution strategy.

The strategy consists of declaring two variables: one to store the position of the largest number found and another for the position of the smallest number. These two variables are initialized at 1 to begin comparisons from the first element in the vector.

The following data is considered for the design of the solution:

Input data: number (read 12 times)

Output data: largest value, position of the largest, smallest value, and position of the smallest

Storage structure: a 12-position vector

Process: comparison and assignment

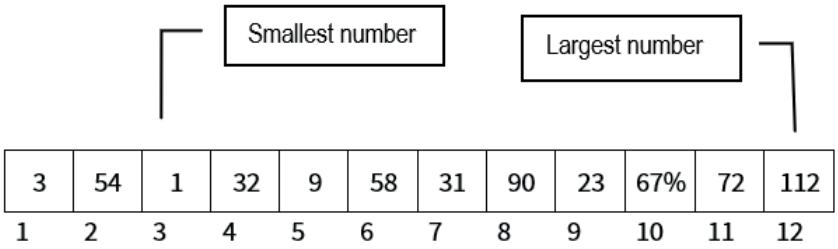
The pseudocode for this algorithm is presented in Table 80.

When verifying the algorithm and entering the data: 3, 54, 1, 32, 9, 58, 31, 90, 23, 67, 72, and 112, the vector shown in Figure 96 is obtained.

Table 80. *Pseudocode to Find the Largest and Smallest Values in a Vector*

```
1   Begin
2       Integer: v[12], i, pmay = 1, pmen = 1
3       For i = 1 to 12 do
4           Read v[i]
5       End for
6       For i = 1 to 12 do
7           If v[i] > v[pmay] then
8               pmay = i
9           Else
10              If v[i] < v[pmen] then
11                  PMEN = I
12              End if
13          End if
14      End for
15      Write “Largest number:”, v[pmay], “at position:”,pmay
16      Write “Smallest number:”, v[pmen], “in position:”,pmen
17  End algorithm
```

Figure 96. *Vector with Values from Example 49*



The output obtained by running the algorithm with the mentioned data is:

Largest number: 112 at position: 12
Smallest number: 1 at position: 3

Example 50. Recurrence of a Data Point in an Array

There is a list of 25 students and their municipality of origin. An algorithm is required to store the list in an array and query students coming from a specific municipality.

This algorithm is structured in two basic operations: the first consists of storing the data in a array of 25 rows and two columns, for which it performs a traversal reading the data for each position; and locating the students from the requested municipality. This involves reading the data to be searched and then performing a traversal in which the reference data is compared with the data stored in the second column of the array, if they match, a counter is incremented, and the content of the first column is displayed. Systematizing the information from the exercise, we have:

Input data: student names, municipality of origin, and queried municipality
Storage structure: 25-row and 2-column array
Output data: student names and counter
Process: compare and count

The solution for this exercise is presented in Figure 97.

Example 51. Difference of Vectors

Design an algorithm that stores data in two vectors and subsequently identifies and displays the elements present in the first vector but not in the second.

In this case, the first step is to declare two vectors and fill them with data. Then, a traversal is performed on the first vector, and for each of its elements, and another traversal is conducted on the second vector to find matches for that element. This method verifies whether the data exists in both arrays, in which case a flag is raised. At the beginning of each inner loop, the flag is initialized, and if, at the end of the traversal over the second vector, the flag remains unset, the element from the first vector is displayed. Figure 98 graphically illustrates the comparisons made for the first element.

Figure 97. *N-S Diagram for Recurrence of Data in an Array*

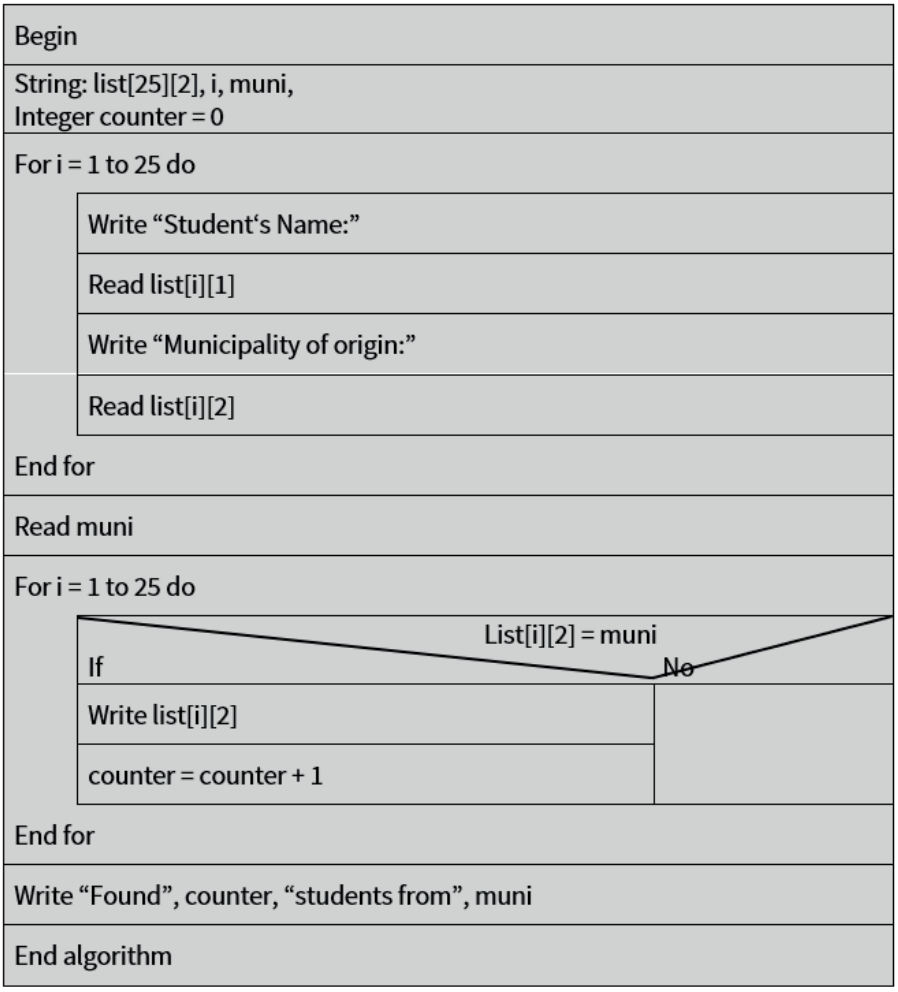
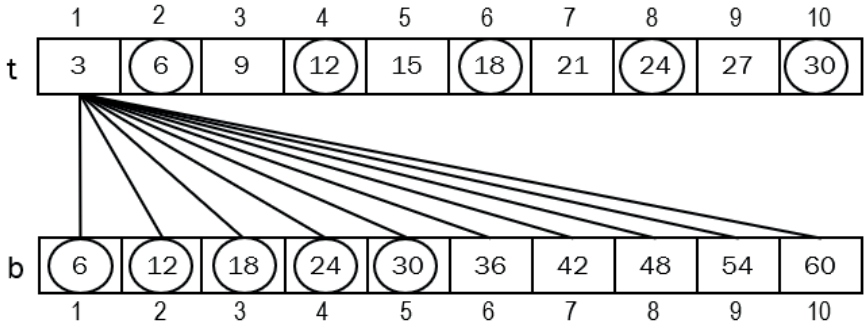
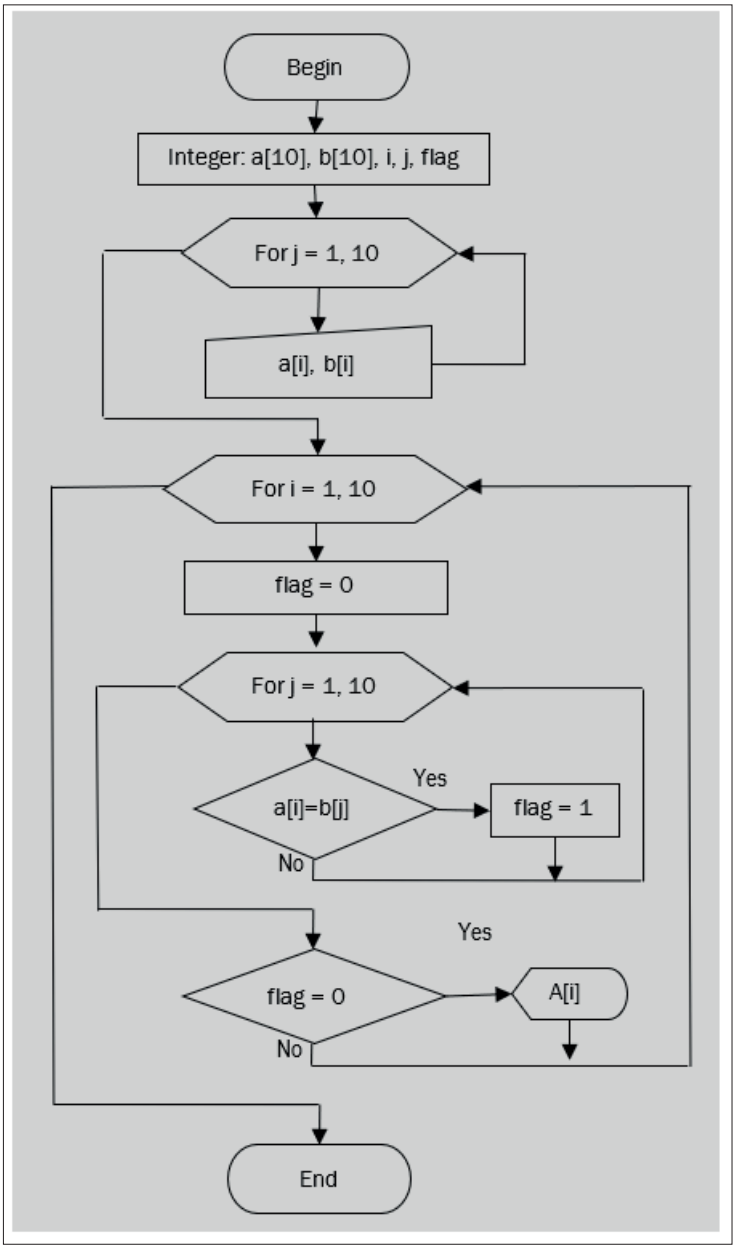


Figure 98. *Difference of Vectors*



In this example, the first vector contains the first 10 multiples of 3, and the second vector contains the first 10 multiples of 6. The lines illustrate the comparisons made from the first element of vector 1 and all the elements of vector 2. The same process will be repeated for the second element and then for the third until the traversal of the first vector is complete. The circles indicate the elements identified during the traversals as common to both vectors. Consequently, the elements of the first array that are not circled correspond to the difference: 3, 9, 15, 21, and 27. That is, multiples of 3 that are not multiples of 6. The flowchart for the solution to this exercise is presented in Figure 99.

Figure 99. *Flowchart for the Vector Difference Algorithm*

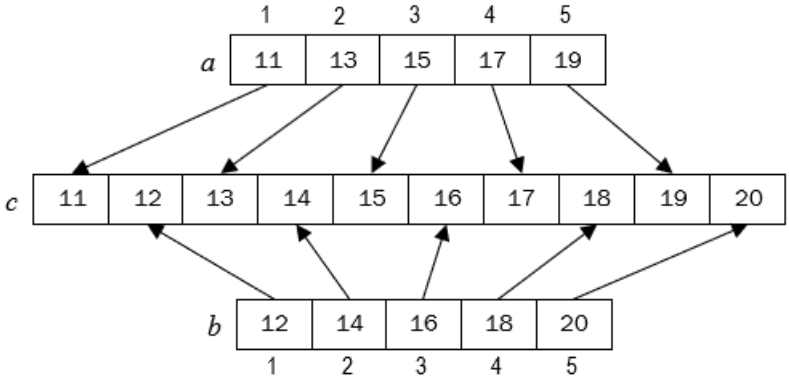


Example 52. Interleaving Vectors

Given two vectors ***a*** and ***b*** of ***n*** elements, an algorithm is required to generate a third vector (***c***) with the data from the first two interleaved. The new vector has the size of ***2n*** positions.

As an example, consider two vectors (***a*** and ***b***) with five elements each, and interleave their elements to create vector ***c*** with 10 elements, as shown in Figure 100.

Figure 100. *Interleaving Vectors*



In the solution of this exercise, three arrays are declared. The first two are filled with data recorded by the user, and the third is the result of interleaving the first two. The reading and storage of data for the two vectors are performed within the same loop, and the interleaving is done using a second loop. The algorithm is shown in Table 81.

Table 81. *Pseudocode for Interleaving Vectors*

```
1     Begin
2         Integer a[5], b[5], c[10], i=1, x=1
3         While i <= 5 do
4             Read a[i], b[i]
5             i = i + 1
6         End while
7         While i <= 5 do
8             c[x] = a[i]
9             x = x + 1
10            c[x] = b[i]
```

```
11              x = x + 1
11              i = i + 1
12          End while
13      End algorithm
```

Example 53. Summation of Rows and Columns of an Array

Given an m*n array, design an algorithm to sum each of the rows and store the results in a vector called sumrow, sum each of the columns and store the results in the vector sumcol, and finally display the two vectors.

Consider an array of four rows by six columns; this implies that the sumrow vector should have four positions and the sumcol vector should have six positions, as shown in Figure 101.

In this example, it is assumed that the array is filled with consecutive numbers from 1 to 24. In the algorithm presented in Figure 102, the data is input by the user.

Figure 101. *Summation of Rows and Columns of an Array*

	1	2	3	4	5	6	
1	1	2	3	4	5	6	21
2	7	8	9	10	11	12	57
3	13	14	15	16	17	18	93
4	19	20	21	22	23	24	129
	40	44	48	52	56	60%	

Figure 102. N-S *Diagram for Summation of Rows and Columns of an Array*

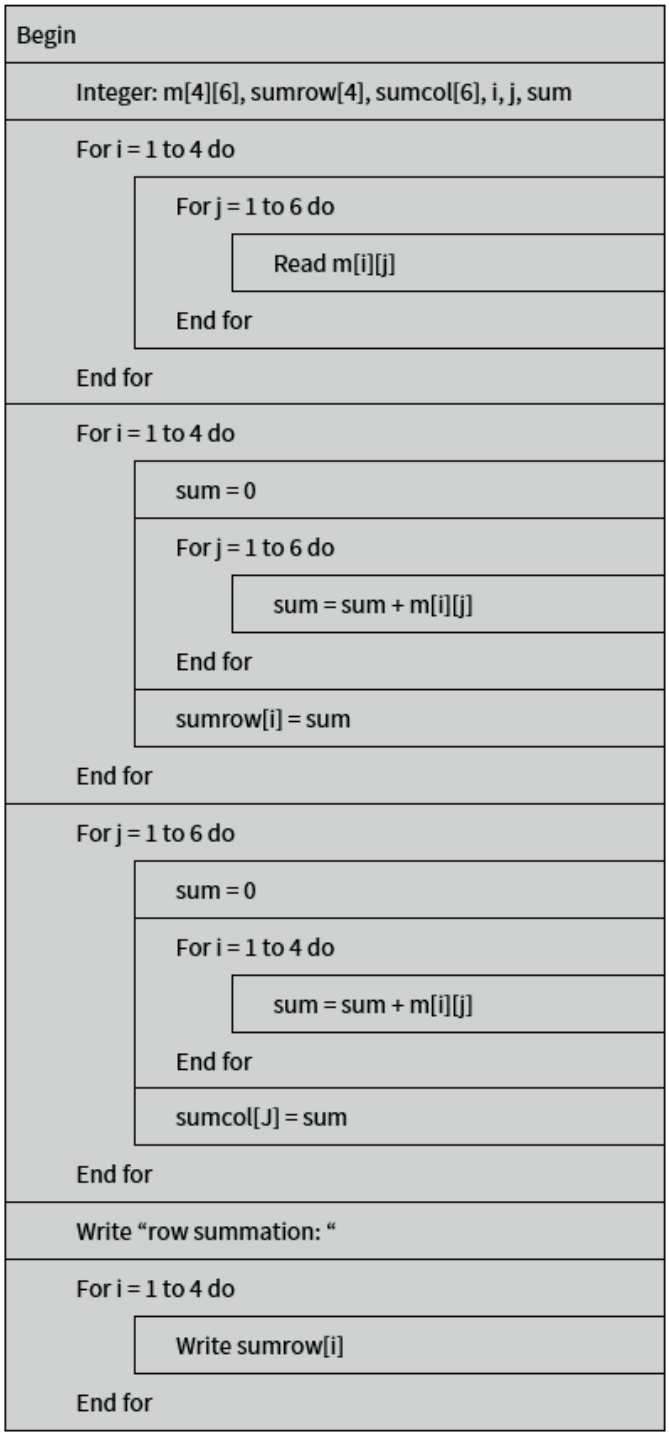
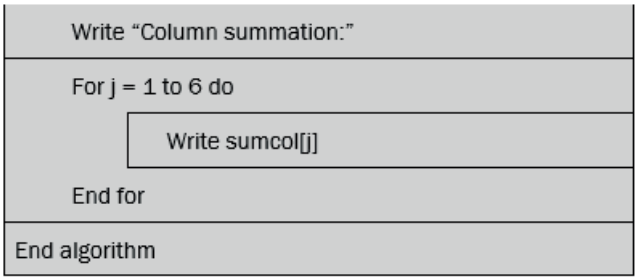


Figure 102. *(Continuation)*



To sum the rows, two nested loops are required. The outer loop controls the row index, and the inner loop controls the column index. The accumulator is initialized before starting the execution of the inner loop, and its value is stored at the end of this loop. To sum up the columns, a similar approach is taken but considering that the outer loop corresponds to the column index and the inner loop to the row index.

When executing the algorithm and entering the numbers from 1 to 24, the results would be:

Row summation: 21 57 93 129
Column summation: 40 44 48 52 56 60

Example 54. Principal Diagonal of an Array

Design an algorithm to display and sum up the principal diagonal of a square array.

Before designing the algorithm, Figure 103 graphically identifies the principal diagonal in a 5 * 5 array.

Figure 103. *Main Diagonal of an Array*

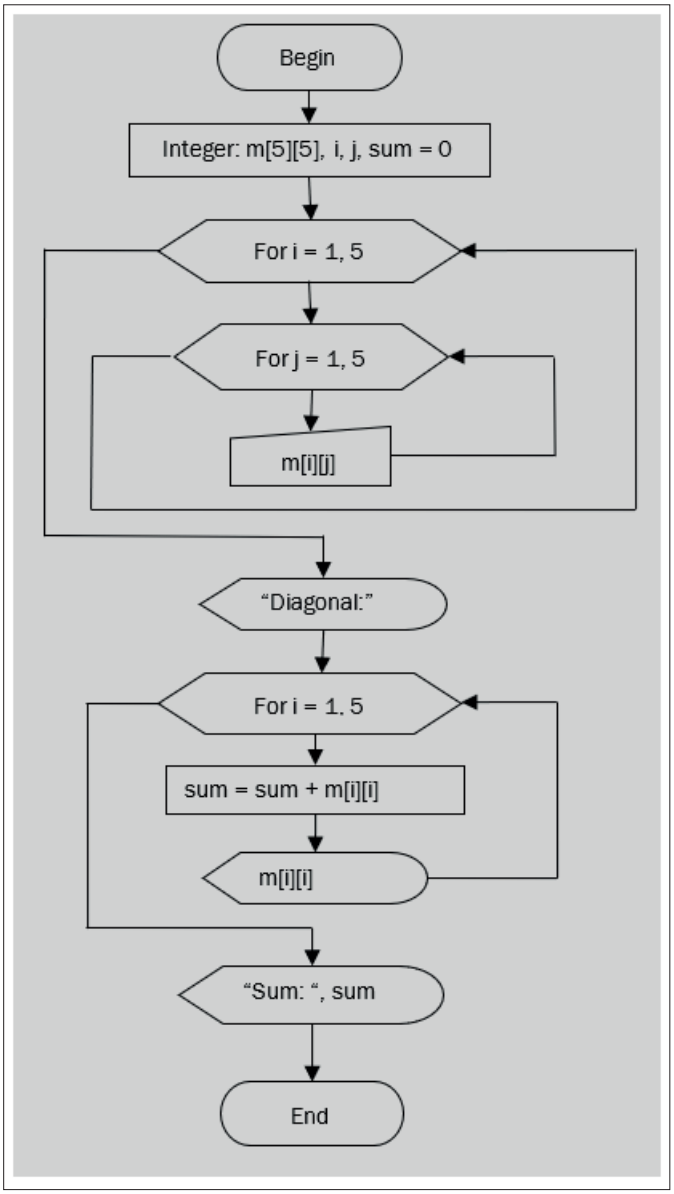
	1	2	3	4	5
1	**1**	2	3	4	5
2	6	**7**	8	9	10
3	11	12	**13**	14	15
4	16	17	18	**19**	20
5	21	22	23	24	**25**

Observing the graph, it can be seen that the elements of the principal diagonal have the same indexes in both rows and columns: *m[1][1], m[2][2], m[3][3], m[4][4] and*

m[5][5]. From this, it is concluded that a single loop is sufficient to traverse the diagonal; the loop control variable can be used as both the row and column index.

In the flowchart of Figure 104, two nested loops are implemented to read numbers and store them in the array. Then, a loop is implemented to traverse the diagonal. In each iteration, the element is displayed, and it is added to the cumulative sum.

Figure 104. *Flowchart for the Vector Difference Algorithm*



Example 55. Processing Grades Using Arrays

An algorithm is required to process the grades of a group of 40 students. Each student has three partial grades, and it is desired to know the final grade, which is obtained by average. After processing the grades for all students, the average for the group should also be calculated.

To solve this exercise, a vector and an array will be utilized. The vector will store the names of the students, while the array will store the partial grades and the final grades. Consequently, the vector will have 40 positions, and the array will consist of 40 rows and four columns. Figure 105 displays the model of the data structure for six students.

Figure 105. *Arrays for Processing Grades*

Vector for names

1	Carlos Salazar
2	Carmen Bolaños
3	Margarita Bermúdez
4	Sebastián Paz
5	Juan Carlos Díaz
6	Verónica Marroquín

Array for grades

	1	2	3	4
1	3.5	2.5	4.0	3.3
2	4.2	3.8	4.5	4.2
3	2.0	4.5	4.8	3.8
4	2.5	3.5	4.4	3.5
5	3.4	3.5	3.8	3.6
6	4.0	4.6	3.3	4.0
	Partial grade 1	Partial grade 2	Partial grade 3	Final grade

Although two data structures are used: a vector and an array, the data can maintain its integrity through the vector's index and the row index. In the array, the columns correspond to the grades, with the first three columns for the partial grades and the last column for the final grade. In this context, the data is read as follows: Carlos Salazar has grades of 3.5, 2.5, and 4.0 for his partial grades, and his final grade is 3.3. The same applies to the other rows.

Since each row in the vector and array refers to the same student, iterations occur student by student; therefore, only one loop is used. Within this loop, the name and partial grades are read, and the final grade is calculated, displayed, and accumulated to subsequently compute the course average. The algorithm is presented in pseudocode notation in Table 82.

