TABLE OF CONTENTS

Programming Topics

Non Programming Topics

1

Introduction to Programming

Introduction

A computer is a device that follows instructions and executes commands. To control a computer, you need to write **programs**.

KEY TERM

Program – A list of instructions that does a specific task.

Example:

Write a program that reads two numbers from the user and print out their sum.
In other words, we need to write a program to solve the following equation:

```
Z = A + B
```

Solution:

To write the program we go through **four** general phases:

1. Analysis
2. Design
3. Coding
4. Testing

1.1 Analysis

- Identification of the problem and requirements
- Decomposition of the problem into its component parts which can be drawn as a **structure diagram**

Components/Items after decomposition of the problem:

- Inputs
- Processes
- Outputs
- Storage

In our example program:

- Inputs: **A, B**
- Processing: **Z = A + B**
- Output: **Z**
- Storage: **Variables A, B,Z**

Operators

Types of operators:

1. Arithmetic operators
2. Logic operators
3. Boolean Operators
4. Assignment operator

Arithmetic Operators

The following table shows the various Arithmetic operators and what they do:

Arithmetic operators		
Operator	Symbol	Description
Multiplication	*	Multiplies the left operand by the right operand. Example: x * y
Division	/	Divides the left operand by the right operand Example: x/y
Subtraction	-	Subtracts the right operand from the left operand Example: x - y
Addition	+	Adds the left operand to the right operand Example: x + y
Power	^	Raises the left operand to the power of the right operand Example: x^y is written as x ^ y

Logic Operators

The following table shows some Logic operators:

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

Assignment Operator

- This operator assigns a value or an expression to a variable.
- The symbol used for this operator is an arrow pointing left: ←
- The statement that has an assignment operator is called **assignment statement**

Examples of assignment statements:

```
Cost ← 10

Price ← Cost * 2

Tax ← Price * 0.14

Service ← Price * 0.12

PaidAmount ← Tax + Service
```

1.2 Design

After the analysis phase is done, we design the **Algorithm**.

We can represent an algorithm using:

- Pseudocode
- Flowcharts
- Structure Diagrams

Pseudocode

INPUT Statement

To allow the user to **input** a value we use one of two keywords:

- `INPUT`
- `READ`

OUTPUT Statement

To show the user the value of a variable we use one two keywords:

- `OUTPUT`
- `PRINT`

Example 1.1 :

Write a statement that will output the words **Hello World!** on the user's screen

```
OUTPUT "Hello World!"
```

KEY TERM

Algorithm – A step-by-step solution to a given problem.

Pseudocode – Uses English words and mathematical notations to design the steps of a program.

Example 1.2:

Design a program that displays on the screen the following:

Hello World!

Welcome to CSTEAM

```
OUTPUT "Hello World!"
OUTPUT "Welcome to CSTeam"
```

Example 1.3:

Design a program that displays on the screen the following:

Welcome to CSTEAM

Hello World!

```
OUTPUT "Welcome to CSTeam"
OUTPUT "Hello World!"
```

Variables

Rules for naming variables:

- Variable names cannot contain symbols **including spaces**
- Variable names cannot start with numbers
- Variable name should not be a keyword of pseudocode
- Variable names should be **meaningful** to the value it stores

Examples of valid variable names:

- `number`
- `total`
- `count`
- `count1`
- `countStudents`

Examples of invalid variable names:

- `count  students`
- `1count`
- `OUTPUT`

Constants

Constants are often used to make the program more readable. **Constants have the same naming rules as variables.**

Defining a constant in pseudocode:

```
CONSTANT NumberOfElements <- 10
```

KEY TERM

Variable – A named memory location that stores a value that can change during the execution of a program.

Constant – A named memory location that stores a value that **does not** change during the execution of a program.

Data Types

Computers need to know the data type for every variable, to make the processing on this variable as effective as possible.

The following table shows the data types used in programs:

Data type	Definition	Examples
Integer	A positive or negative **whole** number be used in mathematical operations	150, -100, 0
Real	A positive or negative number that has a **fractional part** that can be used in mathematical operations	100.5, -15.2
Char	A **single** character from the keyboard	H
String	A **sequence** of characters	Hello World, A312_@odq
Boolean	Data with **two possible values**	TRUE/FALSE

Declaration

When declaring a variable you have to do **two** things:

- Give your variable a **valid name**
- Give it a **data type**

Declaration syntax in pseudocode:

```
DECLARE [VARIABLE_NAME]: [DATA_TYPE]
```

Example 1.4:

Design a program that takes any name from the user and displays a customized hello message. **Hello [any name]**

```
DECLARE AnyName : STRING
OUTPUT "What is your name?"
INPUT AnyName
OUTPUT "Hello ", AnyName
```

Example 1.5:

Design a program that reads from the user two numbers and displays their sum

```
DECLARE Num1: REAL
DECLARE Num2: REAL
DECLARE Sum: REAL
OUTPUT "Please enter two numbers to add"
INPUT Num1
INPUT Num2
Sum <- Num1 + Num2
OUTPUT "The number is ", Sum
```

Example 1.6:

Two variables called `Number1`, `Number2` contain two numbers in a calculator program.
Write a program that would calculate and display the sum of the two numbers
You do **not** need to initialise the data in the variables `Number1` and `Number2`

```
DECLARE Sum: REAL
Sum <- Number1 + Number2
OUTPUT "The sum is ", Sum
```

Example 1.7:

Two variables called `Number1`, `Number2` contain two numbers in a calculator program.
Write a program that would:

- calculate and display the sum of the two numbers
- calculate and display the subtraction of the two numbers
- calculate and display the multiplication of the two numbers
- calculate and display the division of the two numbers

You do **not** need to initialise the data in the variables `Number1` and `Number2`

```
DECLARE Sum: REAL
DECLARE Subtraction: REAL
DECLARE Product: REAL
DECLARE Division: REAL
Sum <- Number1 + Number2
Subtraction <- Number1 - Number2
Product <- Number1 * Number2
Division <- Number1 / Number2
OUTPUT "The sum is ", Sum
OUTPUT "The subtraction is ", Subtraction
OUTPUT "The Product is ", Product
OUTPUT "The division is ", Division
```

Comments

Blocks of text in the pseudocode that are not executed used to explain the code.
Example

Example 1.8:

Design a program that reads from the user two numbers and displays their sum, make sure to add comments

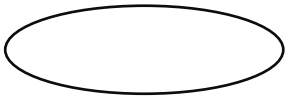
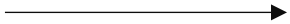
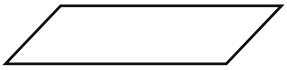
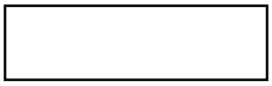
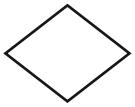
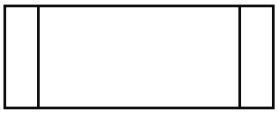
```
// Declare two variables for the first and second number
DECLARE Num1: REAL
DECLARE Num2: REAL
// Declare a third variable to store the sum
DECLARE Sum: REAL
// Ask the user to enter two numbers
OUTPUT "Please enter two numbers to add"
INPUT Num1
INPUT Num2
// Calculate the sum by adding Num1 and Num2
// Store the result in the Sum variable
Sum <- Num1 + Num2
// Display the result on the screen
OUTPUT "The number is ", Sum
```