Table 82. *Pseudocode for Processing Grades Using Arrays*

```
1   Begin
        String: names[40]
2       Real: grades[40][4], sum = 0, average
        Integer: count
3       For count = 1 to 40 do
4           Write “Name: \t”
5           Read names[count]
6           Write “Partial grade 1: \t “
7           Read grades[with][1]
8           Write “Partial grade 2: \t “
9           Read grades[with][2]
10          Write “Partial grade 3: \t “
11          Read grades[count][3]
12          grades[count][4] = (grades[count][1] + grades[count][2] + grades[count]
    [3])/3
13          Write “Final grade: \t”, grades[count][4]
14          sum = sum + grades[count][4]
15      End for
16      average = sum / 40
17      Write “Course average: \t“, average
18  End algorithm
```

Verifying the algorithm and processing the notes that were used in the model of Figure 105 has as output:

```
    Name: Carlos Salazar
    Partial grade 1:        3.5
    Partial grade 2:        2.5
    Partial grade 3:        4.0
    Final grade:            3.3

    Name: Carmen Bolaños
    Partial grade 1:        4.2
    Partial grade 2:        3.8
    Partial grade 3:        4.5
    Final grade:            4.2
…                           …
Group average:  3.7
```

5.6 PROPOSED EXERCISES

Solve the following exercises

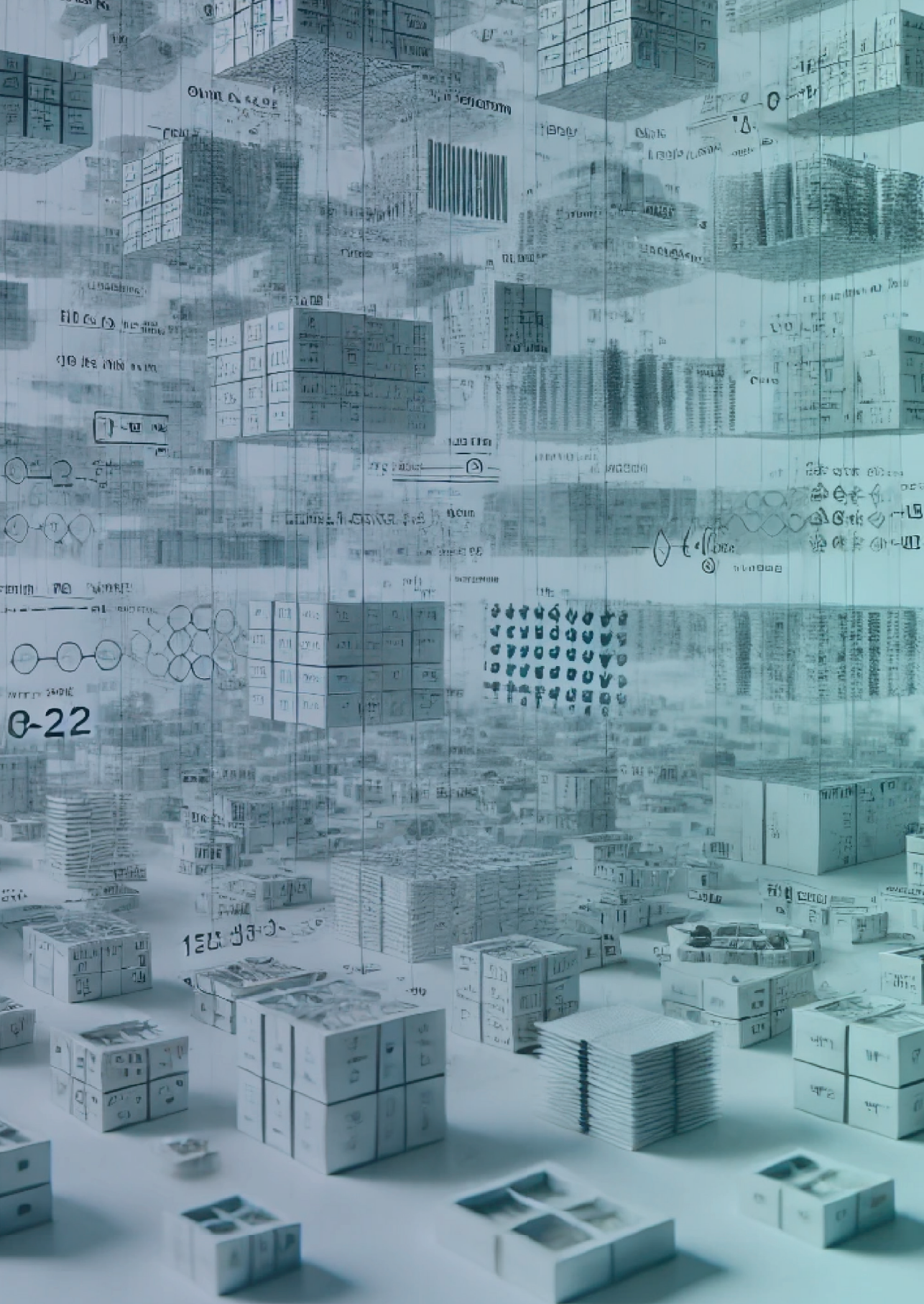
1. Given two numeric vectors, design an algorithm that identifies and displays the numbers they have in common.
2. Design an algorithm to add two numeric vectors
3. Design an algorithm to transpose a square array
4. Given two matrices: *A[m][n] and B[n][m]*, design an algorithm to check if the array B is the transpose of *A*.
5. Design an algorithm to calculate the product of a number by a 10-element numeric vector.
6. Design an algorithm to multiply two vectors
7. Design an algorithm to insert a value into a vector at a position chosen by the user. If the position is occupied, the data shifts to the right to make space to the new value. If the vector is full, the value is not inserted, and a message is displayed.
8. There is a vector of characters in which a phrase has been stored. Design an algorithm that determines if the phrase is a palindrome[10].
9. Design an algorithm to make the data in a vector circular: all elements move one position, and the last element moves to the first position.
10. Design an algorithm to determine if two matrices are equal; that is, verify that for all elements it holds that *A[i][j] = B[i][j]*
11. Design an algorithm to determine if two matrices contain the same elements, even if they are not in the same order.
12. Design an algorithm to add two matrices
13. Design an algorithm to determine if an array is symmetric
14. Design an algorithm to calculate the product of a number by an array

10. Palindromo: word or phrase that reads the same from left to right, as from right to left. (Circle Encyclopedia Universal, 2006:1670)

15. Design an algorithm to calculate the product of two matrices

16. Given the array *Letters[20][20],* which is completely filled with alphabetic characters, calculate the absolute frequency and the relative frequency for each of the vowels.

17. To bill the energy service, the company Energía Para Todos has a list of users stored in a vector and a list of readings for the previous month in a second vector. An algorithm is required to read the kW value, take the current reading of each user and record it in a third vector, calculate the consumption for the month by difference of readings, and display for each user: name, consumption, and amount to be paid.



6. SUBPROGRAMS

A digital computer
is like a calculation.
It can break a problem
into parts as small
as desired.
Van Doren
[Translation of the original epigraph in Spanish]

Numerous examples have been proposed and explained throughout this document; however, these were designed to illustrate a particular topic and do not reflect the complexity of real problems where the solution may involve thousands, hundreds of thousands, or millions of lines of source code.

Programs designed to solve real data processing problems, which cater to large sets of requirements and respond to multiple constraints, those that Booch (1994: 4) refers to as industrial-scale software, cannot be undertaken as a single piece. Building, debugging, and maintaining them would be too difficult and costly.

The most widespread strategy for reducing complexity is known as ***divide and conquer.*** This approach consists of breaking down the problem to be solved into a certain number of smaller subproblems, solving each subproblem successively and independently, and then combining the solutions obtained to, in this way, solve the original problem (Brassard and Bratley, 1997: 247), (Aho, Hopcroft, and Ullman, 1988: 307). Thus, a program is composed of several subprograms.

A subprogram or subroutine implements an algorithm designed for a particular task. Therefore, it cannot constitute a solution to a problem by itself; but it is called by another program or subprogram.

In Oviedo (2002), several advantages of using subprograms can be identified, such as: the fragmentation of the program into modules, which provides greater clarity and ease of distributing work among members of the development team, ease of writing, testing and debugging the code, a more understandable logical structure of the program, and the possibility of repeatedly executing the subprogram.

Considering the nature of the task that the subprogram performs and whether it returns data to the calling program, subprograms are categorized as functions or procedures (Oviedo, 2002).

6.1 FUNCTIONS

The concept of function in the field of programming was taken from mathematics and consists of establishing a relationship between two sets: origin and mirror, where each element of the first corresponds to one of the second (Gran Enciclopedia Espasa, 2005: 5140). In terms of Joyanes (1996: 166), a function is an operation that takes one or more values and generates a new value as a result. Values that enter the function are called arguments. In the same vein, Timarán et al (2009: 57) defines a function as: a subprogram that receives one or more parameters, executes a specific task, and returns a single value as a result, to the program or function that invoked it.

A function is represented as:

y = f(x)
f(x) = 3x - 2

Y is read as and depends on x, or y is a function of ***x***, and it means that to obtain the value of ***y***, it is necessary to replace ***x*** with a value (argument) and perform the operation indicated by the function. For example, if the function is solved with ***x = 2,*** then ***y = 4.***

An important characteristic of functions is that their result depends exclusively on the value or values they receive as arguments. If the Greatest Common Divisor function is taken, it requires two arguments, and the result it generates will be different for each pair of arguments.

Let ***f(x,z)*** be the Greatest Common Divisor of ***x*** and ***z.*** Then:

f(12,18) = 6
f(40,60) = 20

Functions are classified into two types: internal and external.

Internal Functions**.** These are defined within the programming language, distributed in the form of libraries[11] or APIs[12], and are used as a basis for program development. They address general needs and are usually grouped depending on the domain in which they can be used. For example: mathematics, strings, dates, and files.

External Functions. Also known as user-defined functions, these are written by the programmer to meet specific needs in their applications.

6.1.1 Designing a Function

A function has two parts: the definition and implementation. Some authors, such as Joyanes (1996: 167) and Galve et al., (1993: 30), refer to them as the header and the body, respectively. In a program, for example in C language, the function definition provides the compiler with information about the function's characteristics, whose implementation will appear later. This helps verify the program's correctness.

Function Definition**.** This includes three elements that are explicit in pseudocode notations and Nassi-Shneiderman diagrams, but not in the flowcharts. These are:

a. The return type
b. The function's identifier or name
c. The list of parameters

Until a few years ago, it was common to define a function in the following form:

Function identifier(parameters): return_type

Example:

Factor function(integer n): integer

This instruction defines the function called factorial which receives an integer as a parameter and stores it in the variable n and returns an integer as a result.

11 It is a collection of standardized and tested programs and subprograms that can solve specific problems (Lopezcano, 1998: 322).

12 API is an acronym for Application Programming Interface, which in Spanish means: *Interfaz para programación de aplicaciones o conjunto de herramientas para desarrollar un programa*.

Currently, to maintain the similarity with current programming languages, functions are defined as:

Return_type identifier(parameters)

Examples:

Integer factorial(integer n)
Real power(integer base, integer exponent)
Boolean isPrime(integer n)

The reserved word function has been deleted, and the return type is placed at the beginning of the definition. This is the format used throughout the book for defining functions.

Parameters. Since a function specializes in performing a task, which is usually a calculation, it cannot be expected that it is also responsible for reading and printing data. Therefore, it is necessary to provide the data that it requires to perform the calculation. These data are called parameters.

In this vein, parameters are the data sent to the function to perform the task for which it was designed. For instance, if a function is designed to calculate the factorial, it is necessary to pass the number for which it is desired to find the factorial as a parameter. If a function is designed to calculate a power, it is necessary to provide it with the base and the exponent.

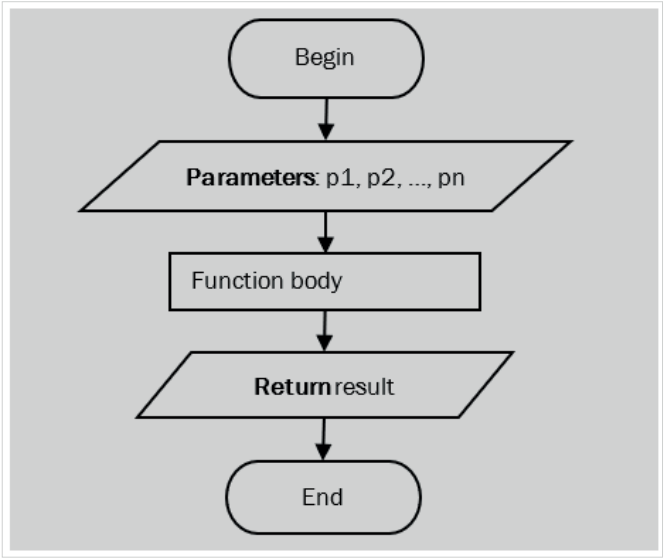
The parameter declaration is part of the function definition. They are enclosed in parentheses, indicating the data types and identifiers that the function will receive. These declarations are constituted as local variables of the function. In the definition:

Real power(integer base, integer exponent)

It expresses that the power function will receive two integer-type data and store them in the variables: ***base*** and ***exponent***.

In flowchart, a function does not have an explicit definition. However, in this document, it is differentiated from a program by including an input/output symbol to receive the parameters and another to return the result, as shown in Figure 106.

Figure 106. *Flowchart of a Function*



While the definition is essential since it gives existence and identity to the function, the implementation is what allows the function to perform an operation.

Implementation**.** It consists of a set of lines of code or instructions that process the data received as parameters and produce the result expected by the program invoking the function. The function body starts after the definition and ends with the reserved word ***end*** followed by the function name. However, in the penultimate line, the instruction *Return or Deliver* must appear, followed by a value or an expression, as shown in examples 54 and 55.

Example 54. Sum Function

Design a function to add two integers

This example aims to show the structure of a function, clearly presenting its definition, parameters, implementation, and return.

The function sum receives two integer parameters, adds them, and returns the result, as shown in Table 83.

Table 83. *Pseudocode for the Sum Function*

```
1    Integer sum(integer a, integer b)
2        Integer result
3        result = a + b
4        Return result
5    End sum
```

Note that the function is defined as an integer type, which means that the data it will return will be of this type. Consequently, the local variable declared to store the result is of the integer type. As for the parameters, although they are of the same type, it is necessary to specify for each one that they are integers since this is not always the case.

Variables that are declared in the body of a function are local in scope, and therefore can only be accessed within the function. Once the execution ends, the variables cease to exist.

Example 55. Factorial Function

Design a function to calculate the factorial of a positive integer. It is known that the factorial of 0 is 1, and the factorial of any number ***n*** greater than 0 is the product of the numbers between 1 and ***n***.
The factorial function requires one parameter: the number, and it returns another number: the factorial. Its implementation includes a loop that calculates the product from 1 to ***n***, as shown in the pseudocode in Table 84.

Table 84. P*seudocode for the Factorial Function*

```
1    Integer factorial(integer n)
2        Integer fac = 1, counter
3        For con = 1 to n do
4            fac = fac * counter
5        End for
6        Return fac
7    End factorial
```

6.1.2 Calling a Function

The invocation or call to a function is done by writing its name followed by the list of parameters that it requires, as follows:

Function_name(parameters)

Example:

Add(123, 432)
Factorial(5)

When an algorithm or program is executed, and a function is invoked, the execution control passes to the function. In the function's execution, the first task performed is the declaration of local variables and their initialization with the data provided as parameters. Therefore, it is essential to note that the arguments written when invoking the function and the list of parameters defined in the function design must match in type and quantity; otherwise, an error will occur, and the function's execution will be canceled.

When the function's execution encounters the *return* statement, the control returns exactly to the line where the function was invoked with the result of the calculation performed.

The result returned by a function can be stored in a variable, sent directly to an output device, or used as an argument to another function, as shown in expressions *a, b*, and *c*, respectively.

x = factorial(8)
Write "Factorial of 8:", factorial(8)
add(factorial(3), factorial(4))

Example 56. Arithmetic Operations

Design the main algorithm and the necessary functions to perform arithmetic operations: addition, subtraction, multiplication, and division of two numbers.

To solve this exercise, it is necessary to design four functions, one for each operation, and a main program that calls the corresponding function based on the user's

request. Since each function is exclusively responsible for performing the calculation, the main program must handle reading the numbers, the operation to be performed, and displaying the result.

In the implementation of programs in some languages, it is required to define the functions before invoking them. To develop good programming practices, in this exercise and the following ones, the functions or procedures are designed first and at the end of the main program.

The functions and the program for this exercise are presented in Tables 85, 86, 87, and 88. The add function has already been designed and appears in Table 83, so it is not included here.

Table 85. *Pseudocode for the Subtract Function*

```
1   Integer subtract(integer a, integer b)
2       Integer difference
3       result = a – a
4       Return difference
5   End subtract
```

Table 86. *Pseudocode for the Multiply Function*

```
1   Integer multiply(integer a, integer b)
2       Integer product
3       result = a * b
4       Return product
5   End multiply
```

Table 87. *Pseudocode for the Divide Function*

```
1   Integer divide(integer a, integer b)
2       Real quotient = 0
3       If b != 0 then
4           quotient = a / b
5       End if
6       Return quotient
7   End divide
```

Table 88. *Pseudocode for the Arithmetic Operations Algorithm*

```
1    Begin
2        Integer x, y, opt
3        Read x, y
4        Write "1. Add 2. Subtract 3. Multiply 4. Divide."
5        Read opt
6        Switch opt do
7            1: Write "summation =", sum(x,y)
8            2: Write "Difference =", subtract(x,y)
9            3: Write "Product =", multiply(x,y)
10           4: If y = 0 then
11                   Write "Error, divisor = 0"
12               Else
13                   Write "Quotient =", divide(x,y)
14               End if
15       End switch
16   End sum
```

In lines 7, 8, 9, and 13 the calls to the previously defined functions are made. Although the divide function includes a conditional to avoid the error of dividing by zero, the data validation is done in the main algorithm (line 10) because the function can only return one value: the quotient, not an error message.

6.1.3 More Examples of Functions

Example 57. Power Function

Exponentiation is a mathematical function that consists of a base and an exponent: a^n. The result is the product of multiplying the base by itself as many times as the exponent indicates, as explained in Example 40.

This function requires two parameters: the base and the exponent. Its implementation includes a loop to calculate the product from one to the absolute value of n and a decision to determine whether the exponent is negative, in which case the result is one divided by the product obtained.

$$a^{-n} = \frac{1}{a^n}$$

In Table 89, the function to obtain the absolute value of a number is designed. In Table 90, the power function, and in Table 91, the main algorithm that invokes power.

Table 89. *Pseudocode for the Absolute Value Function*

```
1   Integer absolutevalue (integer n)
2       If n < 0 then
3           n = n * -1
4       End if
5       Return n
6   End absolutevalue
```

Table 90. *Pseudocode for the Power Function*

```
1   Real power(integer a, integer n)
2       Integer p= 1, x
3       For x = 1 to absolutevalue(n) do
4           p = p * a
5       End for
6       If n < 0 then
7           p = 1 / p
8       End if
9       Return p
10  End power
```

Table 91. *Pseudocode for the Main Algorithm for Exponentiation*

```
1   Begin
3       Integer b, e
4       Read b, e
5       Write "Power = ", power(b, e)
6   End algorithm
```

This example shows how the use of functions facilitates solving the problem. In this case, the main algorithm is responsible for reading the data and displaying the result, the *power()* function is responsible for calculating the result, and the *absolute value()* function makes it possible to calculate both positive and negative exponents.

To verify that a function produces the expected result, the same method is followed for the other algorithms: it is executed step by step, and the data stored in the varia-

bles is recorded. To verify the correctness of the overall solution, each variable and function is placed as a column to record the result it returns, as shown in Tables 92 and 93.

Table 92. *Verification of the Power Function*

Execution	a	n	x	Absolutevalue(s)	p
1	4	3			1
			1	3	4
			2		16
			3		64
2	3	-4			1
			1	4	3
			2		9
			3		27
			4		81
					1/81

Table 93. *Verification of the Algorithm to Calculate a Power*

Execution	b	e	Power(b, e)	Output
1	4	3	64	Power = 64
2	3	-4	1/81	Power = 1/81

Example 58. Simple Interest Function

Design a function to calculate the simple interest of an investment.

Simple interest is the benefit obtained by investing a certain amount of capital. Three elements are involved in this operation: the capital invested, the interest rate or ratio per unit of time, and the agreed periods in the investment.

For example: Juan lends 10 million pesos to Carlos for a period of six months, and Carlos agrees to pay an interest rate of 2% per month at the end of the six months. How much will Juan receive in interest?

The formula for calculating simple interest is:

I = C * R * T

Where (expressed in Spanish):
I: Simple interest
C: Invested capital
R: Interest rate or ratio, expressed as a percentage
T: Time

Using the data from the example, we have:

C = 10,000,000
R = 2%
T = 6
Thus:

I = 10,000,000 * 2 * 6 / 100 = 1,200,000

Now, the simple interest formula needs to be expressed as a subroutine or function, as shown in the flowchart notation in Figure 107.

Figure 107. *Flowchart of the Simple Interest Function*

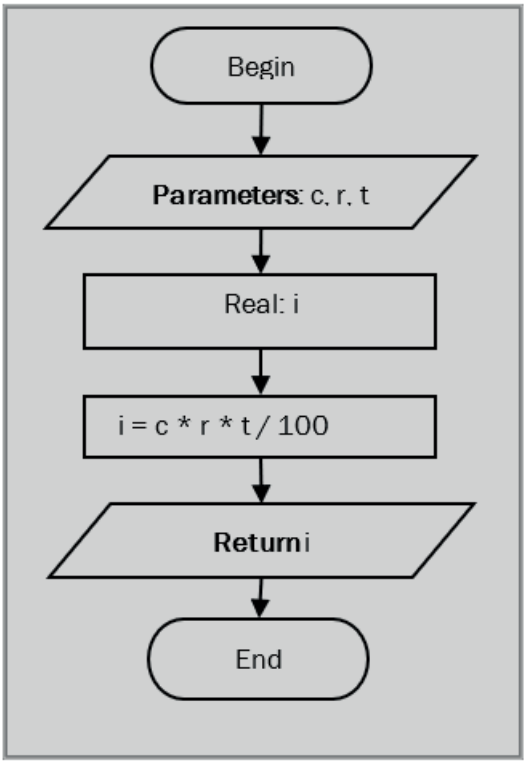
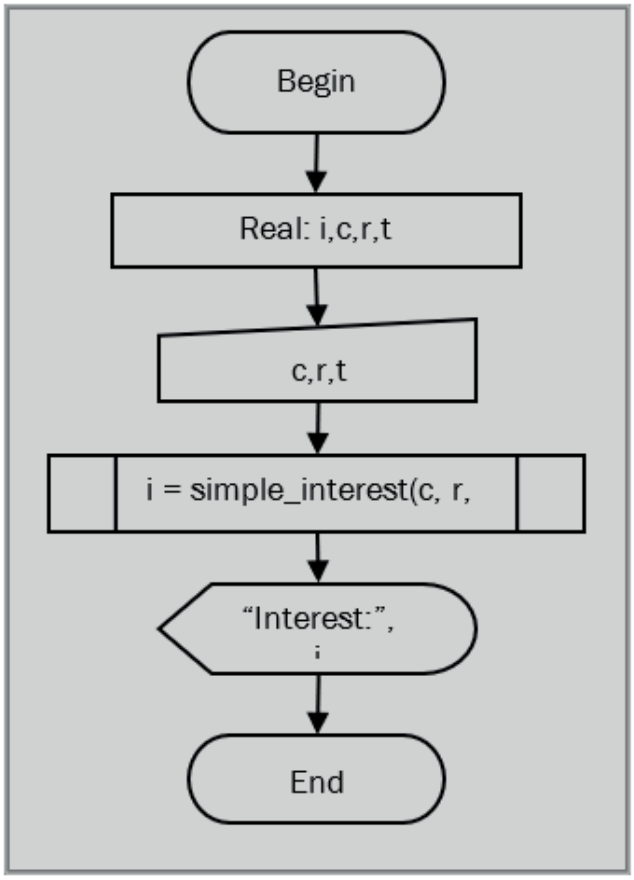


Figure 108 shows the flowchart of the main algorithm for calculating the simple interest of an inversion. It is responsible for reading the data, invoking the function, receiving the result, and finally displaying it to the user.