Features of a maintainable program

- Using Meaningful identifiers (names) for variables, constants, arrays, and procedures/functions
- Using the comment feature provided by the programming language
- Using procedures and functions
- Using indentation and whitespaces

Flowcharts

Flowcharts have some shapes that we use to draw the diagram with:

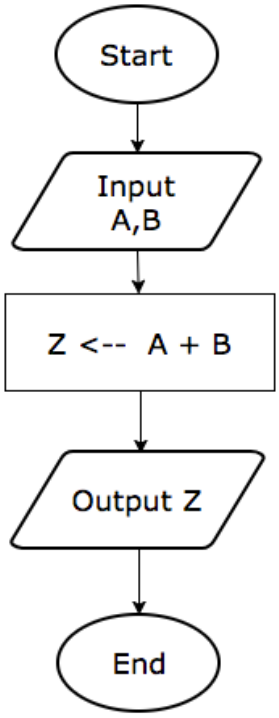
Shape	Name	Description
	Start/End	An oval represents a start or end point
	Arrows	A line is a connector that shows the relationship between the shapes
	Input / Output	A parallelogram represents output or input
	Process	A rectangle is a process
	Decision	A diamond indicates a decision
	Subroutine call	This shape represents that a separate flowchart is being called

KEY TERM

Flowchart – Diagram that designs the steps of a program by using a standard set of symbols joined by lines to show the direction of flow.

Example 1.9:

Design a flowchart to read two numbers from the user and display their sum



1.3 Coding

- The algorithm is transformed into code using a programming language (for example: c++, java, python)
- May include early iterative testing (testing the program over and over)

1.4 Testing

The last phase of the programming, this phase is done after implementation by testing how the program works using test data, we will discuss this later.

2

Conditional Statements

Part I

Conditional/Selection Statements

Purpose of conditional statements:

To allow different paths through a program depending on meeting certain criteria.

KEY TERM

Sequential statements – Statements are executed one after another according to their order.

Conditional/Selection statements – Selective statements are executed depending on meeting certain criteria.

IF .. THEN .. ELSE .. ENDIF – A conditional structure with different outcomes for true and false.

2.1 IF .. THEN .. ELSE .. ENDIF:

Purpose:

It allows for checking complex conditions.

Syntax in pseudocode:

```
IF <condition> THEN
     <statements>    // What to do if the condition is true
ELSE
     <statements>    // What to do if the condition is false
ENDIF
```

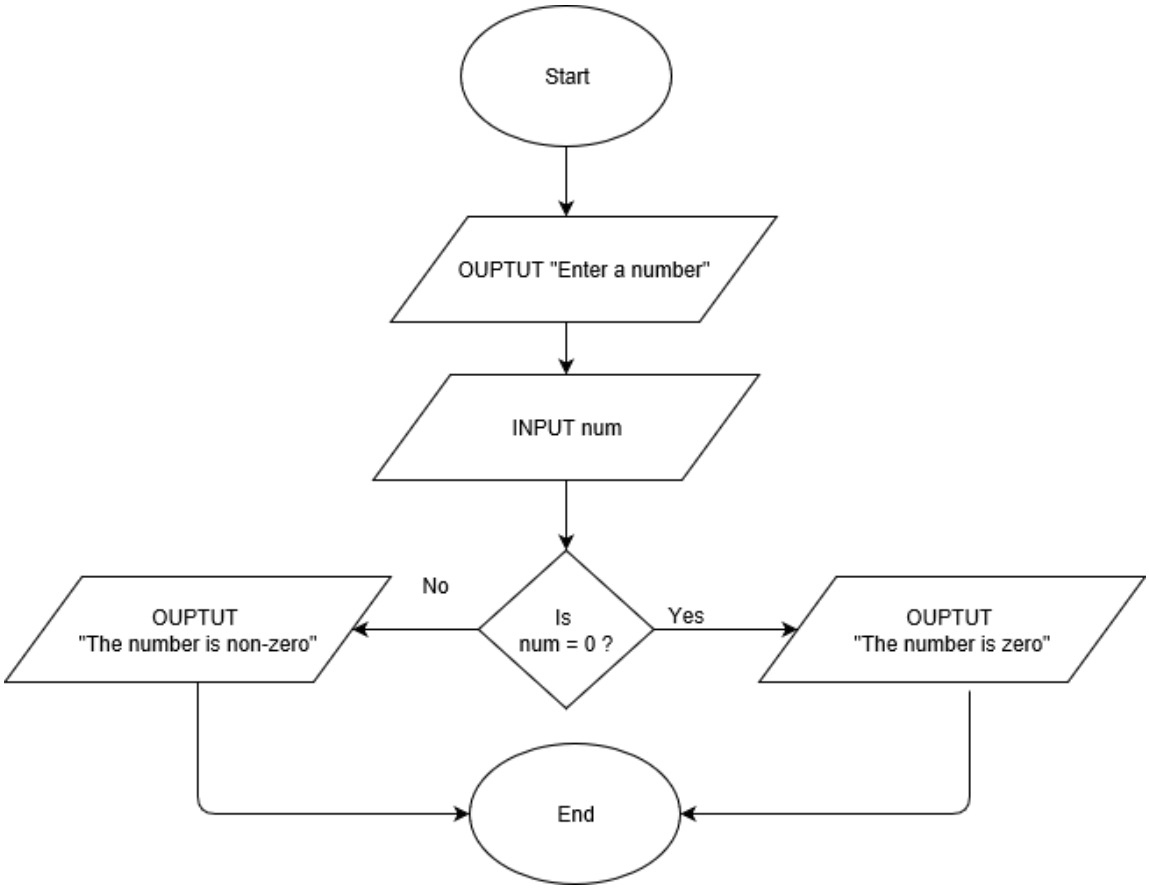
Example 2.1.1:

Design a program, using pseudo-code and flowchart, which takes a number as input and outputs whether the number is zero or non-zero.

Pseudocode Solution:

```
DECLARE num: REAL
OUTPUT "Enter a number"
INPUT num
IF num = 0 THEN
    OUTPUT "The number is zero"
ELSE
    OUTPUT "The number is non-zero"
ENDIF
```

Flowchart Solution:



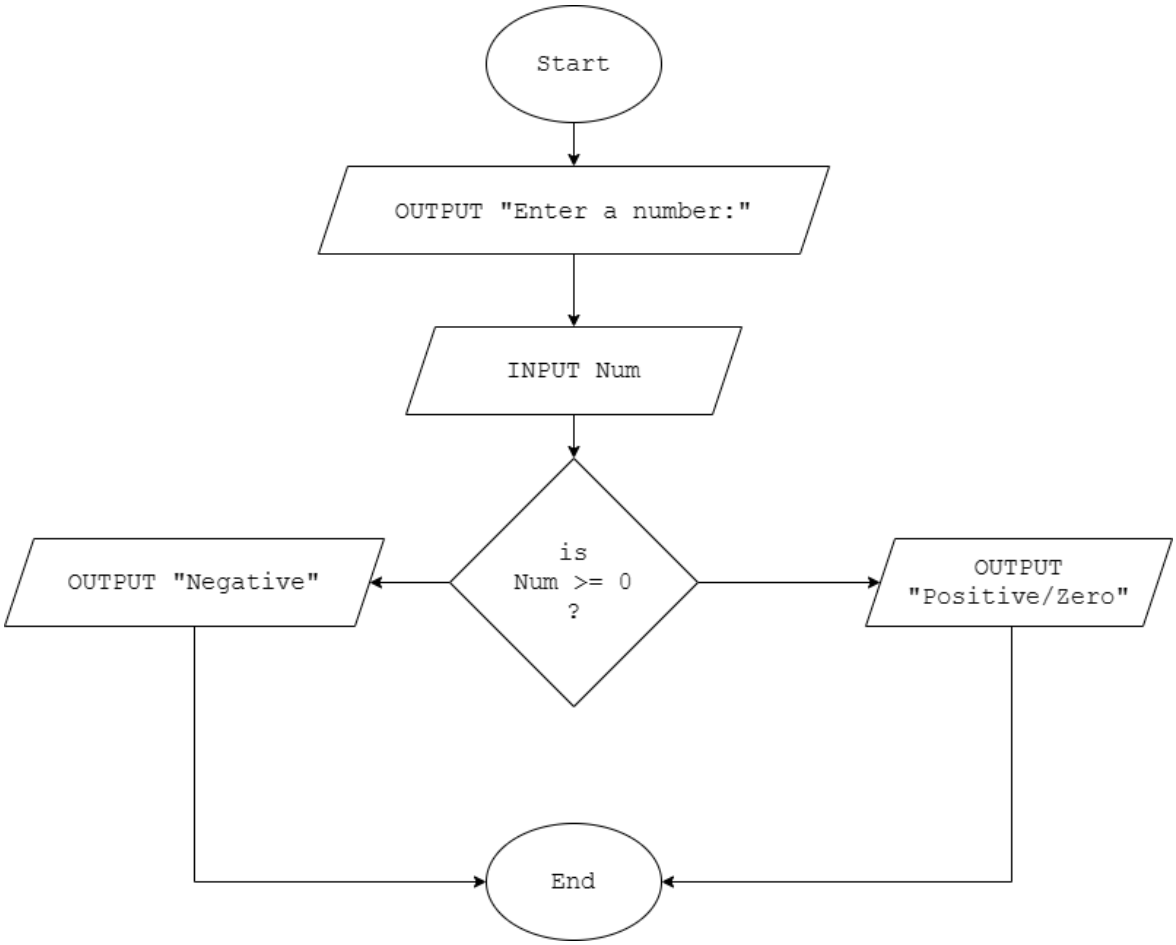
Example 2.1.2:

Design a program that inputs a number and displays **Positive/Zero** if the number is positive or zero and displays **Negative** if the number is negative

Pseudocode Solution:

```
DECLARE num: REAL
OUTPUT "Enter a number"
INPUT num
IF num >= 0 THEN
    OUTPUT "Positive/Zero"
ELSE
    OUTPUT "Negative"
ENDIF
```

Flowchart Solution:



Validation

We need to check that the user has entered sensible data,

Example 2.1.3:

Read two numbers and divide them **Z = A / B.**

A computer cannot divide by zero

Pseudocode Solution:

```
DECLARE Num1,Num2,Result: REAL
OUTPUT "Enter the two number"
INPUT Num1,Num2
IF Num2 = 0 THEN
    OUTPUT "MATH ERROR"
ELSE
    Result <- Num1 / Num2
    OUTPUT "The result of division is:", Result
ENDIF
```

Example 2.1.3:

Write a program to print **Whole number** if the user entered an Integer and output **Real** otherwise.

Pseudocode Solution:

```
DECLARE Number : REAL
OUTPUT "Enter a number"
INPUT Number
IF Number = ROUND(Number, 0) THEN
    OUTPUT "Whole Number"
ELSE
    OUTPUT "Real
ENDIF
```

2

Conditional Statements

Part II

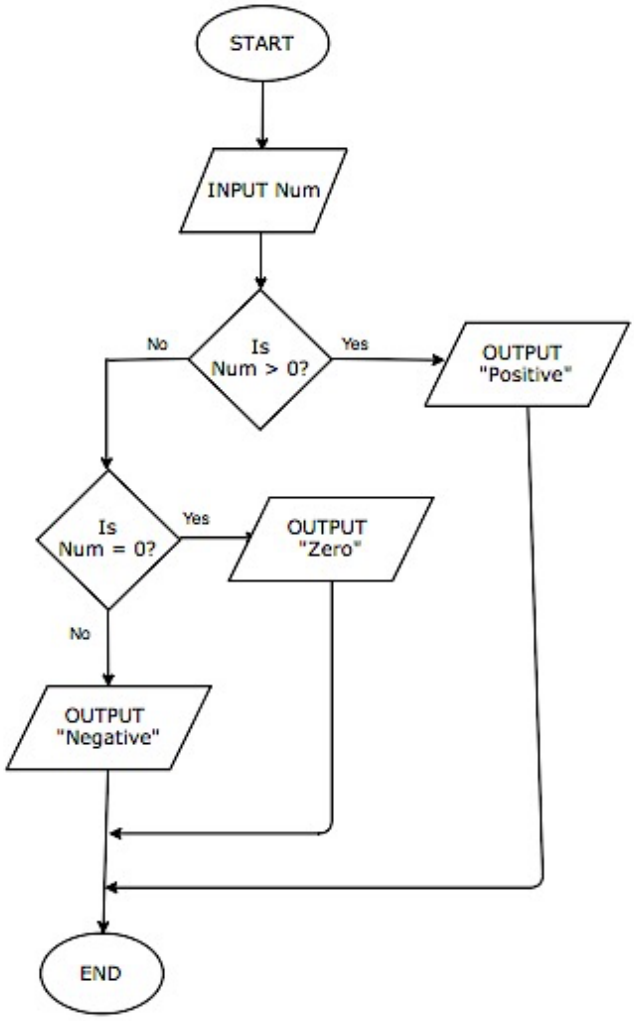
Example 2.2.1:

Design a program, using pseudo-code and flowchart, which takes a number as input and outputs whether the number is positive or negative or zero.