In this example, a variable i is used in both the main program algorithm and the function. It should be noted that these are two independent variables because they are declared as local variables in each algorithm. Therefore, one does not exist in the scope of the other.

Figure 108. *Flowchart for the Simple Interest Algorithm*



The results of the step-by-step verification of this algorithm are presented in Table 94.

Table 94. *Verification of the Solution for Calculating Simple Interest*

Execution	C	R	T	I	Output
1	10000000	2%	6	1200000	Interest: 1200000
2	20000000	1.5%	12	3600000	Interest: 3600000

Example 59. Final Grade Function

At Buena Nota University, the final grades for each subject are obtained by averaging three partial grades, where the first and second grades are equivalent to 30% each and the third to 40% of the final grade. A function is required to perform this calculation.

According to the above, if a student receives the following grades:

Grade 1 = 3.5
Grade 2 = 4.3
Grade 3 = 2.5

Their final grade will be:
Final grade = 3.5 * 30/100 + 4.3 * 30/100 + 2.5 * 40/100
Final grade = 3.3

In general, the grade is obtained by applying the formula:

Final grade = grade 1 * 30/100 + grade 2 * 30/100 + grade 3 * 40/100

The function to perform this task is shown in Table 95.

Table 95. *Pseudocode for the Final Grade Function*

```
1    Real finalgrade (real g1, real g2, real g3)
2        Real fg
3        fg = g1 * 30/100 + g2 * 30/100 + g3 * 40/100
4        Return fg
5    End finalgrade
```

Example 60. Triangle Area Function

The area of a triangle is equivalent to half the product of its base times its height.

Area = base * height / 2

Therefore, the function for calculating the area of the triangle requires the base and height to be provided. The pseudocode for this function is shown in Table 96.

Table 96. *Pseudocode for the Triangle Area Function*

```
1    Real areatriangle(real base, real height)
2        Real area
3        area = base * height / 2
4        Return area
5    End areatriangle
```

Example 61. Leap Year Function

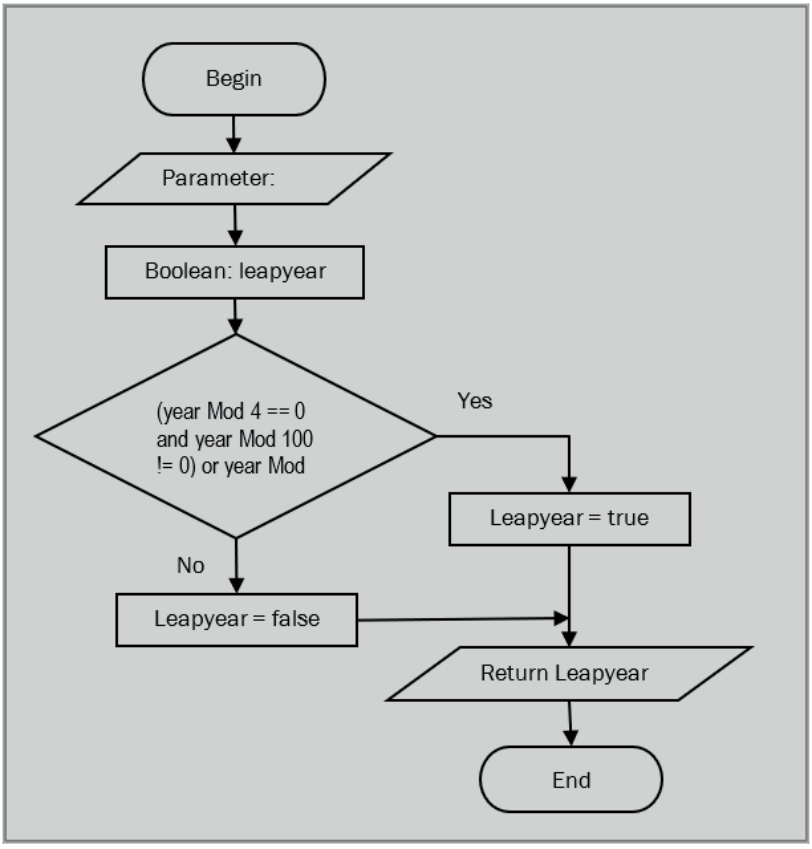
In the Gregorian calendar, there is a leap year13 every four years, except for the last one of a century, which is not divisible by 400. Design a function to determine if a year is a leap year.

Take, for example, the year 1700. It is divisible by 4 and by 100, but it is not exactly divisible by 400; therefore, it is not a leap year. Meanwhile, the year 1904 is divisible by 4, but not divisible by 100, so it is a leap year. The year 2000 is divisible by 4, by 100, and by 400, so it is a leap year.

The condition for a leap year is: that it is divisible by 4 and 400 but not by 100. That is, any year divisible by 4 that is not the last of a century is a leap year, but if it is the last year of a century, it must be divisible by 400. The function is shown in Figure 109.

13 A leap year has 366 days as opposed to the others with 365. This additional day is included in February, which in this case has 29 days.

Figure 109. *Flowchart for the Leap Year Function*



Example 62. Count Characters Function

Design a function that receives as parameters an array of characters and a character. The function must count the number of times the character appears in the array and return the counter.

For example, if the array consists of the characters: a, b, c, d, e, d, c, b, a and the character to search for is b, it appears twice in the array.

The function must iterate through the vector and, in each position, compare its content with the reference character, and if they match, it increments the counter. The problem is that for the traversal, it seems necessary to know the size of the vector, and that information is not available.

Some programming languages have an internal function that returns the size of a vector, some others report a false or null value when trying to access a position that does not exist in the vector. In this case, to avoid any possibility of error, a third parameter corresponding to the size of the vector is included, so that the traversal can be performed without difficulty. This function is shown in Table 97.

Table 97. *Pseudocode for the Character Count Function*

```
1  Real countcharacters(character v[], character char, integer n)
2      Integer i, count = 0
3      For i = 1 to n do
4          If char = v[i] then
5              count = count + 1
6          End if
7      End for
8      Return count
9  End countcharacters
```

Example 63. Adding Days to a Date

A function is required to add a certain number of days to a date (year, month, day).

Suppose the date is: 2000/07/01, and 5 days are added. The date obtained will be: 2000/07/06. In this case, only the days are added. But if the date is 2005/12/28 and 87 days are added, what will be the new date? What operations should be performed?

To perform this addition, the first step is to extract the years and months that are in the number of days, thus obtaining data in date format. The purpose of this is to add days to days, months to months, and years to years.

Years = 87 / 365 = 0
Months = 87 – (years * 365) / 30 = 2
Days = 87 - (years * 365) – (months * 30) = 27

From this, the operation to perform is:
2005/12/28 +
0000/02/27

Now, the days are added: 28 + 27 = 55. Since the result is greater than 30, it is separated into months and days: 1 month and 25 days. These days are recorded, and 1 is added to sum it with months.

Months are added: 12 + 2 + 1 = 15. Since the result is greater than 12, it is divided into years and months: 1 year and 3 months. The 3 months are recorded, and 1 is added to sum it with years.

The years are added: 2005 + 0 + 1 = 2006.

Therefore, the solution is:

$$\begin{array}{r} 2005/12/28 + \\ 0000/02/27 \\ \hline 2006/03/25 \end{array}$$

These operations must be expressed algorithmically and placed in a function that can be invoked as many times as needed. Something very important to note is that, by definition, a function returns a value, and in this case, it is required to return three values (year, months, and days). The solution consists of grouping the data using a vector, where the first position will correspond to the year, the second to the months, and the third to the days. The vector is then set as a parameter, and similarly, the function will return a vector as a result. The pseudocode for this function is shown in Table 98.

Table 98. *Pseudocode for the Function to Add Days to a Date*

```
1   Integer[] adddaystodate(Integer d[], Integer days)
2       Integer aux[3], nd[3]
3       aux[1] = days / 365
4       days = days Mod 365
5       aux[2] = days / 30
6       aux[3] = days Mod 30
7       nd[3] = d[3]+aux[3]
8       nd[2] = d[2]+aux[2]
9       nd[1] = d[1]+aux[1]
10      If nd[3] > 30 then
11          nd[3] = nd[3]-30
12          nd[2] = nd[2] + 1
13      End if
14      If nd[2] > 12 then
```

```
15              nd[2] = nd[2]-12
16              nd[1] = nd[1] + 1
17          End if
18          Return nd[]
19      End adddaystodate
```

This function receives a vector of three integers and a variable as parameters. Internally, it declares two vectors, one auxiliary vector to discriminate the content of the received variable into years, months, and days, and a second vector to store the date generated as a result. Initially, the positions of the vectors are added directly, then the validity of the days and months is verified, and if they are not valid, the corresponding adjustment is made.

Table 99 shows the behavior of the vectors when executing the function.

Table 99. *Verification of the Function to Add Days to a Date*

Vector d[]			**Days**	**Vector aux[edit]**			**Vector nd[]**		
d[1]	d[2]	d[3]		aux[1]	aux[2]	aux[3]	nd[1]	nd[2]	nd[3]
2005	12	28	87	0	2	27	2005	14	55
								15	25
							2006	3	25

6.1.4 Proposed Exercises

Design functions for the following approaches

1. Determine the minimum grade a student must obtain in the final exam of a subject to pass it with a minimum grade (3.0) based on the grades obtained from two midterms exams. It is known that each midterm exam is equivalent to 30%, and the final exam is equivalent to 40% of the final grade.

2. Calculate the perimeter of a rectangle

3. Calculate the area of a circle

4. Calculate the circumference of a circle

5. Calculate the volume of a cylinder

6. Determine the amount of paper required to completely cover a box
7. Establish an athlete's speed in km/h, knowing it took him/her x minutes to complete a lap around the stadium, with a distance of n meters.
8. Subtract a number of days from a date in year/month/day format
9. Add a number of years, months, and days to a date
10. Find the difference between two dates expressed in years, months and days.
11. Calculate the future value of an investment with compound interest
12. Determine if a number is prime
13. Find out if a number n is part of the Fibonacci series
14. Evaluate a date and report whether it is valid or not. Consider months with 28, 30, and 31 days, and leap years.
15. Calculate the selling price of a product by applying 30% profit on cost and 16% value added tax (VAT).
16. Calculate the least common multiple of two numbers
17. Calculate the sum of the first n numbers
18. Reverse the order of the elements in a vector

6.2 PROCEDURES

A procedure is a set of lines of code written separately from the main algorithm with the purpose of requesting its execution from various parts of the algorithm or program. This makes the code more organized and easier to understand, write, correct and maintain.

Procedures, like functions, perform a specific task, but unlike functions, they are not designed to carry out calculations; therefore, they do not return any value.

The design of a procedure includes its definition and implementation. In the definition, a name is assigned, and the parameters it will receive are specified. In the

plementation, detailed instructions that will allow it to fulfill its task are written. The general form to design a procedure is:

Procedure_name(parameters)
Instructions
End procedure_name

Examples:

Print(integer a, integer b, integer c)
Write a
Write b
Write c
End print

If variables are declared within the body of the procedure, they have a local scope.

To invoke a procedure, its name and the list of parameters are written, considering that these must match the number and type of the list of parameters of the procedure definition. When the execution of an algorithm encounters the invocation of a procedure, it executes its instructions, and once completed, it returns to the line following the invocation.

Example 64. Multiplication Table Procedure

An algorithm is required to generate multiplication tables for the number the user enters, the execution ends when 0 is entered.

In this exercise, two tasks can be observed: on one hand, managing the inputs, that is, reading the number and deciding whether to generate the table or end the execution of the algorithm, and repeating this operation until a 0 is entered; on the other hand, generating the multiplication table for the entered number. The second task can be delegated to a procedure

This procedure receives the number for which the table is required as a parameter and is responsible for carrying out the iterations and displaying the product in each of them. The pseudocode for this procedure is shown in Table 100.

Table 100. *Procedure for Multiplication Table*

```
1    Multiplytable(Integer n)
2        Integer i
3        For i = 1 to 10 do
4            Write n, "*", i, " =", n*i
5        End for
6    End multiplytable
```

The main algorithm, from which this procedure is invoked, is presented in Table 101. The call to the procedure is made from line 5.

Table 101. *Pseudocode for the Algorithm to Generate Multiplication Tables*

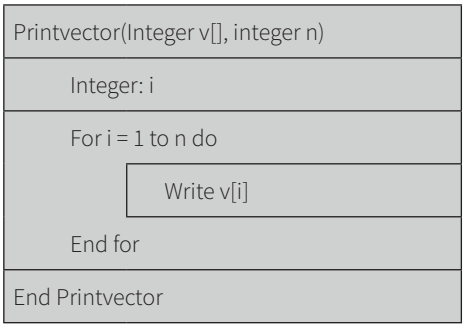
```
1    Begin
2        Integer n
3        Do
4            Read n
5            Multiplytable(n)
6        While n != 0 do
7    End algorithm
```

Example 65. Print Vector Procedure

Design a procedure that can be invoked as often as desired to list the data of a vector.

This procedure must receive as parameters the vector and the number of elements it contains, iterate through it, and display each element. The N-S diagram of this procedure is shown in Figure 110.

Figure 110. *N-S Diagram for the Print Vector Procedure*



Example 66. Write Date Procedure

Design a procedure that receives three integers corresponding to the month, day, and year, and displays the date as text.

This procedure must determine the name of the month and then format the date properly. For example, if the input values are: month = 2, day = 11, and year = 2012, it should display: "***February 11, 2012***." The pseudocode for this procedure is shown in Table 102.

Table 102. *Pseudocode for the Procedure to Write Date*

```
1   Writedate(integer m, integer d, integer y)
2       String monthname
3       Switch m do
4           1: monthname = "January"
5           2: monthname = "February"
6           3: monthname = "March"
7           4: monthname = "April"
8           5: monthname = "May"
9           6: monthname = "June"
10          7: monthname = "July"
```

Table 102. *(continued)*

```
1           8: monthname = "August"
2           9: monthname = "September"
3           10: monthname = "October"
4           11: monthname = "November"
5           12: monthname = "December"
6       End switch
7       Write monthname, d ",", y
8   End Writedate
```

Example 67. Procedure to Display a Sequence

Design a procedure to generate the first ***n*** terms of the sequence created by the expression*: $x^2 - x$.*

The procedure consists of creating a loop from 1 to ***n*** and applying the expression. The results from the sequence:

0, 2, 6, 12, 20, 30, . . .
The pseudocode is shown in Table 103.

Table 103. *Pseudocode for the Procedure to Display a Sequence*

```
1    showsequence(integer n)
2        Integer x, t
3        For x = 1 to n do
4            t = x * x – x
5            Type x
6        End for
7    End showsequence
```

7. SEARCH AND SORTING

A man who does not know
the way to the sea
ought to seek a river
to accompany him.
Plautus

Search and sorting are two fundamental tasks in data management, especially when dealing with large volumes. Both operations are performed based on reference data, commonly called a key. Searching allows finding a particular element in a set while sorting consists of organizing the data according to a criterion, making it easier to locate the element required or to identify relationships between the data.

In this chapter, search and sorting algorithms are studied and explained in detail, supported by examples and graphs that facilitate understanding.

7.1 SEARCH ALGORITHMS

Search is a crucial operation when handling large datasets where locating an element is not an easy task, as stated by Lopez, Jeder and Vega (2009: 129).

There are two methods to search for data in a vector: sequential or linear search, and binary search. The former is easier to implement but may take more time, the latter is more efficient but requires the vector to be sorted.

7.1.1 Linear Search

This method consists of taking key data that identify the element being searched for and traversing the entire array, comparing the reference data with the data of each position.

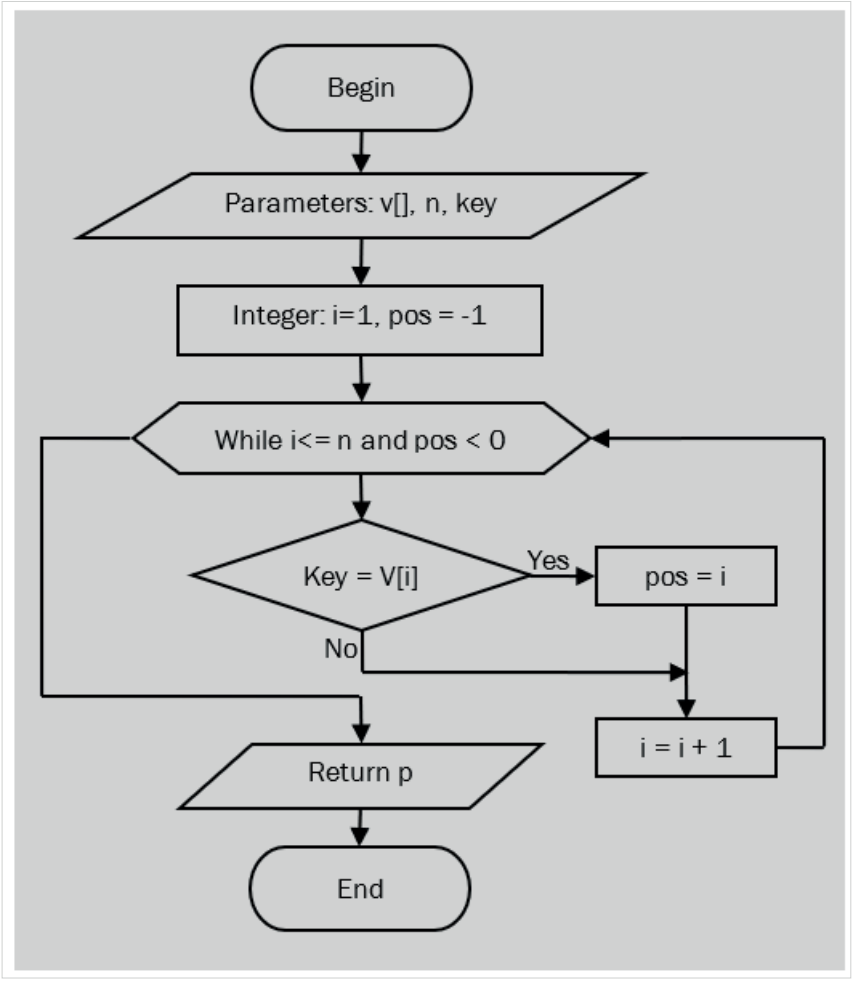
Suppose there is a list of students, and you want to locate the one identified by the number 27844562. The search consists of comparing this number with the identification of each student in the list. The search will end in the event of finding a coincidence in the numbers or if, at the end of the list, no identification equal to the

searched number is found, in which case it is concluded that the data does not exist in the vector.

Figure 111 shows the flowchart of a function with the general steps for performing a sequential search in a vector.

The function receives as parameters the vector and the key of the element to be searched and traverses the vector until the element is found, in which case the position is recorded in the variable pos or until the end of the vector is reached. When the algorithm finishes, it returns the pos variable. If the data was found, it contains the position it occupies in the vector; otherwise, it reports the value minus one (-1), indicating that the data is not in the vector.

Figure 111. *Flowchart for the Linear Search Algorithm*

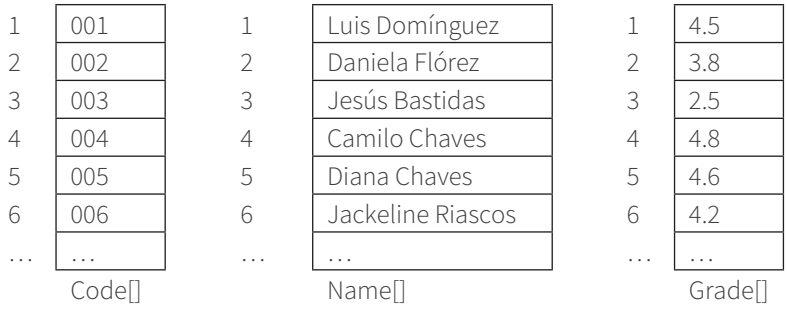


7.1.2 Examples of Linear Search

Example 68. Search for a Student

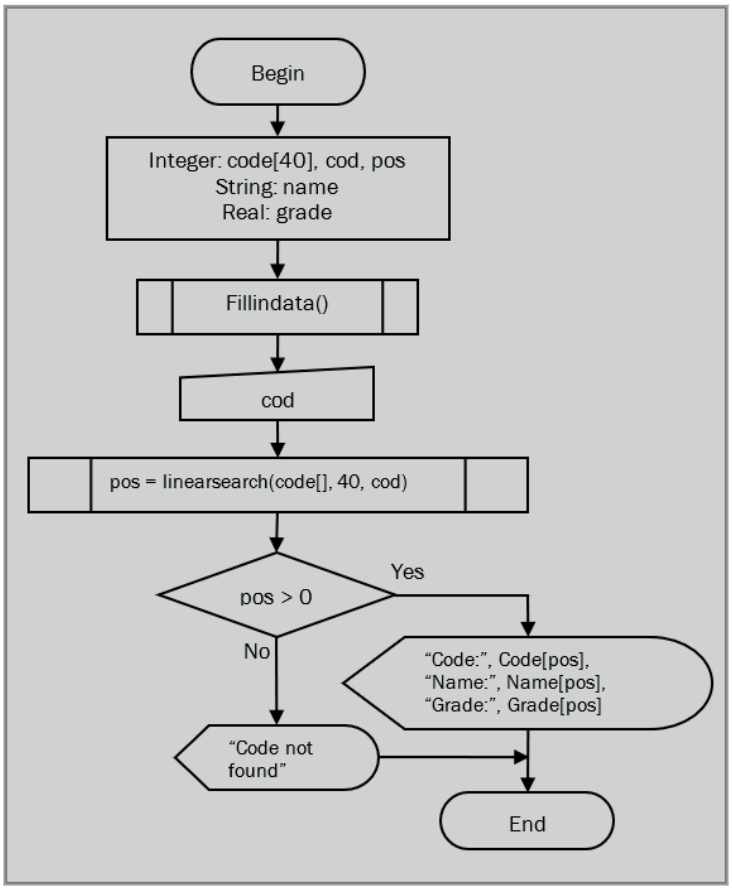
A teacher stores the data of their student in three vectors: code, name, and grade, as shown in Figure 112. An algorithm is required to query a student's grade based on their code.

Figure 112. *Student Data Stored in Vectors*



By using the *linearsearch()* function and passing the code vector, its size, and the student's code as parameters, it is possible to find the position of the student's data in the vectors. The algorithm is shown in Figure 113, which assumes the existence of a procedure to fill the data in the vectors.

Figure 113. *Flowchart for the Student Search Algorithm*



Example 69. Check Balance

In a hypothetical ATM, when a customer requests their account balance, they swipe the card, the reader takes the account number, and the system searches for that number in the database. If it finds it, it reports the balance; if not, it displays an error message. Assuming the database consists of three vectors: account number, cardholder and balance. Design an algorithm to check an account balance.