Pseudocode Solution:

```
DECLARE Num: REAL
INPUT num
IF num > 0 THEN
    OUTPUT "Positive"
ELSE
    IF num = 0 THEN
        OUTPUT "Zero"
    ELSE
        OUTPUT "Negative"
    ENDIF
ENDIF
```

Flowchart Solution:



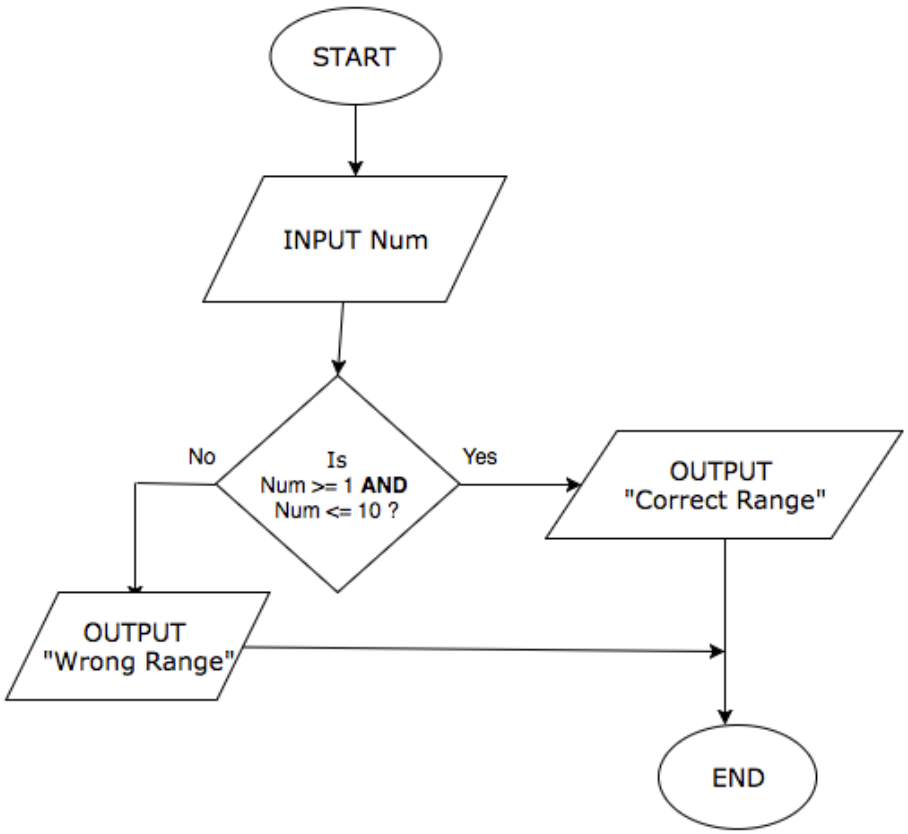
Example 2.2.2:

Design a program, using pseudo-code and flowchart, which takes a number and prints **In range** if the number is between 1 and 10 inclusive, otherwise the program outputs **Outside range**.

Pseudocode Solution:

```
DECLARE Num: REAL
INPUT Num
IF Num >= 1 AND Num <= 10 THEN
    OUTPUT "Correct Range"
ELSE
    OUTPUT "Wrong Range"
ENDIF
```

Flowchart Solution:



Example 2.2.3:

Design a program, using pseudocode and flowchart, which takes marks in an exam as input and outputs the grade according to the following grade boundaries:

Mark	Grade
90 - 100	A
80 – 89	B
70 - 79	C
< 70	F

Pseudocode Solution:

```
DECLARE Mark: INTEGER
OUTPUT "Enter the mark"
INPUT Mark
IF Mark > 100 OR Mark < 0 THEN
    OUTPUT "Invalid Mark"
ELSE
    IF Mark >= 90 THEN
        OUTPUT "A"
    ELSE
        IF Mark >= 80 THEN
            OUTPUT "B"
        ELSE
            IF Mark >= 70 THEN
                OUTPUT "C"
            ELSE
                OUTPUT "F"
            ENDIF
        ENDIF
    ENDIF
ENDIF
```

2

Conditional Statements

Part III

2.3 CASE .. OF .. OTHERWISE .. ENDCASE:

Purpose:

Used to check for a number of discrete values.

Syntax in pseudocode:

```
CASE OF <variable>
    <value 1> : <statement> // do something if variable equals to value 1
    <value 2> : <statement> // do something if variable equals to value 2
    .....
OTHERWISE <statement>
     // do something if the variable doesn't equal any of the cases
ENDCASE
```

KEY TERM

CASE statement– A conditional structure that allows selection to be made based on value of a variable.

Example 2.3.1:

Design a program, using pseudocode and flowchart, which takes a course number as input and outputs the corresponding course name according to this table:

Course Number	Course Name
1	Computer Science
2	ICT
3	Math
4	Biology

Pseudocode Solution:

```
DECLARE CourseNo: INTEGER
OUTPUT "Enter the course number"
INPUT CourseNo
CASE OF CourseNo
    1: OUTPUT "Computer Science"
    2: OUTPUT "ICT"
    3: OUTPUT "Math"
    4: OUTPUT "Biology"
OTHERWISE
    OUTPUT "Invalid course number"
ENDCASE
```

Example 2.3.2:

Write pseudocode that takes two numbers as input and performs any of the basic operation
(+, -, *, /) as per user requires.

Solution:

```
DECLARE Number1, Number2, Result: INTEGER
DECLARE Operation: CHAR
OUTPUT "Enter the first number"
INPUT Number1
OUTPUT "Enter the operation"
INPUT Operation
OUTPUT "Enter the second number"
INPUT Number2
CASE OF Operation
    '+':  Result <- Number1 + Number2
          OUTPUT Result
    '-':  Result <- Number1 - Number2
          OUTPUT Result
    '*':  Result <- Number1 * Number2
          OUTPUT Result
    '/':  IF Number2 = 0 THEN
            OUTPUT "Math Error"
          ELSE
           Result <- Number1 / Number2
           OUTPUT Result
          ENDIF
OTHERWISE
        OUTPUT "Invalid operation"
ENDCASE
```

3

Loop Statements

Part I

REPEAT ... UNTIL

3.1 Post-Conditional Loop (`REPEAT ... UNTIL ...`)

Definition:

- A loop that iterates until a condition is met
- Criteria is post-tested (It has criteria check at the end)
- It iterates at least once

Syntax in pseudocode:

```
REPEAT

     <statements>    // statements to be repeated

UNTIL <condition>
```

Example 3.1.1:

Write a program, using pseudocode, that would read numbers from the user **until the user enters 0.**

Only then the program outputs **Thank you for using our program**

```
DECLARE Number : REAL
REPEAT
    OUTPUT "Enter a number"
    INPUT Number
UNTIL Number = 0
OUTPUT "Thank you for using our program"
```

Example 3.1.2:

Write a program, using pseudocode, that would read numbers from the user **until the user enters a negative number.**

Only then the program outputs **Thank you for using our program**

```
DECLARE Number : REAL
REPEAT
    OUTPUT "Enter a number"
    INPUT Number
UNTIL Number < 0
OUTPUT "Thank you for using our program"
```

Example 3.1.3:

Design a program that takes from the user a radius and the program outputs the circumference

Circumference = 2 * π * Radius

Pseudocode solution:

```
DECLARE Radius,Circumference: REAL
REPEAT
    OUTPUT "Enter the radius:"
    INPUT Radius
    IF Radius <= 0 THEN
        OUTPUT "Radius cannot be less than zero, try again:"
    ENDIF
UNTIL Radius > 0
Circumference <- 2 * 3.14 * Radius
OUTPUT "The Circumference:",Circumference
```

Example 3.1.4:

Design a program, using pseudocode only, which asks the user to enter his age and displays on the screen **You're a teenager** if the user is younger than 20 years, and displays on the screen **You're an adult otherwise.**

Age can't be less than 1

Age can't be more than 100

Pseudocode solution:

```
DECLARE Age : INTEGER
REPEAT
    OUTPUT "Enter the age:"
    INPUT Age
    IF Age < 1 OR Age > 100 THEN
        OUTPUT "Invalid age, try again:"
    ENDIF
UNTIL Age <= 100 AND Age >= 1
IF Age >= 20 THEN
    OUTPUT "You're an adult"
ELSE
    OUTUPT "You're a teenager"
ENDIF
```

3

TEAM

Loop Statements

Part II

WHILE .. DO .. ENDWHILE

3.2 Pre-conditional Loop (`WHILE ... DO ... ENDWHILE`)

Definition:

- A loop that iterates whilst a specified condition exists
- Criteria is pre-tested (It has criteria check at the start)
- It may never run

Syntax in pseudocode:

```
WHILE <condition> DO

     <statements>    // statements to be repeated in here

ENDWHILE
```

Example 3.2.1:

Write a program, using pseudocode, that would read numbers from the user **until the user enters 0.**

Only then the program outputs **Thank you for using our program**
Use `WHILE .. DO .. ENDWHILE`

```
DECLARE Number : REAL
OUTPUT "Enter the first number"
INPUT Number
WHILE Number <> 0 DO
    OUTPUT "Enter a number"
    INPUT Number
ENDWHILE
OUTPUT "Thank you for using our program"
```

Example 3.2.1:

Design a program that will read the side length of a square and displays its area.

Area = side * side

Repeatedly display an error message and ask the user to **re-enter** the side length if the side length is negative.

Pseudocode solution:

```
DECLARE Side, Area : REAL
OUTPUT "Enter the side length of a square : "
INPUT Side
While Side <= 0 DO
    OUTPUT "Invalid side length, try again: "
    INPUT Side
EndWhile
Area <- Side * Side
OUTPUT "Area = ", Area
```

Counting

Definition:

The process of finding how many numbers/items are in a list.

Examples:

```
Count ← Count + 1
Count ← Count + 5
Count ← Count - 1
```

Example 3.2.2:

Design a program that takes a mark from the user and validates that mark.

Quiz is out of 100.

Count how many times the user entered invalid data

Pseudocode solution:

```
DECLARE Mark, CountInvalid : INTEGER
CountInvalid <- 0
OUTPUT "Enter the mark:"
INPUT Mark
While Mark < 0 OR Mark > 100 Do
    OUTPUT "Invalid mark, try again: "
    INPUT Mark
    CountInvalid <- CountInvalid + 1
EndWhile
OUTPUT CountInvalid, " invalid entries"
```

Totaling

Definition:

Totaling is a process of summing a list of number.

Examples:

```
Total ← Total + 5
Sum ← Sum + Num
```

Example 3.2.3:

Design a program that would read numbers until the user enters **0**.

Display the total (sum) of those numbers.

```
DECLARE Number, Total : REAL
Total <- 0
OUTPUT "Enter the first number"
INPUT Number
WHILE Number <> 0 DO
    Total <- Total + Number
    OUTPUT "Enter another number"
    INPUT Number
ENDWHILE
OUTPUT "Total of the numbers is ", Total
```

Example 3.2.3:

Design a program that takes a mark from the user and validates that mark. Display whether the student has passed or failed.

Quiz is out of 100.

Pseudocode solution:

```
DECLARE Mark: INTEGER
OUTPUT "Enter the mark:"
INPUT Mark
While Mark < 0 OR Mark > 100 Do
    OUTPUT "Invalid mark, try again: "
    INPUT Mark
EndWhile
If Mark >= 50 Then
    OUTPUT "Passed"
Else
    OUTPUT "Failed"
EndIf
```

KEY TERM

Totalling – Totaling is a process of summing a list of number.

Counting – The process of finding how many numbers/items are in a list.

Differences between REPEAT … UNTIL and WHILE … DO … ENDWHILE:

- In WHILE the criteria is pre-tested but in REPEAT..UNTIL the criteria is post-tested
- WHILE may never run while REPEAT UNTIL runs at least once
- WHILE iterates whilst a condition is TRUE but REPEAT UNTIL iterates till a condition is TRUE

Purpose of Iteration/Repetition statements:

To repeat same or similar code a number of times, which allows for shorter code.

IMPORTANT:

- Both REPEAT..UNTIL and WHILE..DO..ENDWHILE loops are called **Condition Controlled Loops**
- FOR loop is called **counter-controlled loop.**

3

Loop Statements

Part III

FOR … TO … NEXT

3.3 Count-controlled Loop (FOR … TO … NEXT)

Syntax in pseudocode:

```
FOR <variable> <- <initial value> TO <final value>
     <statements>   // statement(s) to be repeated
NEXT
```

Example 3.3.1:

Write a program, using pseudocode, that would read **5 numbers** from the user

Only then the program outputs **Thank you for using our program**

```
DECLARE Number : REAL
FOR Count <- 1 TO 5
    OUTPUT "Enter a number"
    INPUT Number
NEXT Count
OUTPUT "Thank you for using our program"
```

Example 3.3.2:

Design pseudo-code and flowchart that prints the word **Hello** 10 times.

Pseudocode solution:

```
FOR i <- 1 TO 10
    OUTPUT "Hello"
NEXT i
```

KEY TERM

Repetition/Iterative statements – One or more statements are repeated till a test condition is met or whilst a test condition is TRUE.

FOR Loop – A loop structure that iterates a set number of times. The FOR loop is a counter controlled loop

Example 3.3.3:

Design a program **using loops** that takes 5 numbers from the user. Calculate and display the total of these numbers.

Pseudocode solution:

```
DECLARE Num,Sum: INTEGER
Sum <- 0
FOR Count <- 1 TO 5
    OUTPUT "Enter the number"
    INPUT Num
    Sum <- Sum + Num
NEXT Count
OUTPUT "The total is: ", Sum
```

Example 3.3.4:

Write pseudocode that inputs 50 numbers and counts the numbers that are greater than 100.

Pseudocode solution:

```
DECLARE Num,Count: INTEGER
Count <- 0
FOR Index <- 1 TO 50
    OUTPUT "Enter the number"
    INPUT Num
    IF Num > 100 THEN
        Count <- Count + 1
    ENDIF
NEXT Index
OUTPUT "The count of numbers greater than 100 is: ",Count
```

Example 3.3.5:

Write pseudocode that reads 100 numbers and prints out their **average.**

Pseudocode solution:

```
DECLARE Sum,Num,Average: REAL
Sum <- 0
FOR Index <- 1 TO 100
    OUTPUT "Enter number"
    INPUT Num
    Sum <- Sum + Num
NEXT Index
Average <- Sum / 100
OUTPUT "The average is ", Average
```

Example 3.3.6:

Rewrite the pseudocode given below by using REPEAT .. UNTIL loop and another time using WHILE … DO … ENDWHILE:

```
DECLARE Num,Total : INTEGER
Total <- 0
FOR i <- 1 TO 100
     INPUT Num
     Total <- Total + Num
NEXT
OUTPUT Total
```

Pseudocode solution (WHILE … DO … ENDWHILE):

```
DECLARE Num, Total,i : INTEGER
Total <- 0
i <- 1
WHILE i <= 100 DO
    INPUT Num
    Total <- Total + Num
    i <- i + 1
ENDWHILE
OUTPUT Total
```

Pseudocode solution (REPEAT … UNTIL):

```
DECLARE Num, Total,i : INTEGER
Total <- 0
i <- 1
REPEAT
    INPUT Num
    Total <- Total + Num
    i <- i + 1
UNTIL i > 100
OUTPUT Total
```

Example 3.4.1:

Write a pseudocode that takes from the user 50 marks of a CS quiz (out of 100).