In this algorithm, the account number is read, and the linearsearch() function is invoked by providing the vector with the account numbers and the read data. This function performs the search and returns the position where the corresponding account data is located. If the data returned by the function is a positive integer, the data from all vectors are displayed. If it is -1, it means the account does not exist, and

a message is displayed informing of this situation. The pseudocode for this algorithm is shown in Table 104

Table 104. *Pseudocode for the Algorithm to Check Balance*

```
1    Begin
         Integer num, pos, account[1000]
2        String: holder[1000]
         Real: balance[1000]
3        Fillindata()
4        Read num
5        pos = linearsearch(account[], 10000, num)
6        If pos > 0 then
7            Write “Account:”, account[pos]
8            Write “Holder: “, holder[pos],
9            Write “Balance: “, balance[pos]
10       Else
11           Write “Account number not found”
12       End if
13   End algorithm
```

7.1.3 Binary Search

This method is more efficient than sequential search but can only be applied on sorted vectors or lists of data, as confirmed by López, Jeder, and Vega (2009: 130)

In binary search, the search is not conducted from beginning to end; instead, the search space is progressively narrowed until the searched element is reached. The first comparison is made with the middle element of the array. If it is not the searched data, it is decided whether to search in the lower half or the upper half, depending on whether the key is smaller or greater than the element in the middle. Half of the corresponding vector is taken as search space and proceeds in the same way; it is compared with the middle element. If that is not the searched element, a new search space is taken corresponding to the lower or upper half of the previous space, it is compared again with the middle element, and so on, until the element is found, or the search space is reduced to one element.

In binary search, the number of elements making up the search field is halved with each iteration, as follows:

n for the first iteration (n = vector size),
$n/2$ for the second,
$n/2^2$ for the third, and
$n/2^3$ for the fourth.

In general, the search space in the i-th iteration is made up of $n/2^{i-1}$ elements.

Assume a vector ***v*** with 20 elements arranged d in ascending order, as shown in Figure 114, where the number 10 is searched. (key = 10).

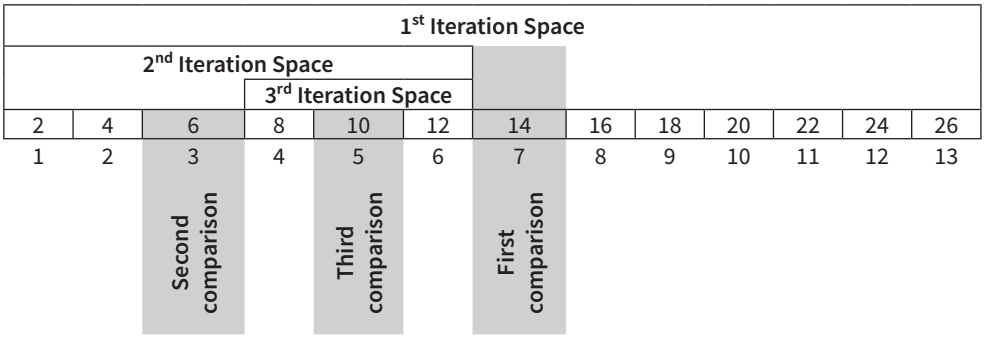
The first iteration takes the entire vector as the search space and locates the middle element. To determine which is the middle element, the index of the lower endpoint is added to the index of the upper endpoint of the search space and divided by two.

(1 + 13) / 2 = 7

It is checked if the key is in the middle element:

key = v[7]?
10 = 14?

Figure 114. *Flowchart for the Linear Search Algorithm*



Since the data is not in position 7, a new search space is defined corresponding to the lower half of the vector, given that the key (key = 10) is smaller than the middle element ***(v[7] = 14)***. The space for the second comparison is between positions 1 and 6.

The middle is recalculated:
(1 + 6) / 2 = 3

Now the middle element is the one in position 3, and it is checked if this is the data being searched for:

key = v[3]?
10 = 6?

Since the data is not found at this position either, a new search is performed, this time taking the upper half of the space because the key is greater than the data in the middle: 10 > 6.

The new search space comprises elements between positions 4 and 6. The middle element is recalculated:

(4 + 6) / 2 = 5
The key is compared with the element of the middle position

key = v[5]?
10 = 10?

The values are equal; the data was found in this iteration. The search ends. Three comparisons were needed to find the element.

The function to perform the binary search receives the vector with the data, the size of the vector, and the data to be searched or search key as parameters. The pseudo-code is shown in Table 105.

Table 105. *Pseudocode for the Binary Search Function*

```
1   Integer binarysearch(integer v[], integer n, integer key)
2       Lower Integer = 1, upper = n, middle, pos = -1
3       While lower <= upper and pos < 0 do
4           middle = (lower + upper) / 2
5           If v[middle] = key then
6               pos = middle
7           Else
8               If key > v[middle] then
9                   lower = middle + 1
10              Else
11                  upper = middle – 1
```

```
12                End if
13            End if
14        End while
15        Return pos
16    End binarysearch
```

The function declares four local variables, two to maintain the limits of the search space: ***lower*** and ***upper***. These start at 1 and ***n***, respectively, which allows the first iteration to be applied over the entire vector. The variable middle is updated at each iteration, and its purpose is to identify the data located at the center of the search space to compare it with the key. Finally, the variable ***pos*** is used to store the position of the element in case of finding it; it is initialized to -1, indicating that the data has not been found. The loop has two conditions: that the lower limit is less than the upper limit, that is, there is a search space, and that the data has not been found. Once the data is found, the variable ***pos*** takes a positive value, and the loop stops executing.

At the end of the loop, the variable ***pos*** is returned. If it has a positive value, it corresponds to the position where the searched data is found. If it contains -1, it means the data is not in the vector.

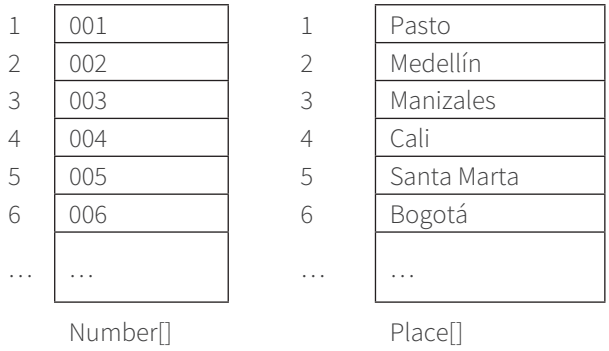
7.1.4 Binary Search Examples

Example 70. Winning Number

The Mundo Rico lottery records the number of tickets sold and the place where they were sold in ascending order. At the time of playing the lottery, once the winning number is known, it is required to immediately know the place where it was sold. Design an algorithm to perform this query.

The data is stored in a two-vector structure as shown in Figure 115. Since the numbers are sorted, the binary search algorithm can be applied. It will find the number much faster than if a linear search was used.

Figure 115. *Number of Lottery Tickets and Sales Location*



The operations to be carried out in this exercise are: reading the winning number, passing it to the search function, and displaying the results. The function provides the position where the data is found; otherwise, it will report that the data is not found. This is interpreted as an unsold number. Reading data and printing results are actions that the main algorithm must perform. See Table 106.

Table 106. *Pseudocode for the Winning Number Algorithm*

```
1   Begin
2     Integer number[1000], place[1000], winner, pos
3     Fillindata()
4     Read winner
5     pos = binarysearch(number[], 1000, winner)
6     If pos> 0 then
7           Write “The number: “, winner, “ was sold at: “, place[pos]
8     Else
9           Write “The number was not sold”
10    End if
11  End algorithm
```

In line 3 of the algorithm, a procedure is invoked to fill the data, ensuring that the vectors contain data, although its implementation is not included. In line 5, the binarysearch() function is invoked, sending the vector containing the numbers, its size, and the winning number as parameters.

Example 71. Medical Record

At the Curatodo clinic, there is an index of medical records using two vectors: the first records the medical record number, and the second the patient's identification. An algorithm that reads the patient's identification and reports the medical record number is required.

As seen in the previous examples, in both sequential and binary search, once the function is available, it can be used for any vector. In this case, the algorithm basically consists of reading the patient's identification number, invoking the function, and displaying the vector data at the position reported by the function or, failing that, an "Identification not found" message. The algorithm is shown in Table 107.

Table 107. *Pseudocode for the Medical Record Algorithm*

```
1   Begin
2     Integer identification[1000], record[1000], idnumber, pos
3     Fillindata()
4     Read idnumber
5     pos = binarysearch(Identification[], 1000, idnumber)
6     If pos> 0 then
7         Write "The patient's medical record is: ", record[pos]
8     Else
9         Write "No medical record found for the patient"
10    End if
11  End algorithm
```

7.1.5 Proposed Exercises

1. The telephone directory information is stored in a 3-column array. The first column contains the last name, the second the first name, and the third the phone number. An algorithm is required to search for a person's phone number by using their last name and first name as the key.

2. The list of books available in a library is stored in a 5-column array. The first column contains the author, the second the title, the third the publisher, the fourth the year, and the last, the signature. Design the algorithm and the search function by title.

3. On election day, at each polling station, there is a list of people who will vote at that station. The data is stored in two vectors; the first contains the ID card number, and the second the name. An algorithm is required that enters a person's ID card number and reports whether they can vote at that polling station.

4. The payroll of the company BuenSalario is recorded in two vectors and one array. The first vector stores the ID number, and the second the name. The array stores the following data: base salary, deductions, and net payable. An algorithm is required that reads an employee's ID card and displays their payroll data.

5. In the Todocaro store, data such as the invoice number, customer name, billing date, and invoice value are stored in vectors. An algorithm is required that reads the invoice number and displays the other data.

6. The communications company Línea Segura records the origin, destination, date, time, and duration of all calls made by its users. It uses five vectors for this purpose. An algorithm is required to check if a specific phone has made a call to a destination and, if so, provides the date, time, and duration of the call.

7. The airline company Vuelo Alto maintains information about all its flight routes in a database with details such as: places of origin and destination, schedule, and price. It uses an array where each column is dedicated to one of the mentioned pieces of information. Design an algorithm to check if a flight exists between a user-provided origin and a destination, and if so, retrieve information about it.

7.2 SORTING ALGORITHMS

The data in a vector, a list, or a file are sorted when each element occupies the correct position according to its value, a key data, or criterion (this document only addresses the sorting of vectors). If each element of the vector, other than the first one, is greater than the previous ones, it is said to be sorted in ascending order, while if each element, other than the first one, is smaller than the previous ones, it is sorted in descending order.

In a small data set, the order in which they are presented may be irrelevant, but when the amount increases, the order becomes important. How useful would a dictionary be if it were not sorted? Or the telephone directory?

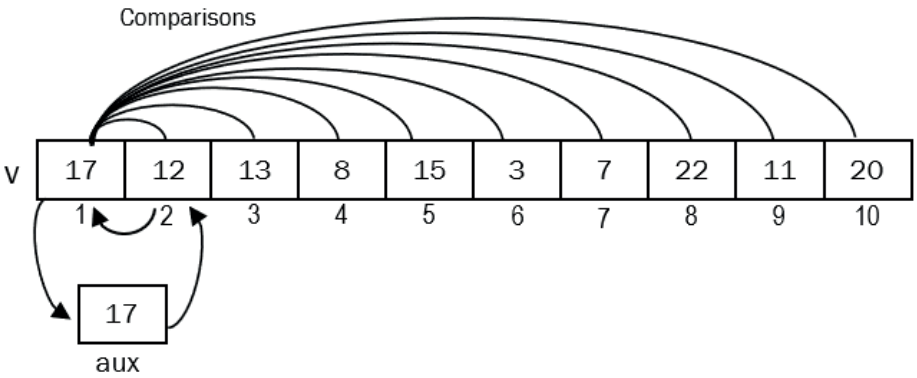
Baase and Van Gelder (2002: 150) mention that sorting data was one of the main concerns of computer science, proof of which is the number of sorting methods that have been designed. This chapter explains the most commonly used algorithms for this purpose.

7.2.1 Swap Algorithm

This algorithm is not the most efficient, but it is highly educational; therefore, it is commonly used in introductory programming courses. The method involves taking each element and comparing it with those to its right. Whenever a pair of elements that do not meet the applied sorting criterion is identified, they are swapped. The element that corresponds to a specific position in each iteration is found.

Consider vector v in Figure 116. To sort the numbers in ascending order, the first element *v[1]* is taken and compared with *v[2].* If *v[1]* is less than *v[2]* they are in order; however, if *v[1] > v[2]* they are out of order and must be swapped.

Figure 116. *Exchange Algorithm Comparisons of the First Element*



As shown in the figure, to swap two elements in a vector, it is necessary to use an auxiliary variable and perform three assignments:

aux = v[1]
v[1] = v[2]
v[2] = aux

After this swap, *v[1]* is compared with *v[3]*. If they are not sorted, the swap is conducted, followed by comparing *v[1]* with *v[4]*, and so on, until *v[1]* is compared with *v[n]*. More detailed explanations of this process can be found in Joyanes (2000: 248) and Chaves (2004: 235).

The instructions shown in Table 108 are executed to compare the first element of the vector with all the subsequent ones and position the corresponding value in the first spot of the sorted vector.

After executing the loop and finding the value corresponding to the first position of the sorted vector, the same process is repeated for the second position, then the third, and so on until the penultimate one. Since each position requires a pass through the elements on the right, it is necessary to use two nested loops.

Table 108. *Sorting by Swap Locating the First Element*

```
1    For j = 1 to n do
2        If v[1] > v[j] then
3            aux = v[1]
4            v[1] = v[j]
5            v[j] = aux
6        End if
7    End for
```

In summary, the swap method consists of implementing a traversal for the vector where, for each element, a second loop is made, in which it is compared with all the subsequent elements. When the elements are not sorted, a swap is made. Each iteration of the outer loop sorts one position of the vector. Table 109 shows a function for sorting an integer vector using this method. The function receives an unsorted integer vector and its size as parameters, applies the sorting algorithm, and returns the sorted vector.

The first loop starts at the first element and ends at the penultimate one. The second loop starts with the element following the one to be sorted (determined by the variable of the first loop) and ends at the last position

Table 109. *Function to Sort a Vector sing the Swap Method*

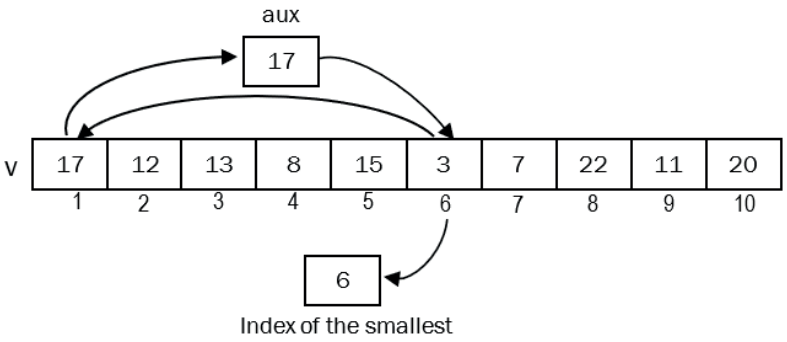
```
1   Integer[] swap(integer v[], Integer n)
2       Integer i, j, aux
3       For i = 1 to n-1 do
4           For j = i+1 to n do
5               If v[i] > v[j] then
6                   aux = v[i]
7                   v[i] = v[j]
8                   v[j] = aux
9               End if
10          End for
11      End for
12      Return v[]
13  End swap
```

7.2.2 Selection Sort Algorithm

This method is similar to the previous one in terms of traversing a vector and comparisons but with fewer swaps. In the swapping method, each time two positions are compared, and these are not sorted, the data is swapped so that in the same traversal, there can be several swaps before the number is placed in its correct position in the sorted vector. In the selection method, each traversal identifies the element that belongs to a particular position, and only one swap is made.

Consider the vector in Figure 117, where the swap made for the first position is shown.

Figure 117. *Selection Algorithm – Exchange of the First Element*



A variable is declared to store the position of the smallest element and is initialized to 1 to start comparisons in the first position. A traversal is made from the second position to the last, and each time an element smaller than the one indicated by the variable is found, the variable is updated.

Being at position ***i*** of vector ***v***, the traversal to identify the element corresponding to that position in the ascending sorted vector is carried out with the instructions shown in Table 110.

Table 110. *Traversal to Select the i-th Element*

```
1    possmallest = i
2    For j = i+1 to n do
3        If v[possmallest] > v[j] then
4            possmallest = j
5        End if
6    End for
7    Aux = v[i]
8    v[i] = v[possmallest]
9    v[possmallest] = aux
```

As shown in the pseudocode in Table 110, the traversal places the index of the smallest element found in the ***possmallest*** variable, and at the end of the traversal, the swap is made between position ***i*** and the position indicated by ***possmallest***.

To sort the entire vector, the variable ***i*** needs to move from the first element to the penultimate one, as shown in Table 111, which presents the function for sorting a vector using the selection algorithm. This function receives an unsorted vector as a parameter and returns it sorted.

Table 111. *Function to Sort a Vector Using the Selection Method*

```
1    Integer[] selection(integer v[], integer n)
2        Integer: i, j, possmallest, aux
3        For i = 1 to n-1 do
4        possmallest = i
5            For j = i+1 to n do
6                If v[possmallest] > v[j] then
7                    possmallest = j
```

```
8                   End if
9               End for
10              aux = v[i]
11              v[i] = v[possmallest]
```

Table 111. *(Continued)*

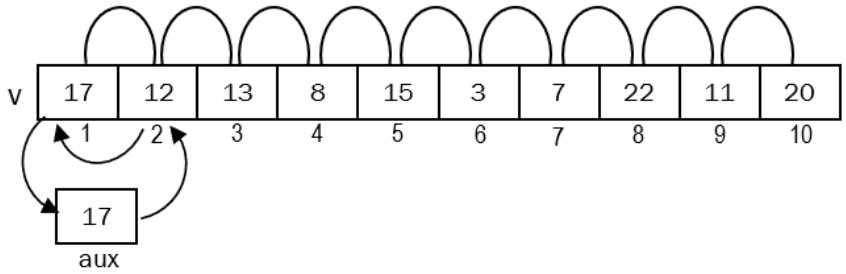
```
1               v[possmallest] = aux
2           End for
3           Return v[]
4   End selection
```

7.2.3 Bubble Sort Algorithm

Due to the way the traversals and comparisons are made, this is one of the easiest sorting algorithms to understand and program. However, as explained by Joyanes (2000: 252), its use is not recommended in software development because of its low efficiency.

The sorting technique known as bubble or bubble sort consists of comparing the first element with the second, and if they do not meet the sorting criterion being applied, they are swapped. Then, the second element is compared with the third, and if they are not sorted, they are also swapped. This process continues by comparing the third element with the fourth, then the fourth with the fifth, and so on, until comparing elements *v[n-1]* with *v[n]*. As shown in the vector in Figure 118, *n-1* comparisons and all necessary swaps are made in a traversal.

Figure 118. *Bubble Sort Algorithm – Comparisons of the First Traversal*



In the first traversal of the vector, since each element is compared with the adjacent one, if an ascending order is applied, the highest value will progressively move to the right, reaching the last position. However, the other values will remain unsorted in the remaining positions. The instructions for a single iteration are shown in Table 112.

Table 112. *Traversal to Bubble Up an Element*

```
1    For j = 1 to n-1 do
2        If v[j] > v[j+1] then
3            aux = v[j]
4            v[j] = v[j+1]
5            v[j+1] = aux
6        End if
7    End for
```

To completely sort the vector, considering that in a comparison, two elements are sorted, and in an iteration, an element is placed to its sorted position, it is necessary to make ***n-1*** iterations.

To improve the performance of this algorithm, the number of comparisons and, in some cases, the number of iterations can be reduced.

Since each iteration takes the largest data and moves it to the right, the first iteration places the largest data in position ***n***, the second iteration places the second largest data in position ***n-1***, and the third iteration places the next largest data in position ***n-1***. From this, it follows that in the second iteration, the comparison ***v[n-1] > v[n]*** is unnecessary, as the largest element is already in the last position. Similarly, in the third iteration, the comparisons ***v[n-2] > v[n-1]*** and ***v[n-1] > v[n]*** are redundant, as the final positions are sorted in each iteration. In conclusion, not all iterations need to reach ***n-1***; the second iteration to ***n-2***, the third to ***n-3***, and so on. In a vector with many elements, the reduction in the number of comparisons is significant. Table 113 shows the improved traversal for the i-th iteration

Table 113. *Improved Iteration to Bubbe Up the i-th Element*

```
1    For j = i to n-i do
2        If v[j] > v[j+1] then
3            aux = v[j]
4            v[j] = v[j+1]
```


```
5                v[j+1] = aux
6            End if
7        End for
```

On the other hand, a vector may be completely sorted in fewer iterations than ***n-1***, making the last iterations unnecessary. To avoid this inefficiency, it is monitored that in each iteration occur swaps. If no swaps occur during the i-th iteration, it indicates that the vector is already sorted, and it is unnecessary to continue with the remaining iterations. For this purpose, a swap counter or a switch (flag) that changes the state when a swap is made can be used. Table 114 shows the function for sorting a vector using the bubble sort algorithm with the two improvements discussed.

7.2.4 Insertion Sort Algorithm

This method involves progressively sorting the elements of the array, starting from the first elements and continuing until all are sorted. Given the element at position ***i*** (where ***i*** starts at 2 and progresses to the end of the array), it is stored in an auxiliary variable, and this variable is compared with the element to the left. If the elements are not sorted, the element on the left is shifted to the right. The auxiliary variable is compared again with the next element to the left. Every element that is not sorted is shifted one position to the right until an element that is sorted is found, or the beginning of the vector is reached. Once this traversal is completed, the element contained in the auxiliary variable is placed in the available position.

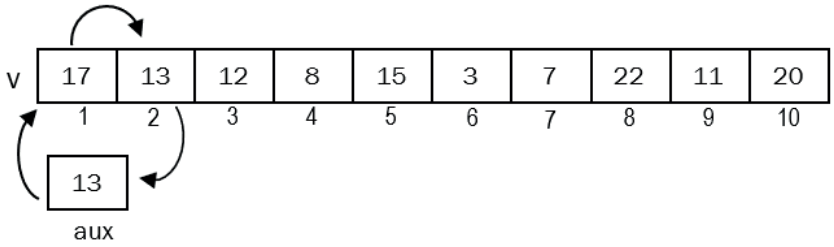
Table 114. *Function to Sort a Vector Using the Bubble Sort Method*

```
1       Integer[] bubblesort(integer v[], integer n)
2           Integer i = 1, j, aux
            Logical: change = true
3           While i < n-1 and change = true do
4               change = false
5               For j = 1 to n-i do
6                   If v[j] > v[j+1] then
7                       aux = v[j]
8                       v[j] = v[j+1]
9                       v[j+1] = aux
10                      change = true
11                  End if
12              End for
13              i = i + 1
```

```
14          End while
15          Return v[]
16      End bubblesort
```

Consider the vector shown in Figure 119. To sort it in ascending order, the content of the second position is copied to the auxiliary variable. It is then compared with the first element; since they are not sorted, the value in position 1 is shifted to position 2. As there are no more elements to the left, the content of the auxiliary variable is placed in position 1. In this way, the first two positions of the vector are already sorted: *v[1] = 13* and *v[2] = 17*.

Figure 119. *Insertion Sort Algorithm – Exchange for the Second Element*



Now, the element from the third position is taken, copied to the auxiliary variable, and compared with the element on the left. Since they are not sorted, the content from position two is shifted to position three. It is then compared with the data in position one, and as this is not sorted, it shifts to the right. Since there are no more elements on the left, the content of the auxiliary variable is placed in position one.

In summary, the process involves taking the i-th element and placing it in the correct position in the sorted vector. To achieve this, the space it occupies is freed by copying it to an auxiliary variable, and the elements greater than it are shifted to the right, creating a space in the position where it should be. The instructions in Table 115 move the i-th element to its corresponding position in the sorted vector.

To sort the entire vector, it is needed to repeat the instructions in Table 115 for each element, starting from the second position to the last position, as shown in Table 116.

Table 115. *Instructions for Inserting the i-th Element Into the Correct Position*

```
1       j = i – 1
2       aux = v[i]
3       While v[j] > aux and j > = 1 do
```

```
4           v[j+1] = v[j]
5           j = j -1
6       End while
7       v[j+1] = aux
```

Table 116. *Function to Sort a Vector Using the Insertion Method*

```
1   Integer[] insertion(integer v[], integer n)
2           Integer: aux, i, j
3           For i = 2 to n do
4               j = i – 1
5               aux = v[i]
6               While v[j] > aux and j >= 1 do
7                   v[j+1] = v[j]
8                   j = j -1
9               End while
10              v[j+1] = aux
11          End for
12          Return v[]
13  End Insertion
```

This algorithm can improve its efficiency by changing the search method applied to the left part of the vector. Since the left sub-vector of each element is sorted, binary search can be used instead of sequential search. This reduces the time the algorithm takes to find the place that corresponds to the element. This reduction in search time can be significant for vectors with many elements. Hernández et al (2001: 53) describe this change in the following steps:

a. Take an element in position i
b. Perform a binary search for its position among the previous positions
c. Shift the remaining elements to the right
d. Insert the element

7.2.5 Donald Shell's Algorithm

It is an insertion algorithm with decreasing jumps designed by Donald Shell[14] and generally recognized by the name of its creator. It operates similarly to the insertion

14 Donald Shell worked in the engineering division of ***General Electric*** and received his Ph.D. in mathematics from the University of Cincinnati in 1959. In the same year, he published the algorithm that now bears his name under the title ***"A High-Speed Sorting Procedure"*** in ***Communications of the ACM.***

algorithm, but instead of moving elements by one position at a time, it shifts elements to several positions at once, allowing them to reach their correct placement more efficiently. This method is particularly suitable for vectors with large amounts of data.

In this algorithm, an element is taken and compared with those to its left. If an ascending order is sought, the elements greater than the reference are shifted to the right, and this is placed in its corresponding position. However, instead of comparing it with the element immediately to the left, it is compared with the one that is *x* positions behind. The distance *x* is called the "gap." In the first iteration, the gap is half the size of the vector so that elements from the left half are compared with the elements from the right half, and each swap moves elements across both sides.

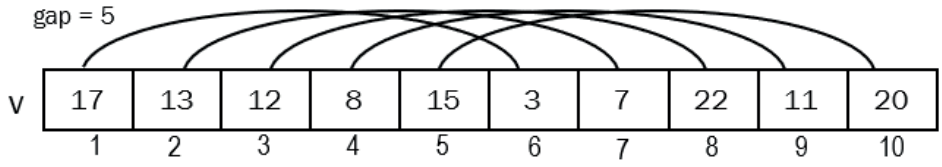
In the second iteration, the gap is halved, and for the third one, this process continues, halving the gap each time until comparisons are made one by one.

To illustrate how this algorithm works, consider a 10-element vector. Then:

Gap = 10/2 = 5

In this case, during the first iteration, comparisons are made with a gap of 5 positions, meaning *v[1]* is compared with *v[6]*, *v[2]* with *v[7]*, and so on, as shown in Figure 120.

Figure 120. *Shell Sort Algorithm – First Iteration*



In the first iteration, each element on the left side is compared with one on the right side as the gap divides the vector into two parts.

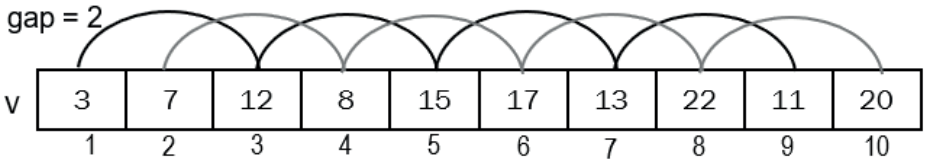
Comparison	Result	Swap
V[1] > v[6]	True	Yes
V[2] > v[7]	True	Yes
V[3] > v[8]	False	No
V[4] > v[9]	False	No
V[5] > v[10]	False	No

For the second iteration, the gap is updated, taking as a value the fourth part of the vector's size, which implies that some elements will involve multiple gaps to the left.

gap = gap/2
gap = 5/2 = 2

Figure 121 shows the vector after the first iteration and the comparisons to be made in the second iteration.

Figure 121. *Shell Sort Algorithm – First Iteration*



In the second iteration of the analyzed example, the gap = 2, consequently the loop runs from the third element to the last one. The variable ***i*** is initialized at 1, and the variable ***aux*** holds the third element (12). The first comparison is between ***v[1] and v[3] (12 < 3 = false).*** If the elements are in order, the variable ***follow*** takes the false value, the loop ***while*** ends and the iterations then proceed with the next element (*v[4]*) and so on.

Table 117 shows the pseudocode for any iteration. In this algorithm, each element is compared with those on its left. The ***while*** loop manages the traversal towards the left, provided that there are elements and the data intended to be inserted is smaller than the one indicated by index ***i***. If the element in position ***i*** is smaller than ***j***, it is not necessary to continue towards the left since those elements are already sorted.

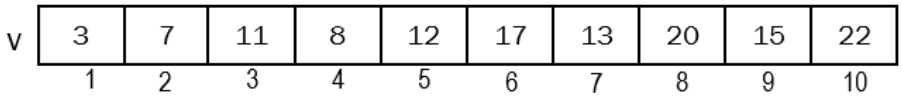
Table 117. *Shell Algorithm – One Iteration*

```
1    For j = (1 + gap) to n do
2        i = j – gap
3        aux = v[j]
4        continue = true
5        While i > 0 and continue = true do
6            If aux < v[i] then
7                v[i + gap] = v[i]
```

```
8                   i = i – gap
9               Else
10                  continue = false
11              End if
12          End while
13          v[i + gap] = aux
14      End for
```

Figure 122 shows the order of the elements after the second iteration with two-by-two gaps.

Figure 122. *Shell Sort Algorithm – First Iteration*



In this example, because a small vector (10 elements) was used, the gaps were made one by one by the third iteration, and then the vector was completely sorted. Arrays with larger numbers of elements require more iterations.

Table 118 presents a function for sorting a vector by applying the Shell algorithm.

Table 118. *Function to Sort a Vector Using the Shell Method*

```
1   Integer[] shell(integer v[], integer n)
2       Integer: aux, gap, i, j
        Logical: continue
3       gap = n/2
4       While gap >= 1 do
5           For j = (1 + gap) to n do
6               i = j – gap
7               aux = v[j]
8               continue = true
9               While i > 0 and continue = true do
10                  If aux < v[i] then
11                      v[i + gap] = v[i]
12                      i = i – gap
13                  Else
14                      continue = false
```

```
15                      End if
16                  End while
17                  v[i + gap] = aux
18              End for
19              Gap = gap/2
20          End while
21          Return v[]
22  End Shell
```

7.2.6 Quick Sort Algorithm

This algorithm is considered the fastest among those studied in this chapter. It was designed by Tony Hoare[15] and is based on the ***divide-and-conquer*** technique, which suggests that sorting two small lists is faster and easier than sorting a large one (Joyanes, 2000: 405).

The principle of the ***quicksort*** algorithm is the partition of the vector or list into three smaller parts distributed to the left, center, and right. The center contains the value used as a reference for partitioning the vector and, thus, a single element.
In general terms, the quicksort algorithm consists of the following operations:

1. Select a reference element called the pivot
2. Traverse the array from left to right, searching for elements larger than the pivot
3. Traverse the array from right to left, searching for elements smaller than the pivot
4. Swap each element smaller than the pivot with one larger than the pivot, so that the smaller values are on the left, larger values are on the right, and the pivot is in the center.
5. Repeat these operations with each sublist until there are sublists of one element.
6. Rebuild the list by concatenating the sublist on the left, the pivot, and the sublist on the right.

Although this algorithm can be implemented iteratively, it is much easier to understand and implement using recursion, a topic that is explained in the next chapter. The pseudocode for partitioning the vector is shown in Table 119.

15 His full name is Charles Antony Richard Hoare. He studied at Oxford University and Moscow State University. He developed the Algol 60 language and designed the Quicksort Algorithm. He worked at Queen's University and the Computing Laboratory at Oxford University.

The recursive version of this algorithm is explained in detail in section 8.7

Table 119. *Function to Partition a Vector*

```
1   Integer[] partition(integer v[], low integer, high integer)
2       Integer: aux, pivot = v[low], m = low, n = high+1
3       While m <= n do
4           Do
5               m = m + 1
6           While v[m] < pivot
7           Do
8               n = n – 1
9           While v[n] > pivot
10          If m < n then
11              Aux = v[m]
12              v[m] = v[n]
13              v[n] = aux
14          End if
15      End while
16      v[low] = v[n]
17      v[n] = pivot
18      Recursive call
19      Return v[]
20  End partition
```

7.2.7 Merging of Sorted Vectors

This operation is also known as interleaving (Joyanes, 1996: 343) and merging (López et al., 2009: 135). It involves combining the elements of two sorted vectors or lists and forming a third vector or list that is also sorted, where the previous order of the entries is used to obtain a sorted set in less time.

Figure 123 shows two vectors of different sizes and their merger into a third sorted vector. The algorithm for merging the two sorted vectors consists of a loop that traverses the two input vectors simultaneously and selects the smallest element in each iteration to add to the new vector. In the array from which the element is taken, the index is incremented, while in the other array, it remains constant until an element is transferred to the new vector. The loop ends when one of the vectors has

been fully copied, and the remaining elements from the other vector are added in their current order. Table 120 presents the function to perform this task.

Figure 123. *Merging Sorted Vectors*

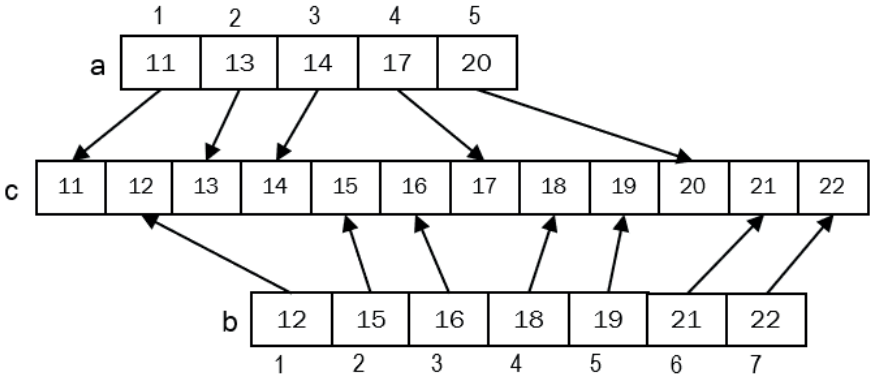


Table 120. *Function to Merge Sorted Vectors*

```
1   Integer[] merge(integer a[], integer b[], integer m, integer n)
2       Integer: c[m+n], i = 1, j = 1, k = 1, x
3       While i <= m and j <= n do
4           If a[i] <= b[j] then
5               c[k] = a[i]
6               i = i + 1
7           Else
8               c[k] = b[j]
9               j = j + 1
10          End if
11          k = k + 1
12      End while
13      If i <= m then
14          For x = i to m do
15              c[k] = a[x]
16              k = k + 1
17          End for
18      End if
19      If j <= n then
20          For x = j to n do
```

```
21                  c[k] = b[x]
22                  k = k + 1
23              End for
24          End if
25          Return c[]
26      End merge
```

7.2.8 Other Sorting Algorithms

In addition to the six sorting algorithms presented in this section, there are many more, some more efficient than those studied here and, therefore, more complex, requiring more advanced knowledge in computer science. Some are mentioned below for readers who want to delve into this topic.

Mergesort: This is a recursive algorithm that applies the ***divide and conquer*** principle similarly to Quicksort. It consists of partitioning the vector or list into two equal parts, recursively sorting each of them, and then merging them through the process known as merging sorted sequences or merging sorted vectors.

Heapsort: This algorithm combines the advantages of Quicksort and Mergesort. It uses a data structure called a heap, which is a complete binary tree from which some extreme leaves are removed.

Radix Sort: This method involves distributing the elements of the set to be sorted into several subsets according to a certain criterion; each subset is sorted using any sorting method and then merging them while preserving the order. For example, suppose there is a large set of invitation cards that need to be organized alphabetically by the name of the recipient. One can form 26 subsets based on the starting letter of the name, then sort each heap using, for instance, the insertion method, and finally, merge the groups starting with the one with the letter A.

7.2.9 A Complete Example

Example 72. Grade List

A teacher manages the list of students and grades using arrays: in an integer vector, he/she stores the codes, in a string array, the names; and in a four-column array keeps the grades: three partial grades and the final grade.

Among the operations the professor performs are: entering the student names, entering the grades as they are generated, calculating the final grade, and listing the grades; sometimes he/she checks a student's grades, and sometimes he/she needs to correct a grade. The list is sorted according to one of these criteria: alphabetical order by last name or descending order by final grade. Design the algorithm to perform the mentioned tasks.

According to the problem description, the data structure used is shown in Figure 124.

Figure 124. *Arrays to Store Grade Sheets*

Vector Codes		Vector Names		Array Grades				
					1	2	3	4
1	101	1	Salazar Carlos	1	3.5	2.5	4.0	3.3
2	102	2	Bolaños Carmen	2	4.2	3.8	4.5	4.2
3	102	3	Bermúdez Margarita	3	2.0	4.5	4.8	3.8
4	104	4	Paz Sebastián	4	2.5	3.5	4.4	3.5
5	105	5	Díaz Juan Carlos	5	3.4	3.5	3.8	3.6
6	106	6	Marroquín Verónica	6	4.0	4.6	3.3	4.0
…	…	…	…	…	…	…	…	…
33	133	33	Hernández María	33	3.2	4.0	4.6	3.9
34	134	34	Ordóñez Miguel	34	4.5	3.8	3.0	3.8
35	-1	35		35				
…	-1	…		…				
					Partial grade 1	Partial grade 2	Partial grade 3	Final grade

In this exercise, a brief description of the requirements that the application to be designed must meet is given. Up to this point, it is already known what to do; now, it must be thought about how to do it. The following paragraphs will address that.

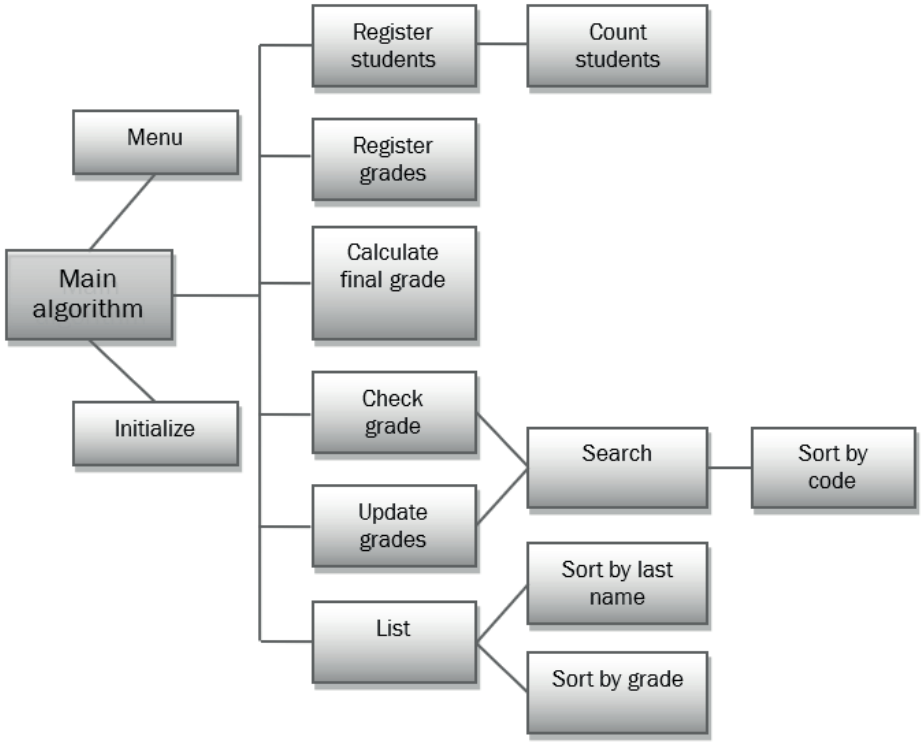
This problem is very complex to attempt solving it with a single algorithm; it is necessary to divide it into modules or subroutines that can be easily designed. How should the tasks be separated?

One way to do this is based on the tasks the user needs the program to perform, which are technically known as requirements. These include: entering the students, which requires a procedure or a function to input these data; after grading the

exams, the professor needs to record the grades, so a procedure is required to enter grades; another procedure to check a student's grades, and another to modify them if necessary; finally, a procedure to generate the lists is also required, and since the lists are presented in order, a procedure is needed to sort the data in the arrays.
An important point to consider is that the algorithms that have been studied operate on a certain amount of data. In this exercise, the number of students in the course is unknown. The strategy to solve cases in which the amount of space required is unknown is to analyze the problem domain and declare the arrays considering the case where more space is required. In the case of a grade sheet, it is likely that there are no more than 40 students, but it is possible that, exceptionally, there could be more than 50. To avoid anyone is left unrecorded, it is better to consider the largest space required and declare the arrays with that size. So, in this example, space for 65 students is reserved.

Figure 125 presents a diagram with the procedures and functions to be designed to solve this exercise.

Figure 125. *Diagram of Procedures and Functions*



However, since sufficient space is reserved for a very large group, in most cases, the course will be smaller, and part of the arrays will be empty. This implies that to ensure the efficiency of the algorithms, the arrays should only be traversed up to the point where there is data available. The strategy applied in this case is to initialize the vector of codes with -1 in each position so that when entering students this value is replaced with the student's code. To know how many records are present, a function is designed to count the elements with a value different from -1. Table 121 shows the procedure to initialize the code vector, and Table 122 shows the function to determine the number of registered students.

Table 121. Function to Initialize the Vector

```
1    initialize()
2        Integer: i
3        For i = 1 to 65 do
4            code[i] = -1
5        End for
6    End initialize
```

In this procedure, as well as in the function that follows, parameters are not defined, this indicates that the vector they access is defined as global in the main algorithm.

Table 122. *Function to Count Students*

```
1    Integer countstudents()
2        Integer: i=0
3        While code[i+1] != -1 do
4            i = i + 1
5        End while
6        Return i
7    End countstudents
```

Procedures and functions are designed to implement the application's requirements. Table 123 shows the procedure for registering student names.

Table 123. *Function to Register Students*

```
1    registerStudents()
2        Integer: i
3        Character: ans
```

```
4           i = countStudents()
5           Do
6               i = i + 1
7               Write “Code: “
8               Read code[i]
9               Write “Name: “
10              Read name[i]
11              Write “More students (y/n)? “
12              Read ans
13          While ans = ‘Y’ or ans = ‘y’
14  End registerStudents
```

In the registerStudents procedure, the function ***countStudents()*** is used, which allows new students to be registered at the end of the list. Table 124 shows the procedure for recording grades. This procedure receives as a parameter the partial exam to which the grade belongs (first, second, or third), displays the student's name, reads the grade, and saves it in the corresponding column.

Once the three grades have been recorded, the final grade is calculated. The procedure for calculating the final grade is presented in Table 125.

Table 124. *Procedure to Record Grades*

```
1   registerGrades (Integer: j)
2           Integer: i,n
3           n = countStudents()
4           For i = 1 to n do
5               Write “Code: “, code[i]
6               Write “Name: “, name[i]
7               Write “Grade “, j, “:”
8               Read grades[i][j]
9           End for
10  End registerGrades
```

Table 125. *Procedure to Calculate Final Grade*

```
1   calculateFinalGrade ()
2           Integer: i,n
```


```
3           n = countStudents()
4           For i = 1 to n do
5               Grades[i][4] = (grades[i][1] + grades[i][2] + grades[i][3])/3
6           End for
7   End calculateFinalGrade
```

Once the data has been entered, three operations can be performed: querying, modifying, and reporting. For all three operations, it is advisable to have the data sorted. Before proceeding to write the procedure to query a grade, a procedure to sort the vectors by student code using the selection method is designed. This will allow binary search to be applied in the query and modification operations. This is presented in Table 126.

Table 126. *Procedure to Sort Vector Using Selection Method*

```
1   sortByCode()
        Integer i, j, posMin, aux, n
2       Real: v[4]
        String: name
3       n = countStudents()
4       For i = 1 to n-1 do
5           posMin = i
6           For j = i+1 to n do
7               If code[posMin] > code[j] then
8                   posMin = j
9               End if
10          End for
11          aux = code[i]
12          code[i] = code[posMin]
13          code[posMin] = aux
14          name = name[i]
15          name[i] = name[posMin]
16          name[posMin] = name
17          V[1] = grades[i][1]
18          V[2] = grades[i][2]
19          V[3] = grades[i][3]
20          V[4] = grades[i][4]
21          grades[i][1] = grades[posMin][1]
22          grades[i][2] = grades[posMin][2]
23          grades[i][3] = grades[posMin][3]
24          grades[i][4] = grades[posMin][4]
```


```
25                grades[posMin][1] = v[1]
26                grades[posMin][2] = v[2]
27                grades[posMin][3] = v[3]
28                grades[posMin][4] = v[4]
29            End for
30    End sortByCode
```

In this algorithm, between lines 11 and 28 data swapping occurs in both vectors and the matrix. A vector of 4 elements is used to move the row of the matrix.

To query a student's grades, it is necessary to perform a search based on their code. Considering that the sorting procedure has already been developed, binary search can be applied to the code vector. Table 127 shows the algorithm for the binary search function, adapted to the logic being applied in this exercise. This function returns the position corresponding to the student, allowing access to the other arrays using the index.

Table 127. *Function for Binary Search for Student Code*

```
1     binarySearch(Integer code)
2         Integer inf, sup, center, pos = -1
3         Inf = 1
4         sup = countStudents()
5         sortByCode()
6         While inf <= sup and pos < 0 do
7             center = (inf + sup) / 2
8             If v[center] = code then
9                 pos = center
10            Else
11                If code > v[center] then
12                    inf = center + 1
13                Else
14                    sup = center – 1
15                End if
16            End if
17        End while
18        Return pos
19    End binarySearch
```

The binary search algorithm is explained in detail in Section 7.1.2.

Thus, the procedure to query a student's grades only has to perform three tasks: read the code, search for it, and access the data. This procedure is shown in Table 128.

Table 128. *Procedure to Query Grades*

```
1   query()
2       Integer: code, pos
3       Write "Student code: "
4       Read code
5       pos = binarySearch(code)
6       If pos > 0 then
7           Write "Code:", code[pos]
8           Write "Name:", name[pos]
9           Write "Grade 1:", grades[pos][1]
10          Write "Grade 2:", grades[pos][2]
11          Write "Grade 3:", grades[pos][3]
12          Write "Final grade:", grades[pos][4]
13      Else
14              Write "Code ", code, "not found"
15      End if
16  End query
```

At times, the teacher will need to change a grade. For this, the "modify" procedure is provided. This procedure performs the following operations: it reads the student's code, searches for it, and if found, displays the data, reads the new value, and saves it in the corresponding column. The pseudocode for this procedure is shown in Table 129.