The program should output the average mark of the class.

DON'T FORGET TO VALIDATE THE MARKS

Pseudocode solution:

```
DECLARE Mark, Sum: INTEGER
DECLARE Average : REAL
Sum <- 0
FOR Counter <- 1 TO 50
    OUTPUT "Enter the mark"
    INPUT Mark
    While Mark < 0 OR Mark > 100 Do
        OUTPUT "Invalid mark, try again:"
        INPUT Mark
    EndWhile
NEXT Counter
Average <- Sum / 50
OUTPUT "Average mark is: ",Average
```

Example 3.4.2:

Write pseudocode that reads 100 numbers and prints out their **minimum and maximum**.

Pseudocode solution 1:

```
DECLARE Min,Max,Num: INTEGER
Min <- 99999
Max <- -99999
FOR Count <- 1 TO 100
    OUTPUT "Enter number"
    INPUT Num
    IF Num < Min THEN
        Min <- Num
    ENDIF
    IF Num > Max THEN
        Max <- Num
    ENDIF
NEXT Count
OUTPUT "Minimum number entered is: ",Min
OUTPUT "Maximum number entered is: ",Max
```

Pseudocode solution 2:

```
DECLARE Min,Max,Num: INTEGER
OUTPUT "Enter number"
INPUT Num
Min <- Num
Max <- Num
FOR Count <- 1 TO 99
    OUTPUT "Enter number"
    INPUT Num
    IF Num < Min THEN
        Min <- Num
    ENDIF
    IF Num > Max THEN
        Max <- Num
    ENDIF
NEXT Count
OUTPUT "Minimum number entered is: ",Min
OUTPUT "Maximum number entered is: ",Max
```

4

1D Arrays

4.1 Arrays

Example 4.1.1:

Write a pseudocode that outputs numbers from 1 to 50

Pseudocode solution:

```
For Index <- 1 TO 50
    OUTPUT Index
NEXT Index
```

Example 4.1.2:

Write a pseudocode that outputs numbers from 50 to 1

Pseudocode solution:

```
For Index <- 50 TO 1 STEP -1
    OUTPUT Index
NEXT Index
```

Declaring a 1D array

To declare a 1D array, you need to specify the name, data type and size of the

Pseudocode declaration:

```
DECLARE <array_name>[1:N] : <data_type>
//N is the number of elements in the array
```

Example 4.1.3:

Write a program that would:

- allow the user to input 5 numbers and store them in a 1D array

Pseudocode solution:

```
DECLARE Numbers[1:5] :REAL
FOR Index <- 1 TO 5
    OUTPUT "Enter a number"
    INPUT Numbers[Index]
NEXT Index
```

Accessing array elements

To access an element of the array, the **index** (position) should be specified.

The index can be a variable, or an expression that is evaluated to an integer

For example:

```
OUTPUT Numbers[1]
```

The above statement will output the value stored in the first place in the array Numbers

Example 4.1.4:

Write a program that would:

- allow the user to input 5 numbers and store them in a 1D array

Pseudocode solution:

```
DECLARE Numbers[1:5] :REAL
DECLARE Total : REAL
Total <- 0
FOR Index <- 1 TO 5
    OUTPUT "Enter a number"
    INPUT Numbers[Index]
    Total <- Total + Numbers[Index]
NEXT Index
OUTPUT "The total is ", Total
```

Purpose of using arrays:

- To store multiple values under the same identifier making the code shorter
- Allows for simpler programming

KEY TERM

Arrays – A data structure that holds collection of elements of the same data type.

Example 4.1.5:

Write pseudocode that reads the ages of 100 students from the user and stores them. Then use the stored values to find the count of students who are older than 18 years.

Pseudocode solution:

```
DECLARE Ages [1:100] : INTEGER
DECLARE Count: INTEGER

FOR Index <- 1 TO 100
    INPUT Ages[Index]
NEXT Index
Count <- 0
FOR Index <- 1 TO 100
    IF Ages[Index] > 18 THEN
        Count <- Count + 1
    ENDIF
NEXT Index
OUTPUT "Count of students older than 18 is: ",Count
```

4.2 Associative Arrays

Sometimes you might need to store two values about a certain thing, for example, the name and age of all students. In this case we'd need to have two arrays, one array for storing the names, and one for storing the ages.

Example 4.2.1:

The 1D array `Names[]` contains the names of 5 students. Another 1D array `Attended[]` contains whether each student has attended or not

Write a program that would:

- input the name of 5 students and whether they have attended or not
- find and output the names of students who were absent

Pseudocode solution:

```
DECLARE Names[1:5] : STRING
DECLARE Attended[1:5] : BOOLEAN
FOR Index <- 1 TO 5
    OUTPUT "Enter the student name"
    INPUT Names[Index]
    OUTPUT "Has he/she attended? (TRUE/FALSE)"
    INPUT Attended[Index]
NEXT Index
OUTPUT "Student(s) who were absent"
FOR Index <- 1 TO 5
    IF Attended[Index] = FALSE THEN
        OUTPUT Names[Index]
    ENDIF
NEXT Index
```

Example 4.2.2:

The names of students are stored in a 1D array `Names[]`. The marks of students in a single quiz are stored in the 1D `Marks[]`.

There are 5 students in the class and all of them took the quiz.

The arrays are already setup and the data is stored.

Write a program that would:

- calculate and output the average mark of the class
- find and output the names of students **who scored less than 50**

```
DECLARE Total, Average : REAL
Total <- 0
FOR Index <- 1 TO 5
    Total <- Total + Marks[Index]
NEXT Index
Average <- Total / 5
OUTPUT "Those who scored less than 50:"
FOR Index <- 1 TO 5
    IF Marks[Index] < 50 THEN
        OUTPUT Names[Index]
    ENDIF
NEXT Index
```

5

String Operations

5. String Operations

Purpose of string operations:

Allows to process the strings and transform them

5.1 UCASE(STRING)

Purpose:

Changes the whole string into **uppercase** letters

Example 1:

Write a program that reads a name and converts it into all capital (**uppercase**) letters

```
DECLARE Name, UpperName : STRING
OUTPUT "Enter a name:"
INPUT Name
UpperName <- UCASE(Name)
OUTPUT "The name in uppercase : ", UpperName
```

5.2 LCASE(STRING)

Purpose:

Changes the whole string into **lowercase** letters

Example 2:

Write a program that reads a name and converts it into all small (**lowercase**) letters

```
DECLARE Name, LowerName : STRING
OUTPUT "Enter a name:"
INPUT Name
LowerName <- LCASE(Name)
OUTPUT "The name in lowercase : ", LowerName
```

5.3 LENGTH(STRING)

Purpose:

Counts and returns the number of characters in a string

Example 3:

Write a program that reads a name and displays the number of characters in the name

```
DECLARE Name : STRING
DECLARE NumberOfCharacters : INTEGER
OUTPUT "Enter a name:"
INPUT Name
NumberOfCharacters <- LENGTH(Name)
OUTPUT "The number of characters : ", NumberOfCharacters
```

5.4 SUBSTRING(STRING, START, NumberOfCharacters)

Purpose:

Takes a string, a starting index and a number of characters and returns this portion of the string

Example 4:

Write a program that reads a name and displays **the first 3 characters** of the name

```
DECLARE Name, Part : STRING
OUTPUT "Enter a name:"
INPUT Name
Part <- SUBSTRING(Name, 1, 3)
OUTPUT "The first 6 characters are ", Part
```

5.5 Accessing characters from string

A string can be treated as an array of characters

Example 5:

Write a program that reads a name from the user and prints each character on a separate line.

```
DECLARE Name : STRING
OUTPUT "Enter the name:"
INPUT Name
FOR Index <- 1 TO LENGTH(Name)
    OUTPUT Name[Index]
NEXT Index
```

Example 6:

Design a program that takes a string as input and display the last 3 characters only

Hint: Use the function `SUBSTRING`

```
DECLARE Name : STRING
OUTPUT "Enter the name:"
INPUT Name
OUTPUT "Last 3 chars : ", SUBSTRING(Name, LENGTH(Name) - 2, 3)
```

Example 7:

Design a program that would read a full name as input and display the number of As in the name

```
DECLARE Count : INTEGER
DECLARE Name : STRING
Count <- 0
OUTPUT "Enter the name:"
INPUT Name
FOR Index <- 1 TO LENGTH(Name)
    IF Name[Index] = 'A' OR Name[Index] = 'a' THEN
        Count <- Count + 1
    ENDIF
NEXT Index
OUTPUT "Number of As : ", Count
```

6

Nested Loops
&
2D Arrays

6.1 Nested For Loops

Example 5.1.1:

Design a program that would output the following using **nested for loops**

```
Set 1
Rep : 1
Rep : 2
Rep : 3
Set 2
Rep : 1
Rep : 2
Rep : 3
Set 3
Rep : 1
Rep : 2
Rep : 3
Set 4
Rep : 1
Rep : 2
Rep : 3
Set 5
Rep : 1
Rep : 2
Rep : 3
```

```
FOR Set <- 1 TO 5
    OUTPUT "Set ", Set
    FOR Rep <- 1 TO 3
        OUTPUT "Rep : ", Rep
    NEXT Rep
NEXT Set
```

Example 6.1.2:

Write a program to output the numbers from 1 to 5 **four** times

```
FOR Time <- 1 TO 4
    FOR Num <- 1 TO 5
        OUTPUT Num
    NEXT Num
NEXT Time
```

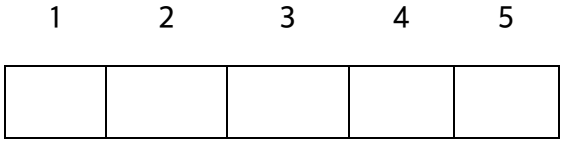
Example 5.1.3:

The array `Numbers[]` already stores **5** numbers. Write a program that would display each number in the array **3 times**

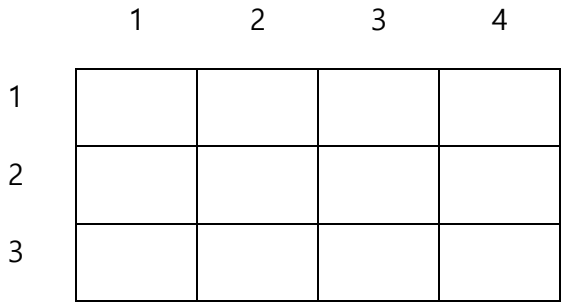
```
FOR Index <- 1 TO 5
    FOR Count <- 1 TO 3
        OUTPUT Numbers[Index]
    NEXT Count
NEXT Index
```

6.2 2-D Array

An array could be one-dimensional or two-dimensional



1D Array



2D Array

Declaring a 2D Array

Pseudocode declaration:

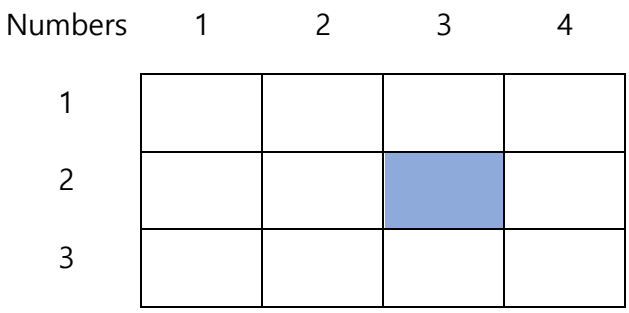
```
DECLARE <array_name>[1:rows, 1:columns] : <data_type>

//rows is the number of rows in the array and columns is the number of columns
```

Accessing an element in a 2D array

In a 2D array you must access the array using two indexes, one for the row number and another for the column number.

For example:



To output the highlighted element, we write the following statement

```
OUTPUT Numbers[2,3]
```

Example 6.2.1:

A computer science teacher has a class of 20 students, he has conducted 5 quizzes.

All the quizzes are out of 100.

Design a program that would read the marks of the 20 students in the 5 quizzes and store the marks in a 2-D array called `Marks[]`, calculate the average mark of each student and store it in 1-D array called `Averages[]`

```
DECLARE Marks[1:5, 1:20] : INTEGER
DECLARE Averages[1:20], StudentTotal : REAL
FOR Column <- 1 TO 20
    StudentTotal <- 0
    FOR Row <- 1 TO 5
        OUTPUT "Enter the student mark: "
        INPUT Marks[Row, Column]
        WHILE Marks[Row, Column] < 0
        OR Marks[Row, Column]> 100 DO
            OUTPUT "Invalid mark, try again"
            INPUT Marks[Row, Column]
        ENDWHILE
        StudentTotal <- StudentTotal + Marks[Row, Column]
    NEXT Row
    Average[Column] <- StudentTotal / 5
    OUTPUT "Average of student ", Column, " is ", Average[Student]
NEXT Column
```

Example 6.2.2:

Write an algorithm that would read the temperature in 3 different residential areas over a week in Cairo.

The program also should input the name of each residential area. Assume valid temperatures are between -10 and 55

The program should display the name of the area with the highest average temperature.

```
DECLARE Temperatures[1:3, 1:7] : REAL
DECLARE Names[1:3] : STRING
DECLARE Averages[1:3], TotalTemperature, MaximumTemperature : REAL
FOR Column <- 1 TO 3
    OUTPUT "Enter the name of the Column: "
    INPUT Names[Column]
    FOR Row <- 1 TO 7
        OUTPUT "Enter the temperature in ", Names[Column], " at Row ", Row
        INPUT Temperatures[Row, Column]
        WHILE Temperatures[Row, Column] < - 10
           OR Temperatures[Row, Column] > 55 DO
            OUTPUT "Invalid temperature, try again"
            INPUT Temperatures[Row, Column]
        ENDWHILE
    NEXT Row
NEXT Column
FOR Column <- 1 TO 3
    TotalTemperature <- 0
    FOR Row <- 1 TO 7
       TotalTemperature <- TotalTemperature + Temperatures[Row, Column]
    NEXT Row
    Averages[Column] <- TotalTemperature / 7
NEXT Column
```

Solution Continued:

```
MaximumTemperature <- Averages[1]
FOR Column <- 2 TO 3
    IF Averages[Column] > MaximumTemperature THEN
        MaximumTemperature <- Averages[Column]
    ENDIF
NEXT Column
OUTPUT "Name of Column with highest average temperature:"
FOR Column <- 1 TO 3
    IF MaximumTemperature = Averages[Column] THEN
        OUTPUT "Column : ", Names[Column]
    ENDIF
NEXT Column
```

Example 6.2.3:

The 1D array `Students[]` contains the names of students in a class. The 2D array `Marks[]` contains the mark for each quiz, for each student.
The position of each student's data in the two arrays is the same, for example, the student in position 10 in `Students[]` and `Marks[]` is the same.
The variable `StudentCount` contains the number of students in the class.
The variable `QuizNumber` contains the number of quizzes taken. All students took the same number of quizzes
Write a program that meets the following requirements:

- calculates the total mark for each student for all their quizzes
- output for each student:
    - name
    - total mark
- find and output the student name with the highest total mark
- find and output the student name with the lowest total mark

The arrays and variables have already been set up and the data stored.
You do **not** need to initialise the data in the array.