Table 129. *Procedure to Modify Grades*

```
1   modify()
2       Integer: code, pos, option
3       Write "Student code: "
4       Read code
5       pos = binarysearch(cod)
6       If pos > 0 then
7           Write "Code:", code[pos]
```

```
8               Write “Name:”, name[pos]
9               Write “Grade 1:”, grades[pos][1]
10              Write “Grade 2:”, grades[pos][2]
11              Write “Grade 3:”, grades[pos][3]
12              Write “Final grade:”, grades[pos][4]
13              Write “Enter the grade to modify (1, 2 or 3): ”
14              Read option
15              According to option do
16                  1: read grades[pos][1]
17                  2: read grades[pos][2]
18                  3: read grades[pos][1]
19              End according
20              Grades[pos][4] = (Grades[pos][1]+ Grades[pos][2]+ Grades[pos][3])/3
21          Else
22                  Write “Code “, code, “not found”
23          End if
24      End modify
```

Next, the subalgorithms to generate the grade report are designed. This report can be sorted alphabetically by name or in descending order by final grade, with the purpose of displaying the highest grades first. When sorting the data, it is important to ensure that the swaps are made across all three arrays. Table 130 presents the pseudocode for organizing the data alphabetically using the direct swap algorithm.

Table 130. *Procedure to Sort Data Alphabetically Using Direct Swap*

```
1       sortByLastName()
            Integer i, j, aux,n, code
2           String: name
            Real v[4]
3           n = countStudents()
4           For i = 1 to n-1 do
5               For j = i+1 to n do
6                   If name[i] > name[j] then
7                       code = code[i]
8                       code[i] = code[j]
9                       code[j] = code
10                      name = name[i]
11                      name[i] = name[j]
12                      name[j] = name
13                      V[1] = grades[i][1]
```

```
14                      V[2] = grades[i][2]
15                      V[3] = grades[i][3]
16                      V[4] = grades[i][4]
17                      grades[i][1] = grades[j][1]
18                      grades[i][2] = grades[j][2]
19                      grades[i][3] = grades[j][3]
20                      grades[i][4] = grades[j][4]
21                      grades[j][1] = v[1]
22                      grades[j][2] = v[2]
23                      grades[j][3] = v[3]
24                      grades[j][4] = v[4]
25                  End if
26              End for
27          End for
28  End sortByLastName
```

To sort the grade sheet by final grade, the Shell sort algorithm is adapted, as shown in Table 131.

Table 131. *Procedure to Sort Data in Descending Order Using Shell Sort Algorithm*

```
1   sortByGrade()
2       Integer: aux, gap, i , j, code, n
        Boolean: keepGoing
        String: name
        Real: v[4]
3       n = countStudents()
4       gap = n/2
5       While gap >= 1 do
6           For j = (1 + gap) to n do
7               i = j – gap
8               code = code[j]
9               name = name[j]
10              V[1] = grades[j][1]
11              V[2] = grades[j][2]
12              V[3] = grades[j][3]
13              V[4] = grades[j][4]
14              keepGoing = true
15              While i > 0 and keepGoing = true do
16                  If V[4] < grades[i][4] then
17                      code[i + gap] = code[i]
```

```
18                          name[i + gap] = name[i]
19                          grades[i + gap][1] = grades[i][1]
20                          grades[i + gap][2] = grades[i][2]
21                          grades[i + gap][3] = grades[i][3]
22                          grades[i + gap][4] = grades[i][4]
23                          i = i – gap
24                      Else
25                          keepGoing = false
26                      End if
27                  End while
28                  code[i + gap] = code
29                  name[i + gap] = name
30                  grades[i + gap][1] = v[1]
31                  grades[i + gap][2] = v[2]
32                  grades[i + gap][3] = v[3]
33                  grades[i + gap][4] = v[4]
34              End for
35              gap = gap/2
36          End while
37      End sortByGrade
```

If an ascending order were desired, it would be sufficient to change the ***less-than*** sign to a *greater-than* sign in line 16.

In the procedure *list(),* presented in Table 132, the user is prompted to select the desired order for the list. Then, one of the two previously designed sorting procedures is invoked, and subsequently, the listing is generated.

Table 132. *Procedure to List Students and Grades*

```
1   List ()
2       Integer: order, i, n
3       n = countStudents()
4       Write “Generate sorted list by:”
5       Write “1. Name  2. Final Grade”
6       Read order
7       If order = 1 then
8           sortByLastName()
9       Else
10          sortByGrade()
```

```
11      End if
12      Write “Code \t Name \t Grade1 \t Grade2 \t Grade3 \t Final Grade”
13      For i = 1 to n do
13          Write code[i], “\t”, name[i], “\t”, grade[i][1], “\t”, grade[i][2], “\t”,
                grade[i][3], “\t”, grade[i][4]
14      End for
15  End list
```

Continuing with the solution algorithm to the proposed problem, there are already several procedures and some functions to carry out specific tasks. Now, a function that interacts with the user and reports to the main algorithm the task that the user wants to perform is required. For this purpose, a menu and a function that displays it and captures the selection are designed. The pseudocode for this is shown in Table 133.

Table 133. *Menu Function*

```
1   Integer menu()
2       Integer: option
3       Write “Menu”
4       Write “1. Register Students”
5       Write “2. Register Grades”
6       Write “3. Calculate Final Grade”
7       Write “4. Query Student Grades”
8       Write “5. Update Student Grades”
9       Write “6. Print Report”
10      Write “7. Exit”
11      Do
12          Read option
13      While option < 1 or option > 7
14      Return option
15  End menu
```

Finally, Table 134 presents the main algorithm, which controls the execution of all the procedures and functions designed previously.

Table 134. *Main Algorithm*

```
1   Begin
2       Global Integer: code[65]
        Integer option, nGrade
        Global String: name[65]
        Global Real: grades[65][4]
3       Initialize()
4       Do
5           option = menu()
6           According to option do
7               1: registerStudents()
8               2: Write "Which grade would you like to enter? (1, 2 or 3)?"
9                   Read nGrade
10                  registerGrades(nGrade)
11              3: calculateFinalGrade()
12              4: query()
13              5: modify()
14              6: list()
15          End according
16      While option != 7
17  End main algorithm
```

Thus concludes this small exercise in designing a program to solve a low-complexity problem such as managing the grades of a course. However, it has allowed for the application of all the concepts studied throughout the previous chapters, including array handling, searching, and sorting data. Therefore, it can be used as a reference for developing the exercises proposed next.

7.2.10 Proposed Exercises

1. The Interesting Readings library wants to manage its materials through a computer program and has requested the design of an algorithm for this purpose. The goal is to store data about books, such as subject, title, publisher, author, and year. The program should allow for the entry of all books and enable the addition of new acquisitions as they are made. It should also support queries based on author, title, or ISBN, as well as generate sorted listings by author or title.

2. The company Well-Paid wants an algorithm to process its payroll information. The aim is to store data in arrays, including employee name, position, base salary, days worked, deductions, and monthly salary. Tasks to be performed include data entry, data modification, payroll processing, inquiries, and report generation.

3. An algorithm that uses matrices to maintain the data of university professors, including name, address, and phone number, and allows for adding data, modifying existing entries, listing in alphabetical order, and searching by name.

8. RECURSION

The judgment of wise men reveals the clear through the obscure, the large through the small, the remote through the near, and the reality through ppearance.
—Séneca [16]

Recursion, also known as recursive processes, is a concept that is especially applied in mathematics and computer programming, and similarly in numerous everyday situations. In mathematics, the term ***inductive definition*** refers to the many methods that use recursion. In programming, this concept applies to algorithms during specification and to functions during implementation (Joyanes, 1999).

The introduction of recursion into programming languages is attributed to Professor John McCarthy from the Massachusetts Institute of Technology (MIT), who advocated for its inclusion in the design of Algol 60 and developed the Lisp language, which introduced recursive data structures along with recursive procedures and functions (Baase, 2002).

Recursion serves as an alternative to iterative structures. A recursive algorithm performs calculations repeatedly by making consecutive calls to itself. This does not mean that recursive algorithms are more efficient than iterative ones; in some cases, programs may even require more time and memory to execute. However, this cost can be offset by a more intuitive solution that is mathematically grounded in a proof by induction.

16 [Translation of the original epigraph in Spanish]

Most current programming languages allow for the implementation of recursive algorithms through procedures or functions. In cases where languages do not support direct implementation, recursion can be simulated using stack-like data structures.

Recursion is a concept that applies interchangeably to algorithms, procedures, and programs. However, this book addresses it from the perspective of algorithm design.

8.1 RECURSION AND ALGORITHM DESIGN

In the field of algorithm design, recursion is defined as the technique of solving a problem by consecutively invoking the same algorithm with a progressively less complex problem, until reaching a version with a known solution. For example, consider calculating the factorial of a number.

Example 73. Recursive Function to Calculate the Factorial of a Number

The factorial of a number is defined as the product of all integers from 1 to ***n,*** inclusive, and is represented as ***n!***

Thus:

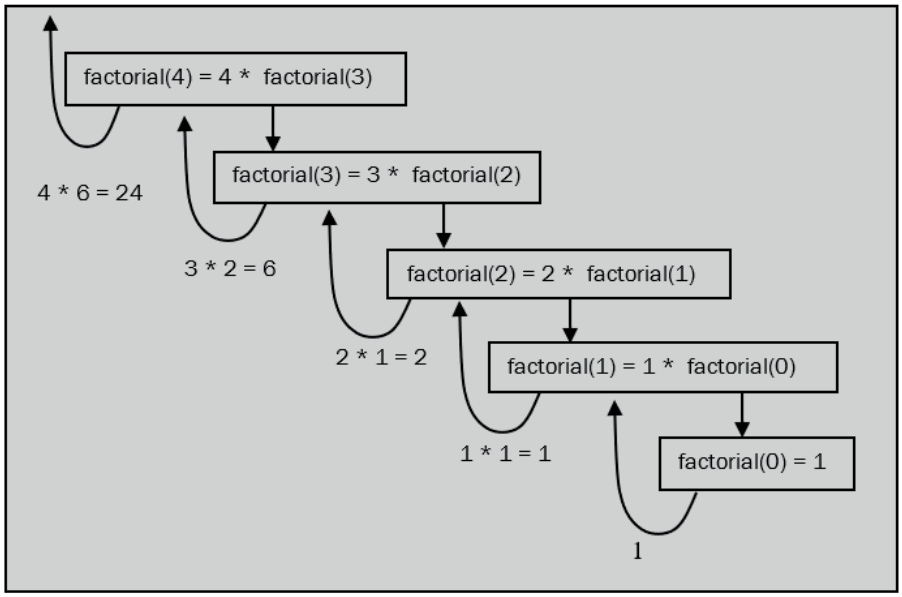
$$n! = 1 * 2 * 3 * 4 * \ldots * (n-2) * (n - 1) * n$$

It is also known that the factorial is defined for the case when ***n = 0***, with the result being ***1***. In other words, ***0! = 1***.

From this, we see that to calculate the factorial of 4 (4!), we multiply 4 by the factorial of 3 (3!); to calculate the factorial of 3 (3!), we multiply 3 by the factorial of 2 (2!); to calculate the factorial of 2 (2!), we multiply 2 by the factorial of 1 (1!); and to calculate the factorial of 1 (1!), we multiply 1 by the factorial of 0. Now, since we know that the factorial of 0 is 1 (0! = 1), the problem is solved. This exercise will be fully developed later.

As illustrated in Figure 126, recursion involves writing a function that includes a call to itself within its lines or statements, using different values for its parameters, progressively leading to the final solution.

Figure 126. *Diagram of a Recursive Function*



Recursion is another way to execute a part of the code repeatedly without implementing iterative structures such as ***while***, ***for***, or ***do while.***

8.2 STRUCTURE OF A RECURSIVE FUNCTION

As mentioned, recursion involves implementing functions that call themselves. This implies that the function must have a structure that allows it to invoke itself and execute as many times as necessary to solve the problem, while also avoiding more invocations than strictly necessary to prevent an infinite sequence of self-calls. When a recursive function specifies in its definition when to self-invoke and when to stop doing so, it is said to be correctly defined (Lipschutz, 1993).

A well-defined recursive function must meet the following two conditions:

1. There must be a *base case* whose solution is known and does not involve a recursive call. In the case of using a recursive function to calculate the factorial, the *base case* indicates that the factorial of 0 is 1 (0! = 1). That is, if the function receives 0 as a parameter, it no longer needs to invoke itself again; it simply returns the corresponding value, in this case, 1.

factorial(n) = ? (requires a recursive call)
factorial(0) = 1 (does not require a call)

2. Each time the function invokes itself, either directly or indirectly, it must do so with a value closer to the *base case*. Since the base case represents a particular instance of the problem with a known solution, this means that with each call to the recursive function, the parameters should be approaching the known solution. Continuing with the factorial function example, since the base case is *0! = 1,* each time the function is invoked, it should use a parameter value closer to 0.

The factorial of a number is the product of that number and the factorial of the next smaller number, continuing until reaching the factorial of 0, which is constant. Based on this, it can be stated that the factorial is defined for any positive integer as follows:

0! = 1	
1! = 1 * 1 = 1	1! = 1 * 0!
2! = 2 * 1 = 2	2! = 2 * 1!
3! = 3 * 2 * 1 = 6	3! = 3 * 2!
4! = 4 * 3 * 2 * 1 = 24	4! = 4 * 3!
5! = 5 * 4 * 3 * 2 * 1 = 120	5! = 5 * 4!

Therefore, the solution for the factorial can be expressed as:

a. If n = 0 → n! = 1
*b. If n > 0 → n! = n * (n – 1)!*

This is the correct definition of a recursive function, where a is the *base case*, that is, the known solution, and b is the recursive invocation with an argument closer to the solution. Expressed in the form of a mathematical function, we have:

Let *f(n)* be the function to calculate the factorial of ***n***, then:

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

The pseudocode for this function is shown in Table 135.

Table 135. *Recursive Function to Calculate the Factorial of a Number*

```
1    Integer factorial(Integer n)
2        If n = 0 then
3            Return 1
4        Else
```

```
5                Return (n * factorial(n – 1))
6            End if
7        End factorial
```

In the definition, the fulfillment of the two conditions discussed in this section is clearly evident. Lines 2 and 3 examine the fulfillment of the base case criterion, for which the solution is known (0! = 1).

2. *If n = 0 then*
3. *Return 1*

If the base case has not yet been reached, it is necessary to invoke the recursive function again, but with a smaller value, as shown in lines 4 and 5.

4. *Else*
5. *Return (n * factorial(n – 1))*

8.3 EXECUTION OF RECURSIVE FUNCTIONS

The execution of a recursive function requires dynamically allocating enough memory to store its data. For each execution, space is reserved for parameters, local variables, temporary variables, and the return value of the function. The code runs from the beginning with the new data. It is important to clarify that the recursive code is not duplicated in each activation frame. (Schildt, 1993).

The space in which each invocation of a recursive function executes is called an ***Activation Frame*** (Baase, 2002). This frame not only provides space to store the variables that the function operates with, but also allocates space for other bookkeeping needs, such as the return address, which indicates the instruction to execute once it exits the recursive function. Thus, an activation frame is created in which the function executes only during a single invocation.

Since the function will be invoked an indeterminate number of times (depending on the parameter values), and a new activation frame is created for each invocation, the compiler must allocate a region of memory for the creation of the ***stack of frames***. This space is referenced by a register called the ***Frame Pointer***, so that while a function invocation is executing, the locations of local variables, input parameters, and return values are known.

A manual execution that shows the states of the activation frames is called an ***Activation Trace*** (Baase, 2002) and allows for analyzing the execution time of the function and understanding how recursion really works in the computer.

Each active invocation has a unique activation frame. A function invocation is active from the moment it is entered until it is exited, after solving the problem passed as a parameter. All simultaneous active invocations of the recursive function have distinct activation frames. When exiting a function invocation, its activation frame is automatically freed so that other invocations can use that space, and execution resumes at the point where the function was invoked.

To better illustrate these ideas, figure 127 presents the activation trace for the factorial function.

Figure 127. *Activation Trace for the Factorial Function*

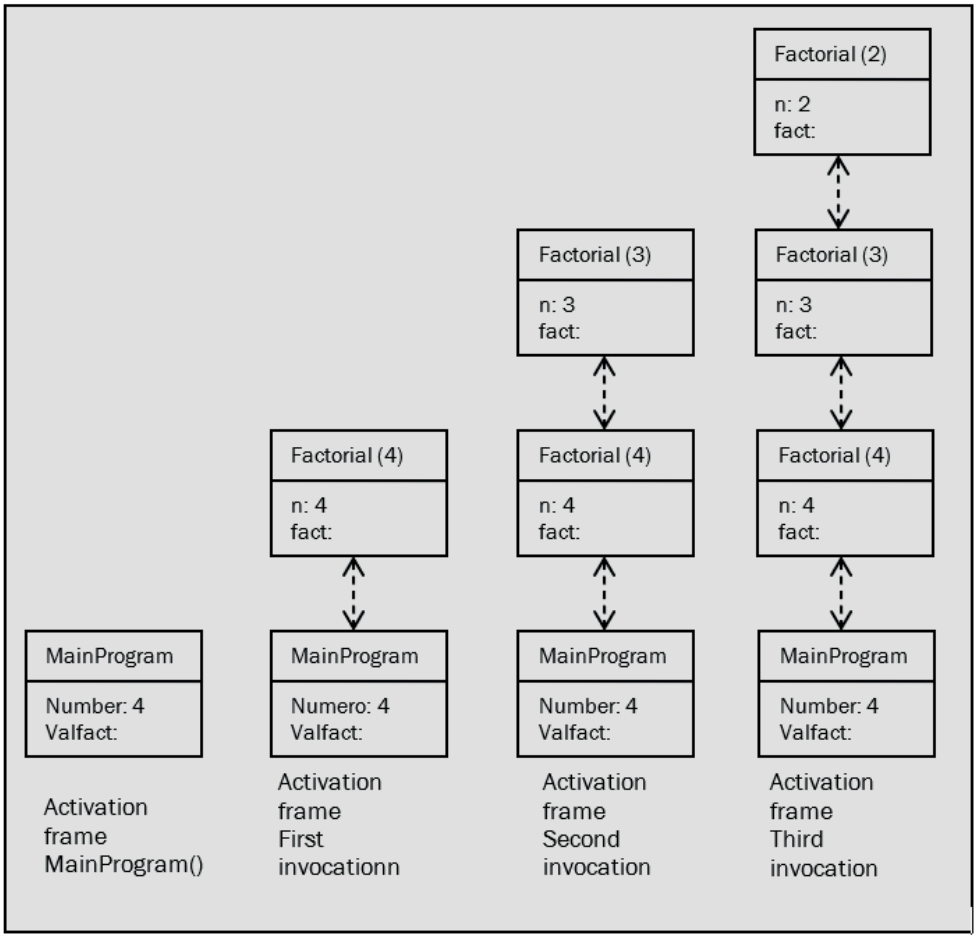
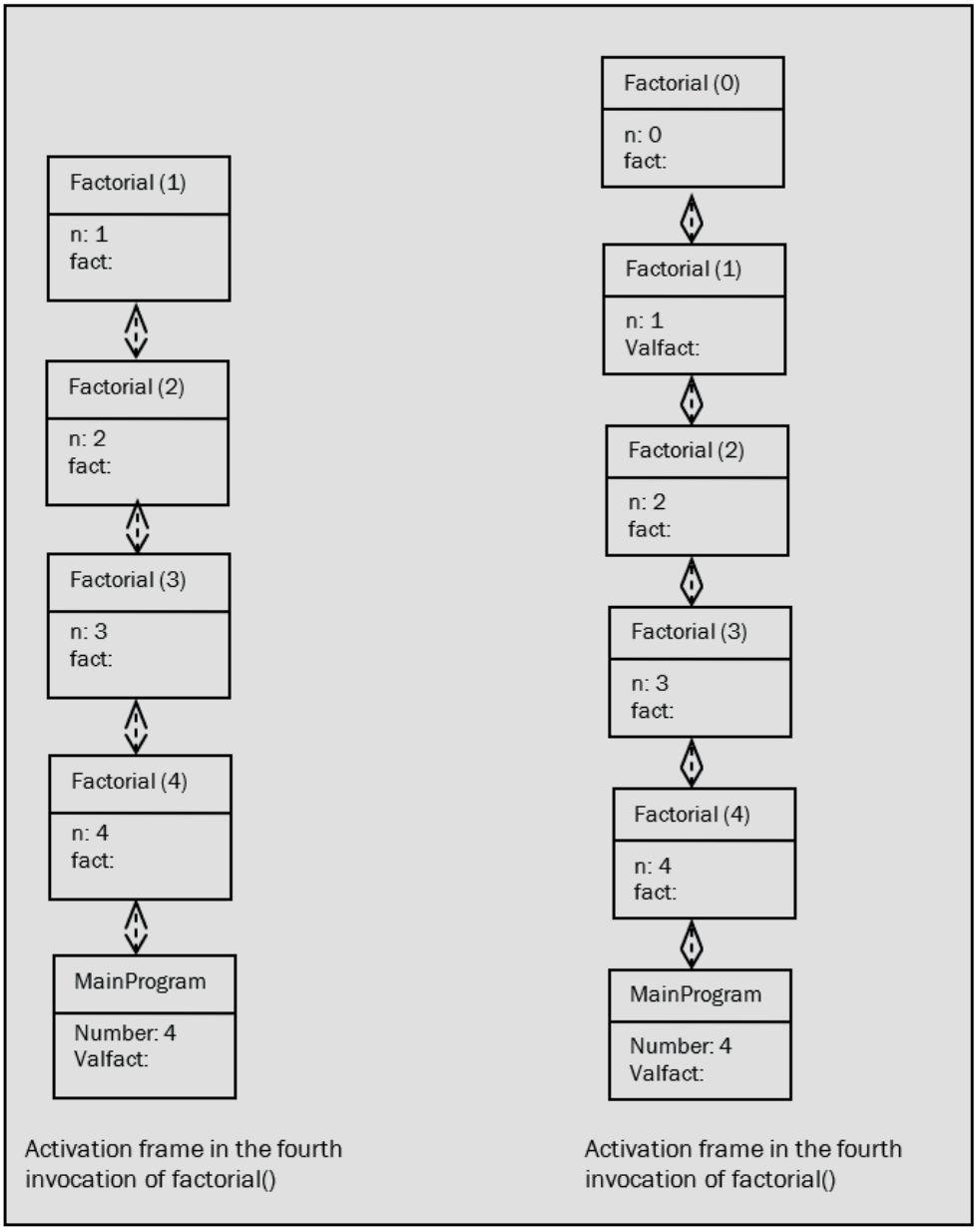


Figure 127. *(Continued)*



Once the function has found the base case, that is, a case with a non-recursive solution, this frame returns the determined value for that criterion to the caller and frees the occupied space; in this case, the last frame returns 1 to the previous frame and disappears. It continues a backward process in which each activation frame performs its operations using the value returned by the invoked recursive function and returns the obtained value to the invoking frame.

8.4 WRAPPERS FOR RECURSIVE FUNCTIONS

It is common for recursive functions to focus on finding a value or performing a specific task, such as calculating a factorial, reversing a string, finding a value in a data structure, or sorting a list. They often do not handle other tasks that do not require recursion, such as reading the data to search for, checking the validity of an argument, or printing results.

In other words, when programming with recursion, you will find some tasks that do not need recursive handling and can occur before or after the execution of the recursive function. To manage these tasks, ***wrapper*** programs, procedures, or functions are defined.

A ***wrapper*** refers to non-recursive procedures that execute before or after the invocation of a recursive function, responsible for preparing the function's parameters and processing the results (Baase, 2002).

In the factorial example, a wrapper algorithm is used to read the number that serves as the function's argument, invoke the function, and display the result. The pseudocode is presented in Table 136.

Table 136. *Pseudocode of the Main Program to Calculate the Factorial*

```
1    Begin
2        Integer: num
3        Read  num
4        Print "factorial of", num, " = ", factorial(num)
5    End algorithm
```

8.5 TYPES OF RECURSION

Recursion can manifest in different ways, and based on these, at least four different types of recursive implementations have been established.

Simple Recursion: This occurs when a function calls itself with a different argument. An example of this type of recursion is the factorial() function.

This type is characterized by being easily converted into an iterative solution.

Multiple Recursion: The body of a function includes more than one call to the same function. For example, the function to calculate a Fibonacci series value (explained in section 8.7).

Nested Recursion: A recursive function is considered nested when one of the parameters passed to the function includes a call to itself. An example of nested recursion is the solution to the Ackermann[17] problem.

Cross or Indirect Recursion: In this type of recursion, the body of the function does not call itself but another function, which includes a call to the first. More than two functions can participate in this type of implementation, also known as ***recursive chains***. An example is the function for validating a mathematical expression.

8.6 EFFICIENCY OF RECURSION

In general, an iterative version will run more efficiently in terms of time and space than a recursive version. This is because, in the iterative version, the overhead associated with entering and exiting a function is avoided. In the recursive version, it is often necessary to stack and unstack anonymous variables in each activation record. This is not necessary for an iterative implementation where results are replaced in the same memory spaces.

On the other hand, recursion is the most natural way to solve some types of problems, such as QuickSort, the Towers of Hanoi, the eight queens problem, conver-

17 Wilhelm Ackermann (March 29, 1896 – December 24, 1962) was a German mathematician who conceived the doubly recursive function that bears his name as an example of computational theory and demonstrated that it is not primitive recursive.

ting prefix to postfix, or tree traversal. Although iterative solutions can be implemented, the recursive solution arises directly from the problem's definition.

Choosing between recursive or iterative methods is essentially a conflict between machine efficiency and programmer efficiency (Langsam, 1997). Considering that the cost of programming tends to increase while the cost of computation decreases, it may not be worthwhile for a programmer to spend time and effort developing a complicated iterative solution for a problem that has a simple recursive solution. Conversely, it is also not worth implementing recursive solutions for problems that can be easily solved iteratively. Many examples of recursive solutions in this chapter are presented for educational purposes rather than for their efficiency.

It is important to note that the extra time and space demand in recursive solutions primarily comes from creating function activation records and stacking partial results, and these can often be optimized by reducing the use of local variables. At the same time, iterative solutions that use stacks can require as much time and space as recursive ones.

8.7 EXAMPLES OF RECURSIVE SOLUTIONS

Example 74. Recursive Summatory

Design a recursive function to calculate the summation of the first ***n*** positive integers, in the form:

$$1 + 2 + 3 + 4 + 5 + 6 + \ldots + (n-1) + n$$

We define the function ***f(n)***, where ***n*** can be any number greater than or equal to 1 (***n >= 1***).

It is known that adding 0 to any number leaves it unchanged. Therefore, the summation of zero is zero. For any other positive integer, the summation is calculated by adding its own value to the summation of the immediately preceding number. Thus:

$$f(0) = 0$$
$$f(1) = 1 + f(0)$$
$$f(2) = 2 + f(1)$$
$$f(3) = 3 + f(2)$$

f(4) = 4 + f(3)
f(n) = n + f(n-1)

From this, we conclude that:

a. If $n = 0 \rightarrow f(n) = 0$
b. If $n > 0 \rightarrow f(n) = n + f(n-1)$

This establishes a recursive solution where ***a*** represents the base case and ***b*** the recursive invocation of the function.

$$f(n) = \begin{cases} 0 & \text{If } n = 0 \\ n + f(n-1) & \text{If } n > 0 \end{cases}$$

The pseudocode for the summatory function is presented in Table 137.

Table 137. *Recursive Function to Calculate the Summation of a Number*

```
1    Integer summatory(integer n)
2        If n = 0 then
3            Return 0
4        Else
5            Return  (n + summation(n – 1))
6        End if
7    End summation
```

Example 75. Recursive Exponentiation

The calculation of a power is defined as solving an operation of the form x^n, where ***x*** is an integer or real number known as the base, and ***n*** is a non-negative integer known as the exponent.

The power x^n is the product of ***n*** times ***x***, in the form:

*xn = x * x * x * … * x (n times x)*

According to the properties of exponentiation, any number raised to the power of 0 equals 1, and any number raised to the power of 1 is the same number.

$x0 = 1$
$x1 = x$

For this exercise, we extend the first property also to the case of x = 0. Although it is typically stated that this result is undefined, for the purpose of explaining recursion, this is irrelevant.
To propose a recursive solution, it is necessary to express x^n in terms of a closer identity: $x^0 = 1$.

We propose the following example:

$2^0 = 1$	
$2^1 = 2$	$2^1 = 2 * 2^0 = 2$
$2^2 = 4$	$2^2 = 2 * 2^1 = 2 * 2 = 4$
$2^3 = 8$	$2^3 = 2 * 2^2 = 2 * 4 = 8$
$2^4 = 16$	$2^4 = 2 * 2^3 = 2 * 8 = 16$
$2^5 = 32$	$2^5 = 2 * 2^4 = 2 * 16 = 32$

As seen in the right column, each power can be formulated using the solution of the previous power, with a smaller exponent.

Consequently:

a. If $n = 0 \rightarrow x^n = 1$
b. If $n > 0 \rightarrow x^n = x * x^{n-1}$

Where *a* is the base case and *b* is the recursive invocation.

Table 138 presents the pseudocode to recursively calculate any power of any number.

Table 138. *Recursive Function to Calculate a Power*

```
1    Integer power(integer x, integer n)
2        If(n = 0)
3            Return 1
4        Else
5            Return (x * power(x, n-1))
6        End if
7    End power
```

In lines 2 and 3, the problem is solved when the base case is encountered; otherwise, a recursive invocation of the function must be made, as per lines 4 and 5.

Example 76. Fibonacci Series

The Fibonacci series has many applications in computer science, mathematics, and game theory. It was first published in 1202 by an Italian mathematician of the same name in his book titled ***Liber Abaci*** (Brassard, 1997).

Fibonacci posed the problem as follows: suppose a pair of rabbits produces two offspring each month, and each new pair begins reproducing after two months. Thus, when purchasing a pair of rabbits, in the first and second months there will be one pair, but by the third month they will reproduce, resulting in two pairs. In the fourth month, only the first pair reproduces, so the number increases to three pairs; in the fifth month, the second pair begins reproducing, leading to five pairs, and so on.

If no rabbits die, the number of rabbits each month is given by the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …, which corresponds to the relationship shown in Table 139:

Table 139. *Terms of the Fibonacci Series*

Month (*n*)	0	1	2	3	4	5	6	7	8	9	…
Pairs of rabbits *f(n)*	0	1	1	2	3	5	8	13	21	34	…

Its formal representation is:

a. $f(0) = 0$
b. $f(1) = 1$
c. $f(n) = f(n-1) + f(n-2)$

That is, in month 0 there are still no pairs of rabbits; in month 1, the first pair is acquired; in month 2, the same pair is retained, as they have not yet begun to reproduce. However, starting from month 3, the number of pairs must be calculated considering the condition that each pair generates a new pair every month, but only starts reproducing after two months.

Thus, the problem is solved for month 0 and month 1, which are taken as the base cases (a, b). From the second month onward, the previous months must be taken into account. This creates a recursive invocation in the form:

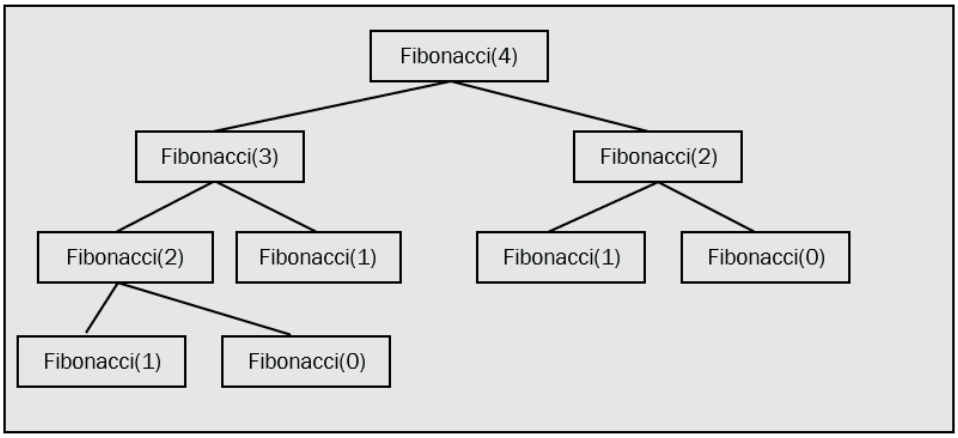
$f(n) = f(n-1) + f(n-2)$ for $n >= 2$.

In the pseudocode presented in Table 140, it can be seen that each activation defines two base cases (line 2), and if these have not yet been reached, two new recursive invocations are generated in each activation frame (line 5). Figure 128 illustrates the tree that forms when calculating the value of the series for the number 4.

Table 140. *Recursive Function to Calculate the n-th Term of the Fibonacci Series*

```
1    Integer Fibonacci(integer n)
2        If (n = 0) or (n = 1) then
3            Return n
4        Else
5            Return (Fibonacci(n – 1) + Fibonacci(n – 2))
6        End if
7    End Fibonacci
```

Figure 128. *Activation Frames of the Fibonacci Series*



Example 77. Greatest Common Divisor

The Greatest Common Divisor (GCD) of two or more numbers is the largest of their common divisors. There are two methods for finding the GCD of two numbers: the first method involves breaking each number down into its prime factors, expressing the numbers in exponential form, and then taking the common factors with the smallest exponent; the product of these factors will be the GCD. The second method is a recursive search where it is checked if the second number is a divisor of the first. If it is, then that number is the GCD. If it is not, the second number is divided by the modulus of the first with respect to the second, and this process continues with progressively smaller numbers, tending towards 1 as the common divisor for all numbers. This latter method is known as the Euclidean algorithm.

For example, to find the GCD of 24 and 18 using the Euclidean algorithm, we divide the first number by the second (24/18 = 1, remainder = 6). If the division is exact, the GCD is the second number; if not, as in this case, we continue searching for the GCD between the second number (18) and the remainder (6), and continue this process until we find the GCD. If the two numbers are prime, the result will eventually reach 1.

If ***f(a,b)*** is the function that returns the GCD of ***a*** and ***b***, then it can be expressed as:

$c = a \bmod b$

a. *If* $c = 0 \rightarrow f(a,b) = b$
b. *If* $c > 0 \rightarrow f(a,b) = f(b,c)$

The recursive function for calculating the GCD of two numbers using the Euclidean algorithm is presented in Table 141.

Table 141. *Recursive Function to Calculate the GCD*

```
1    Integer gcd(Integer a, Integer b)
2        Integer c
3        c = a Mod b
4        If c = 0 then
5            Return b
6        Else
7            Return gcd(b, c)
8        End if
9    End gcd
```

Example 78. Recursive Quick Sort Algorithm

This recursive algorithm for sorting arrays or lists is based on the ***divide-and-conquer*** principle, which, in this case, translates to the idea that it's easier to sort two small arrays than one large one. The algorithm involves selecting an element from the array as a reference, known as the pivot, and partitioning the array into three parts: the elements less than the pivot, the pivot itself, and the elements greater than the pivot.

To partition the array, two indices are declared, and two passes are made: one starting from the first element and moving forward, and another starting from the last element and moving backward. The first index advances until it finds an element greater than the pivot, while the second index moves backward until it finds an element less than the pivot. When the two indices meet, a swap is performed, and the passes continue. The partitioning stops when the two indices cross. For example, consider the array shown in Figure 112 and apply the instructions in Table 142.

Figure 129. *Passes for Partitioning an Array*

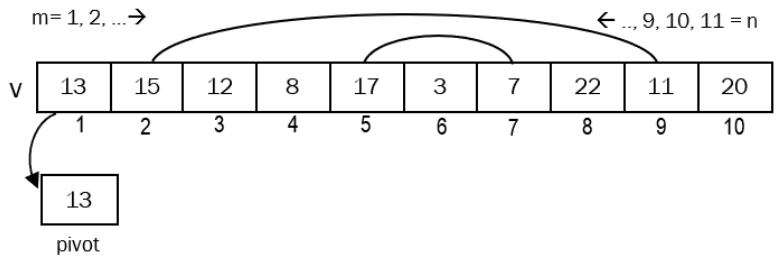
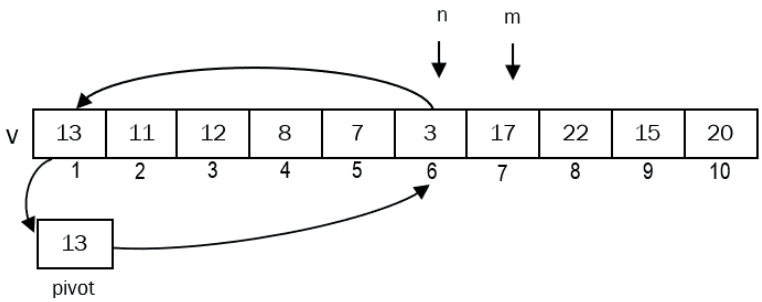


Table 142. *Passes for Partitioning an Array*

```
1     Integer m = 1, n = 11, pivot = v[1]
2     While m <= n do
3         Do
4             m = m + 1
5         While v[m] < pivot
6         Do
7             n = n – 1
8         While v[n] > pivot
9         If m < n then
10            aux = v[m]
11            v[m] = v[n]
12            v[n] = aux
13        End if
14    End while
```

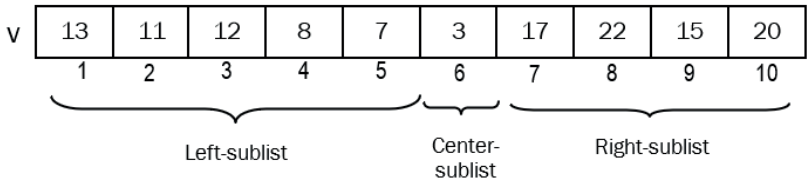
After executing the instructions in Table 142, the array will appear as shown in Figure 130.

Figure 130. *Passes for Partitioning an Array*



After moving the elements greater than the pivot to the right side and the lesser elements to the left, the pivot is swapped with the element indicated by the index that traverses the array from right to left (n), as illustrated by the arrows in Figure 130. After this swap, the elements are in the order shown in Figure 131.

Figure 131. *Order of the Array after the First Partition*



Although the divisions are purely formal, we now have three sublists:

left-list = {3, 11, 12, 8, 7}
center-list = {13}
right-list = {17, 22, 15, 20}

As seen in the sublists, the data has moved from one side of the pivot to the other, but they are not yet sorted. Therefore, the next step in the ***Quicksort*** algorithm is to take each of the sublists and perform the same process, continuing to partition the generated sublists until their size is reduced to a single element.

Table 143 presents the recursive ***Quicksort*** function with the complete algorithm to sort an array using this method.

Table 143. *Recursive Quicksort Function*

```
1   Integer[] Quicksort(integer v[], integer inf, integer sup)
2       Integer: aux, pivot = v[inf], m = inf, n = sup+1
3       While m <= n do
4           Do
5               m = m + 1
6           While v[m] < pivot
7           Do
8               n = n – 1
9           While v[n] > pivot
10          If m < n then
```

```
11                  Aux = v[m]
12                  v[m] = v[n]
13                  v[n] = aux
14              End if
15          End while
16          v[inf] = v[n]
17          v[n] = pivot
18          If inf < sup then
19              v[] = Quicksort(v[], inf, n-1)
20              v[] = Quicksort(v[], n+1,sup)
21          End if
22          Return v[]
23      End Quicksort
```

The ***Quicksort*** function receives an array of integers, in this example, and two values: ***inf*** and ***sup***, which correspond to the positions that define the subset of elements to be sorted. The first invocation of the function will include the entire array, sending the index of the first and last element (***1*** y ***n***) as parameters. However, in subsequent calls, the number of elements significantly reduces with each invocation, corresponding to the sublists generated during each partition.

In this version of the function, the first element is taken as the pivot, but this does not have to be the case; a function can be designed to find an element with a median value that allows for a balanced partition of the list.

8.8 PROPOSED EXERCISES

Design recursive solutions for the following requirements:

1. Display the numbers from 1 a n
2. Generate the multiplication table for a number
3. Sum the divisors of a number
4. Determine if a number is perfect

5. Calculate the product of two numbers without using the * operator, considering that multiplication consists of adding the multiplicand as many times as indicated by the multiplier.

6. Calculate the Least Common Multiple (LCM) of two numbers.

7. Convert a decimal number to binary

8. Convert a binary number to decimal

9. Determine if a string is a palindrome

10. Display the first n terms of the Fibonacci series

11. Reverse the order of the digits that make up a number

12. Determine if a number is prime.

9. THE RUBIK'S CUBE

If I am asleep, or if I am awake,
if I am a dead man dreaming he's alive,
or a living man dreaming he's dead.
Our dreams always come true
as long as we have patience...
Julio Flórez
[Translation of the original epigraph in Spanish]

The Rubik's Cube, commonly referred to as the magic cube, is a mechanical puzzle designed by Hungarian architect and sculptor Ernő Rubik, patented in 1975 in Hungary. This puzzle was initially used as a teaching tool in some Hungarian schools, as it was believed that the attention and mental effort required to solve it made for good mental exercise.

Since its early popularity, players have measured their skills by solving it in the shortest amount of time. The first world championship was held in Budapest in 1982.

In 1976, a similar invention to the Rubik's Cube was patented in Japan by engineer Teruthos Ishige.

9.1 DESCRIPTION OF THE CUBE

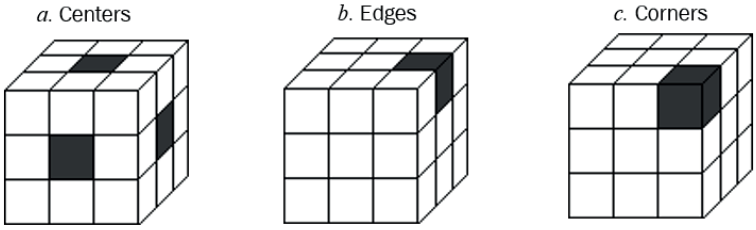
The Rubik's Cube is a puzzle made up of 26 pieces that move around a mechanical device located at the center of the cube, which is hidden from the user's view, as shown in Figure 132.

Many versions of the Rubik's Cube have been produced using numbers, letters, shapes, and colors. The most popular version features six colors, one for each face. The goal is to arrange the pieces so that each face of the cube displays a single color.

Figure 132. *Rubik's Cube*

Each of the 26 pieces that make up the cube appears to be a small cube itself; however, there are actually three different types: centers, edges, and corners, as shown in Figure 133.

Figure 133. *Pieces of the Rubik's Cube*



Centers have only one face, and the color of this piece serves as the reference point for solving the cube. These pieces are fixed to the internal device and thus do not change position; edges have two faces of different colors, while corners have three.

The positioning of the pieces is achieved through successive rotations of the layers that comprise the cube. To facilitate the understanding of the movements necessary to solve the cube, this document presents the layers illustrated in Figure 134.

Figure 134. *Layers of the Rubik's Cube*

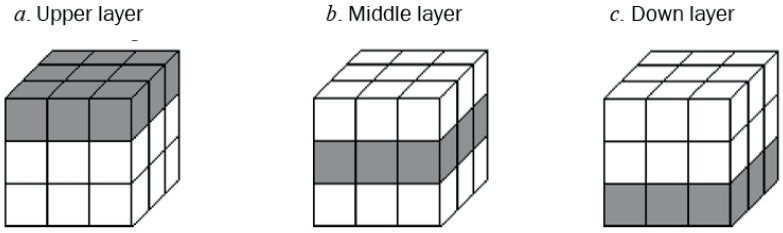
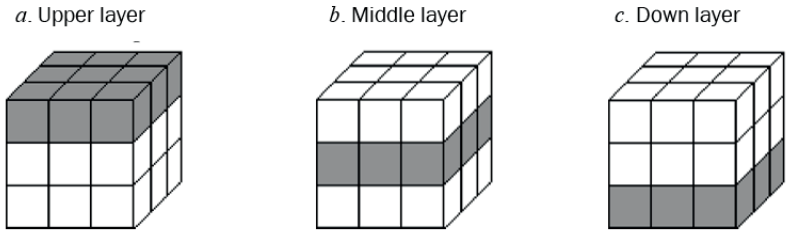


Figure 134. *(Continued)*



9.2 ALGORITHMIC SOLUTION

The proposed solution involves solving the cube in the following order:

1. Solve the upper layer
2. Solve the middle layer
3. Solve the down layer

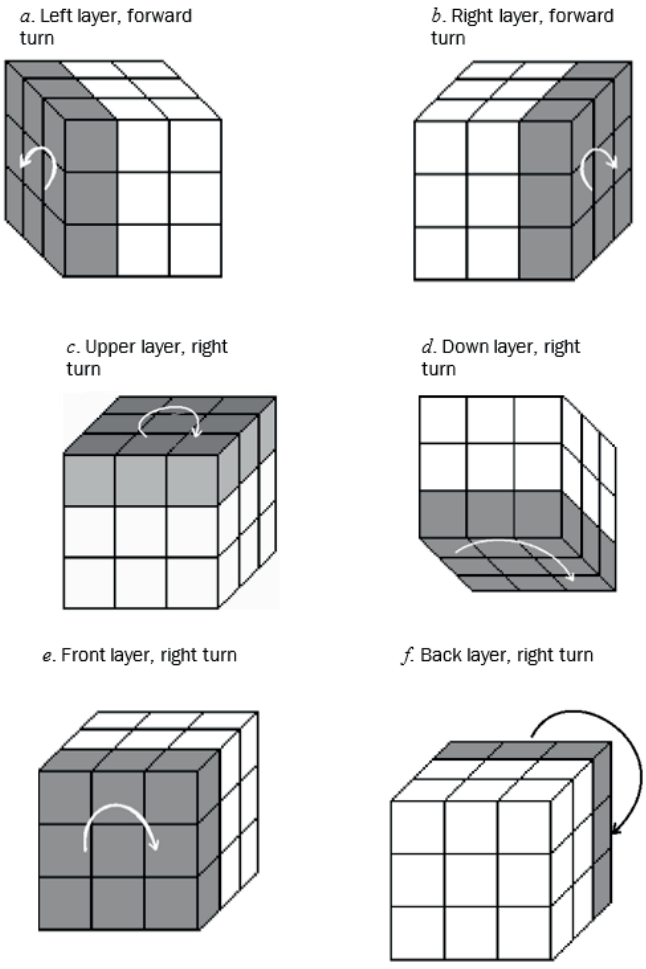
In both the upper and down layers, it is necessary to locate and arrange both the edges and corners, while in the middle layer, only the edges need to be arranged. The main difficulty lies in ensuring that each step does not undo what has been accomplished in the previous steps. Later in this chapter, detailed algorithms will be presented to achieve this goal; all that is required is attention, patience, and perseverance in applying them.

9.2.1 Preliminary Considerations

Before presenting the sequences that allow for solving the cube, it is essential to specify how the movements should be executed and the terminology used to describe them.

Each layer can perform two types of movements: the right and left layers can turn forward or backward; the upper, down, front, and back layers can turn right or left, as indicated in the cubes shown in Figure 135. The center layer does not turn unless the entire cube is rotated, in which case the pieces will not change position.

Figure 135. *Movements of the Rubik's Cube Layers*



Each indicated turn should be made by 90 degrees; that is, a corner will move to the position of the next one according to the type of turn. When a 180-degree turn is required, it will be written twice.

The algorithm for solving the cube is presented in a modular format; that is, it consists of several procedures that are called when needed. This facilitates understanding of the sequences, as each one changes the position of one or two pieces.

In the solution presented, the easiest-to-understand and apply sequences are used, rather than the shortest ones, so that the reader does not become discouraged. Once confidence in executing the algorithm is gained and the movement of the pie-

ces is understood, some modifications can be introduced to solve the cube with fewer moves.

To solve the cube, it is necessary to develop a combination of turns of the different layers that comprise it. Each turn is indicated by stating the name of the layer followed by the type of turn to be made. For example:

Up → right: indicates that the upper layer should be turned to the right.
Right → forward: means that the right layer should be turned forward.

9.2.2 Main Algorithm for Solving the Rubik's Cube

As mentioned on previous pages, the Rubik's Cube is a puzzle, so solving it is merely a game, a form of entertainment; however, given the number of pieces and the potential combinations of movements, learning to solve it through trial and error can take a considerable amount of time, and many people may become discouraged when they find that every time they try to place one piece in order, the previously ordered pieces get scrambled.

To facilitate this activity, the necessary sequences of movements for solving the cube are presented algorithmically. Since the list is extensive, a ***divide-and-conquer*** strategy is applied, and procedures are designed to solve each part of the cube. The main algorithm is presented in Table 144.

Table 144. *Main Algorithm for the Rubik's Cube*

1	Begin
2	Selecting and positioning the reference center
3	Solving the upper layer
4	Solving the middle layer
5	Solving the down layer
6	End of algorithm

Each of the steps in this algorithm is explained and refined in the following sections of this chapter.

9.2.3 Selecting and Positioning a Reference Center

To begin solving the cube, the first step is to select a reference color to avoid confusion when the cube rotates and changes position. Since the center pieces do not change position, they determine the color of each face, and one of them will serve as a reference throughout the entire process.

Following this logic, the first action is to choose one of the six colors and position the face with that color facing upward. This way, the upper layer is defined, allowing for the arrangement of its pieces in the next step.

For example, if white is chosen as the reference color, the layer with the white center piece is oriented upward. Whenever any of the sub-algorithms for solving any layer is executed, the central white piece must remain facing up.

The reference center is maintained throughout the entire process of solving the cube, but before starting any procedure, it is important to pay attention to the color of the front center to avoid mistakenly changing the cube's position.

9.2.4 Solving the Upper Layer

This layer is solved in two phases: first, the edges are arranged using the ***order_upper_edge()*** procedure, and then the corners are arranged by invoking the ***order_upper_corner()*** procedure. The main procedure is presented in Table 145.

Table 145. *Procedure for Solving the Upper Layer*

```
1     Solve_upper_layer()
2        Integer: edge, corner
3        // Loop to arrange the edges of the upper layer
4        For edge = 1 to 4 do
5           If edge is not ordered then
6                 Order_upper_edge()
7           End if
8           Rotate cube to the left
9        End for
10       // Loop to arrange the corners of the upper layer
11       For corner = 1 to 4 do
12          If corner is not ordered then
```

```
13                  Order_upper_corner()
14              End if
15              Rotate cube to the left
16          End for
17      End Solve_upper_layer
```

In this procedure, a loop examines each edge of the cube. If an edge is ordered, the next one is evaluated; if it is not ordered, the designed procedure to order upper edges is invoked. The same operation is then performed on the corners.

An edge is considered ordered if the color of each of its faces matches the color of the center piece on the corresponding face of the cube. For example, if the color of the center piece on the upper face is white and the color of the center piece on the front face is red, the upper front edge will be ordered if the upper color is white and the front color is red; otherwise, it is not ordered, and the *Order_ upper_edge()* procedure will need to be invoked. A corner is ordered if each of its faces corresponds with the color of the center piece on the respective face; otherwise, the *Order_ upper_corner()* procedure must be invoked.

Ordering the Edges of the Upper Layer

In this phase of the process, the edges of the upper layer are arranged such that the colors of each edge correspond with the reference color of the upper center and with the colors of the centers of each face of the cube according to their position.

The ***Order_upper_edge()*** procedure, presented in Table 146, is designed to place the correct piece in the front-upper position. This involves three fundamental operations:

1. Find the edge whose colors correspond to the upper center and the front center;
2. Move the edge from its current position to the *front-down* position, regardless of orientation.
3. Move the edge from the front-down position to the *front-upper* position and orient it.

Table 146. *Procedure for Ordering Upper Edges*

```
1   Order_upper_edge()
2       Identify the current position of the edge to be ordered
3       // Move edge to the front-down position
4       Depending on the current position do
5           Current position = Back layer: upper edge
6               Back →  right
7               Back → right
8               Down → right
9               Down → right
10          Current position = Right layer: back edge
11              Right → forward
12              Down → left
13              Right → back
14          Current position = Right layer: upper edge
15              Right → forward
16              Right → forward
17              Down → left
18          Current position = Right layer: front edge
19              Front → right
20              Current position = Left layer: back edge
21              Left → forward
22              Down → right
23              Left → back
24          Current position = Left layer: upper edge
25              Left → forward
26              Left→forward
27              Down→ right
28          Current position = Left layer: front edge
29              Front →left
30          Current position = Front layer: upper edge
31              Front → right
32              Front → right
33          Current position = Down layer: back edge
34              Down → left
35              Down → left
36          Current position = Down layer: right edge
```

```
37                  Down → left
38              Current position = Down layer: left edge
39                  Down → right
40          End switch
41          // move the piece from the front-down position to the front-upper
                position
42          If front_color of front-down edge =  reference color then
43              Down→ right
44              Right → forward
45              Front → left
46              Right → back
47          Else
48              Front →left
49              Front → left
50          End if
51      End order_upper_edge
```

Executing this procedure will order the ***upper-front*** edge. To order all the edges of the upper layer, it is necessary to execute it four times, as specified in the ***Solve_upper_layer()*** procedure.

After ordering the upper edges, the cube will appear as shown in Figure 136, where the dark gray part indicates the portion of the cube that has already been ordered.

Figure 136. *Upper Layer with Ordered Upper Edges*

Ordering the Corners of the Upper Layer

The corners are formed by three colors corresponding to the colors of the three faces that converge at that point. If the color of each side of the corner matches the color of the center piece on the face, then the corner is in the correct position. Otherwise, it must be moved to its correct corner and oriented according to the colors of the faces.

The movements indicated in the ***Order_upper_corner()*** procedure, presented in Table 147, allow for the ordering of the ***front–upper–right*** corner. This requires three actions:

1. Identify the current position of the corner to be ordered;
2. Move the piece to the *front–down–right* position. For this step, a multiple decision structure is used, which includes a sequence of movements for each of the seven alternatives;
3. Move the piece from the *front-down–right* corner to the front–upper –right position. In this step, three possibilities are presented depending on the orientation of the corner, which are evaluated through nested decisions.

Table 147. P*rocedure for Ordering Upper Corners*

```
1   Order_upper_corner()
2       Identify current_position of the corner
3       // Move piece to the front-down-right position
4       Switch current_position do
5           Current_position = upper layer: back-right corner
6               Right → forward
7               Down → left
8               Down → left
9               Right → back
10              Down → right
11          Current_position = upper layer: back-left corner
12              Left → forward
13              Down → right
14              Down → right
15              Left → back
16          Current_position = upper layer: front-right corner
```

```
17                  Right → back
18                  Down → left
19                  Right → forward
20                  Down → right
21              Current_position = upper layer: front-left corner
22                  Left → back
23                  Down → right
24                  Left → forward
25              Current_position = down layer: back-right corner
26                  Down → left
27              Current_position = down layer: back-left corner
28                  Down → right
29                  Down → right
30              Current_position = down layer: front-left corner
31                  Down → right
32          End switch
33          // Move and orient piece to the upper-front-right position
34          If reference_color = front color of down-right corner then
35              Down → left
36              Right → back
37              Down → right
38              Right → forward
39          Else
40              If reference_color = right color of down-right corner then
41                  Down → right
42                  Front → right
43                  Down → left
44                  Front → left
45              Else
46                  Right → back
47                  Down → right
48                  Right → forward
49                  Front → right
50                  Down → right
51                  Down → right
52                  Front → left
53              End if
54          End if
55      End order_upper_corner
```


This procedure is executed four times, as established by the iterative structure of the *Solve_upper_layer()* procedure, after which the cube will have the first layer ordered, as shown in Figure 136.

Figure 137. *Ordered Upper Layer*

9.2.5 Solving the Central Layer

After ordering the upper layer, the next step is to arrange the four edges of the central layer. It is important to note that there is no point in assembling this face if any of the pieces of the upper layer are not in their correct positions and orientations.

To order the central layer, the down edge of the central layer is observed to determine whether it corresponds to the right or left position of the central layer, and then it is moved accordingly. It is also possible for an edge to be in the central layer but not in the correct position or orientation; in this case, it is moved to the down layer and then positioned correctly in the central layer.

The procedure in Table 148 defines a loop for ordering the four edges, within which a second loop searches the down layer for the edge that corresponds to the left or right central position of each face.

Table 148. *Procedure for Solving the Central Layer*

```
1   Solve_central_layer()
2     Integer: sorted_edges, counter
3     sorted_edges = edges sorted in the central layer
4     // Loop to order the edges of the central layer
5     While sorted_edges < 4 do
6     counter = 1
7         While counter <= 4 do
8             If front color of front-down edge = front-center color and down color
              of front-down edge = right-center color then
9                 Order_right_central_edge()
10            Else
11                If front color of front-down edge = front center color and down
                  color of front-down edge = left center color then
12                    Order_left_central_edge()
13                End if
14            End if
15            Down → left
16            counter = counter + 1
17        End while
          If right-front edge is NOT sorted and front color of central-right edge ≠
18        down-center color and right color of central-right edge ≠ down-center
          color then
19            Move_right_central_edge()
20        End if
21        Rotate cube to the left
22    End while
23  End Solve_central_layer
```

This procedure is designed to identify the edge to be moved, either from the down layer to the central layer or from the central face to the down layer. Once a piece that needs to be moved is located, it invokes the procedures ***Order_left_central_edge()***, ***Order_right_central_edge()*** or ***Move_right_central_edge()***, as appropriate for the position and colors of the edge.

The ***Order_right_central_edge()*** procedure is responsible for moving the edge that occupies the down position of the front layer to the right position of the same layer; in other words, it moves the edge from the down layer to the central layer. This is shown in Figure 137, and it is necessary for the front color of the edge to match the front center color, and the down layer of the edge to be the same color as the center of the right central. This procedure is shown in Table 149.

Figure 138. *Ordering Right Central Edge*

The *Order_left_ central_edge()* procedure, presented in Table 150, performs a similar task to the previous one, with the difference that it moves the down edge to the left side of the front layer.

Table 149. *Procedure for Ordering the Right Central Edge*

```
1    Order_right_central_edge()
2        Down → left
3        Right → back
4        Down → right
5        Right → forward
6        Down → right
7        Front → right
8        Down → left
9        Front → left
10   End procedure
```

Table 150. *Procedure for Ordering the Left Central Edge*

```
1    Order_left_central_edge()
2        Down → right
3        Left → back
4        Down → left
5        Left → forward
6        Down → left
7        Front → left
8        Down → right
```

```
9           Front → right
10      End procedure
```

The two previous procedures are used to move a piece from the bottom layer to the central layer. However, sometimes the edges are located in the central layer but in positions that do not correspond to them, and it is necessary to move them to the bottom layer. For this purpose, the ***Move_right_central_edge()*** procedure is used, as presented in Table 151.

Table 151. *Procedure for Moving the Right Central Edge*

```
1     Move_right_central_edge()
2           Right: back
3           Down: right
4           Right: forward
5           Down: right
6           Front: right
7           Down: left
8           Front: left
9     End Move_right_central_edge
```

Once this phase is complete, the cube will look as shown in Figure 138.

Figure 139. *Cube with the Upper and Central Layers Ordered*

9.2.6 Solving the Down Layer

The down layer must be assembled carefully to avoid disturbing the upper and central layers. To facilitate understanding of the procedure and to prevent the sequen-

ces of movements from becoming excessively lengthy, this layer is ordered in three steps:

1. Position the down corners
2. Orient the down corners
3. Order the down edges

At the start of this phase, there should be two corners that are positioned, although not necessarily oriented. To locate them, the down layer is rotated, the colors of the three faces of the piece are observed, and they are compared with the colors of the three centers of the faces that converge at that corner. A corner is positioned if it has the three colors of the faces, but it is only oriented if the color of the corner matches the center color on each face.

The algorithm in Table 152 presents the algorithm for assembling the down layer. The first loop will run until all four corners are positioned. Within this, there is a second nested loop whose purpose is to rotate the down layer until identifying the two corners that are already positioned. Once they have been found, a procedure is executed to position the remaining two. In each execution of the procedure, the corners change position, but that does not guarantee they will reach their correct position; therefore, after each execution, it is necessary to evaluate the state of the layer and execute again, as indicated by the repetition structure.

Table 152. *Procedure for Assembling the Down Layer*

```
1    Assemble_down_layer()
2        While positioned_corners < 4 do
3            // Identify the two that are already positioned
4            While positioned_corners < 2 do
5                Down → left
6            End while
7            position_down_corners()
8        End while
9        // orient down corners
10       While oriented_corners < 4 do
11           Orient_Down_Corners()
12       End while
13       // Orient down edges
14       While oriented_edges < 4 do
```

```
15                Order_down_edges()
16            End while
17        End procedure
```

After positioning the corners, the next step is to orient them, ensuring that the colors of the faces of the corner match the colors of the cube's centers. This procedure is also executed until all four corners are oriented. Finally, the edges of the down layer are positioned and oriented, for which another loop is included in the algorithm.

Positioning Down Corners

At this point, you have identified which two corners are positioned. These may be adjacent or diagonal with respect to each other, as shown in Figure 139. It is important to note whether they are adjacent or diagonal, as in the latter case an additional movement is required.

The procedure for positioning the down corners is presented in Table 153. As mentioned earlier, if, after executing it, the four corners are not positioned, the two that are positioned should be identified, and the procedure executed again.

Figure 140. *Position of Down Corners*

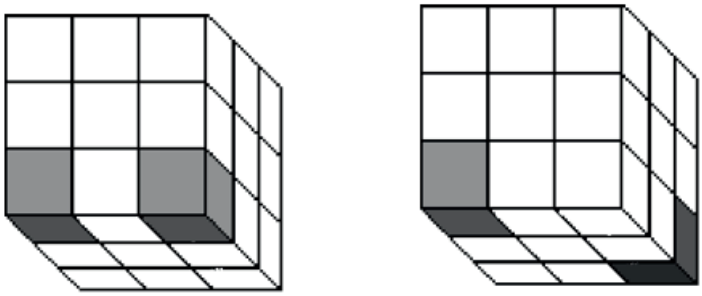


Table 153. *Procedure for Positioning Down Corners*

```
1     Position_down_corners()
2     Integer aux
3         If positioned_corners are consecutive then
4             aux = 1
```

```
5           Place positioned corners in the left front-down and right front-down
            positions
6       Else (if they are diagonal)
7           aux = 2
8           Place positioned corners in the right front-down and left back-down
            positions
9       End if
10      Right → back
11      Down → left
12      Right → forward
13      Front → right
14      If aux = 1 then
15          Down → right
16      Else
17          Down → right
18          Down → right
19      End if
20      Front → left
21      Right → back
22      Down → right
23      Right → Forward
24      Down → right
25  End algorithm
```

Orienting down corners

In the previous step, the corner pieces were placed in their correct positions according to their colors, but they were not oriented so that the colors on their faces match the color of each side of the cube.

At this point in the process, when observing the lower corners of the cube, you may encounter any of the following cases:

1. No corner is oriented
2. Only one is oriented
3. Two are oriented and in adjacent positions
4. Two are oriented and diagonally positioned

In any case, the procedure presented in Table 154 is responsible for orienting the faces of the corners. This subroutine includes two sequences of moves: one for situa-

tions where none or only one of the corners is oriented, and another for when there are two oriented corners. If there are oriented corners, you should rotate the cube until one of the oriented corners is in the down left position, and if there's a second, it should be in the right layer. If there are two oriented corners, whether they are adjacent or diagonal, the sequence of moves remains the same, with the difference that if they are diagonal, an additional move is required in the down layer.

Table 154. *Procedure for Orienting the Down Corners*

```
1     Orient_down_corners()
2         If oriented corners < 2 then
3             If oriented corners = 1 then
4                     Place oriented corner in the front down left position
5             End if
6             Right → back
7             Down → left
8             Right → forward
9             Down → left
10            Right → back
11            Down → right
12            Down → right
13            Right → forward
14            Down → right
15            Down → right
```

Table 154. *(Continued)*

```
1         Else (if two corners are oriented)
2             Place one oriented corner in the front down left position and the
                  other in the right layer
3             Left → forward
4             Upper → right
5             Left → back
6             Front → right
7             Upper → right
8             Front → left
9             If oriented corners are adjacent then
10                Down → right
```

```
11              Else
12                  Down → right
13                  Down → right
14              End if
15              Front → right
16              Upper → left
17              Front → left
18              Left → forward
19              Upper → left
20              Left → back
21              Down → left
22          End if
23      End procedure
```

It may happen that after executing this procedure, the corners are still not oriented; even though they have rotated, the colors may not correspond to the colors of the center pieces on each face. This procedure is executed repeatedly until the corners are sorted, as indicated by the iterative structure of the *Assemble_Down_Layer()* procedure. However, it is important to carefully observe the colors of the pieces; a mistake in executing the moves might cause one of the corners to change position, in which case it is necessary to re-execute the *Position_down_ corners()*.

Once the down corners are oriented, only the edges of the layer will remain, as shown in Figure 140.

Figure 141. *Cube with Down Corners Sorted*

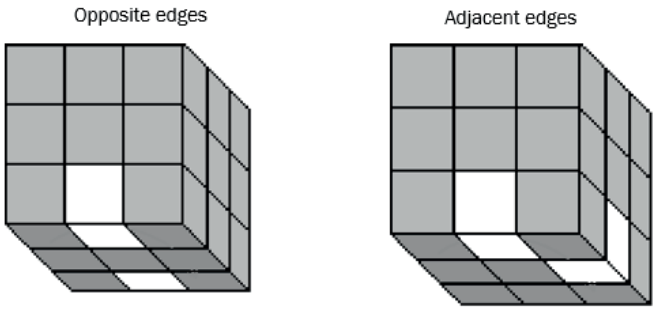
Positioning and Orienting the Down Edges

If the previous procedures have been executed successfully, only the four edges of the down layer remain to be sorted. The following cases may occur with their arrangement:

1. No edge is in the correct position
2. Only one edge is correctly positioned and oriented
3. Two edges are positioned and oriented

If two edges are positioned and oriented, they can be either adjacent or opposite, as shown in Figure 141. The sequence of moves to bring them to their final position depends on this arrangement.

Figure 142. *Position of the Last Two Edges to be Sorted*



In the case that three edges are sorted and the fourth is positioned but not oriented, it means that the cube was disassembled, and upon reassembly, the piece was placed upside down, making it unsolvable unless it is assembled correctly. If, while turning the layers of the cube, one or more pieces accidentally become misaligned, it is advisable to reassemble the cube by placing the pieces such that each face has only one color.

The procedure in Table 156 presents the sequence of moves to sort the edges of the down layer. Three sets of moves are proposed: the first is executed when none or only one edge is positioned and oriented; the second and third are for when two edges are positioned and oriented, depending on whether they are in adjacent or opposite positions.

Table 155. *Procedure for Positioning and Orienting the Down Edges*

```
1   Sort_down_edges()
2       If sorted_edges < 2 then
3           If sorted_edges = 1 then
4               Place sorted edge in the front down position
5           End if
6           Left → forward
7           Right → forward
8           Front → right
9           Left → back
10          Right → back
11          Down → right
12          Down → right
13          Left → forward
14          Right → forward
15          Front → right
16          Left → back
17          Right → back
18      End if
19      If sorted_edges  = 2 then
20          If sorted edges are on adjacent faces then
21              Place the unoriented edges in the positions:
                    front down and left down
22          Left → forward
23          Down → left
24          Upper → right
25          Back → left
26          Back → left
27          Down → left
28          Down → left
29          Upper → right
30          Upper → right
31          Front → left
32          Down → left
```

```
33          Front → right
34          Upper → right
35          Upper → right
36          Down → left
```

```
37                  Down → left
38                  Back → left
39                  Back → left
40                  Down → right
41                  Upper → left
42                  Left → back
43                  Down → right
44              End if
45              If the oriented edges are on opposite faces, then
46                  Place the unoriented edges in the positions:
                        front down and back down
47                  Down → right
48                  Down → right
49                  Right → forward
50                  Left → forward
51                  Front → right
52                  Right → forward
53                  Left → forward
54                  Upper → right
55                  Right → forward
56                  Left → forward
57                  Back → left
58                  Back → left
59                  Right → back
60                  Left → back
61                  Upper → right
62                  Right → back
63                  Left → back
64                  Front → right
65                  Right → back
66                  Left → back
67              End if
68          End if
69      End algorithm
```

As with the previous procedures, it may happen that the pieces do not sort even if their positions or orientations change. This is why this procedure is invoked within a loop, to execute it repeatedly until the goal is achieved. The sequences of moves for sorting the down layer are longer than the previous ones, and as you develop

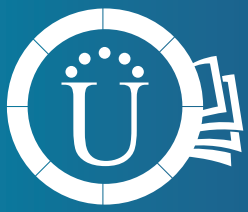
skill in assembling the cube, it is likely that some errors in the routines will occur, causing the central or upper layer to become unsorted. When this happens, do not be discouraged; take a break and start again from the step you are on. Your initial attempts may not reach completion, but if you persist, you will eventually succeed.

If you have successfully assembled it, congratulations! You have shown patience and dedication. If you can solve the Rubik's Cube, you can achieve any goal you set for yourself; many projects, like solving the cube, require more willpower and perseverance than intelligence or strength.

LIST OF FIGURES

LIST OF TABLES

LIST OF EXAMPLES



Sello Editorial
Universidad Nacional
Abierta y a Distancia

Sede Nacional José Celestino Mutis
Calle 14 Sur 14-23
PBX: 344 37 00 - 344 41 20
Bogotá, D.C., Colombia

www.unad.edu.co