Solution:

```
DECLARE Totals[1:100] : INTEGER
// Loop over all the students
FOR Column <- 1 TO StudentCount
    // Initialize the total of each student to zero
    Totals[Column] <- 0
    // Loop over the marks of each student
    FOR Row <- 1 TO QuizNumber
        // Add each mark to the running total
        Totals[Column] <- Totals[Column] + Marks[Row, Column]
    NEXT Row
    // Output the name and the total mark
    OUTPUT "Name : ", Students[Column], " total mark : ", Totals[Column]
NEXT Column
DECLARE Maximum, Minimum : INTEGER
// Initialize the maximum to a very small number
Maximum <- -1
// Initialize the minimum to a very large number
Minimum <- 999999
// Loop over all the students to compare their totals
FOR Column <- 1 TO StudentCount
    // Compare the current student total to the maximum
    IF Totals[Column] > Maximum THEN
        Maximum <- Totals[Column]
    ENDIF
    // Compare the current student total to the minimum
    IF Totals[Column] < Minimum THEN
        Minimum <- Totals[Column]
    ENDIF
NEXT Column
```

```
// Loop over the students one more time to search for the student
// with the highest mark
OUTPUT "The student(s) with the highest total mark:"
FOR Column <- 1 TO StudentCount
    IF Totals[Column] = Maximum THEN
        OUTPUT Students[Column]
    ENDIF
NEXT Column
// Loop over the students to search for the lowest mark
OUTPUT "The student(s) with the lowest total mark:"
FOR Column <- 1 TO StudentCount
    IF Totals[Column] = Minimum THEN
        OUTPUT Students[Column]
    ENDIF
NEXT Column
```

7

Linear Search
&
Bubble Sort

7.1 Linear Search

An algorithm for finding an element within an array.

Example 7.1.1:

Write a pseudocode program that would read 50 elements from the user and stores them in an array. The program should output whether this array contains the value **5** or not

```
DECLARE Numbers[1:50] : REAL
FOR Index <- 1 TO 50
    OUTPUT "Enter a number:"
    INPUT Numbers[Index]
NEXT Index
DECLARE Found: BOOLEAN
DECLARE SearchValue : REAL
Found <- FALSE
SearchValue <- 5
FOR Index <- 1 TO 50
    IF Numbers[Index] = SearchValue THEN
        Found <- TRUE
    ENDIF
NEXT Index
IF Found THEN
    OUTPUT "Found the number 5 in the array"
ELSE
    OUTPUT "Did not find the number 5 in the array"
ENDIF
```

Example 7.1.2:

Write a pseudocode program that would read 50 elements from the user and stores them in an array.
The program then asks the user to enter a search value.
The program should display the last position of this element, if the search value is not found display **Not Found**

```
DECLARE Numbers[1:50] : REAL
FOR Index <- 1 TO 50
    OUTPUT "Enter a number:"
    INPUT Numbers[Index]
NEXT Index
DECLARE Position: INTEGER
DECLARE SearchValue : REAL
Position <- -1
OUTPUT "Enter a value to search for:"
INPUT SearchValue
FOR Index <- 1 TO 50
    IF Numbers[Index] = SearchValue THEN
        Position <- Index
    ENDIF
NEXT Index
IF Position <> -1 THEN
    OUTPUT "Found the number 5 at position ", Position
ELSE
    OUTPUT "The number 5 is not found"
ENDIF
```

Example 7.1.2:

An algorithm has been written in pseudocode to allow the names of **50** cities and their countries to be entered and stored in a two-dimensional (2D) array called `Country[]`

The array is already setup and the data is stored

Write a program that would

- allow the name of a **country** to be input
- display only the stored cities from that **country**.

```
DECLARE UserCountry : STRING
OUTPUT "Enter the country"
INPUT UserCountry
FOR Element <- 1 TO 50
    IF UserCountry = Country[Element, 2] THEN
        OUTPUT Country[Element, 1]
    ENDIF
NEXT Element
```

7.2 Bubble Sort

An algorithm for ordering elements within an array either ascending or descending.

Example 7.2.1:

The 1D array `Students[]` is used to store the names of 5 students.

The array is already setup, and the data is stored

Write a program that would:

- sort the names in ascending order.
- output the names in ascending order.

```
DECLARE Temp : STRING
// Loop over all the elements in the array
FOR Element <- 1 TO 5
    // Loop from 1 to 4
    FOR Count <- 1 TO 4
        // Compare the current element to the next one
        IF Students[Count] > Students[Count + 1] THEN
            Temp <- Students[Count]
            Students[Count] <- Students[Count + 1]
            Students[Count + 1] <- Temp
        ENDIF
    NEXT Count
NEXT Element
FOR Count <- 1 TO 5
    OUTPUT Students[Count]
NEXT Count
```

Example 7.2.2:

The 1D array `Marks[]` is used to store the marks of 10 students.

The array is already setup, and the data is stored

Write a program that would:

- sort the names in descending order.
- output the names in descending order.

```
DECLARE Temp : REAL
// Loop over all the elements in the array
FOR Element <- 1 TO 10
    // Loop from 1 to 9
    FOR Count <- 1 TO 9
        // Compare the current element to the next one
        IF Marks[Count] < Marks[Count + 1] THEN
            Temp <- Marks[Count]
            Marks[Count] <- Marks[Count + 1]
            Marks[Count + 1] <- Temp
        ENDIF
    NEXT Count
NEXT Element
FOR Count <- 1 TO 10
    OUTPUT Marks[Count]
NEXT Count
```

Example 7.2.3:

The 1D array `Marks[]` is used to store the marks of 10 students.
The 1D array `Students[]` is used to store the names of 5 students

The array are already setup, and the data is stored

Write a program that would:

- sort both arrays in descending order of the marks
- output the names and marks in descending order of marks

```
DECLARE TempMark : REAL
DECLARE TempName : STRING
// Loop over all the elements in the array
FOR Element <- 1 TO 10
    // Loop from 1 to 9
    FOR Count <- 1 TO 9
        // Compare the current element to the next one
        IF Marks[Count] < Marks[Count + 1] THEN
            // Swap the marks
            TempMarks <- Marks[Count]
            Marks[Count] <- Marks[Count + 1]
            Marks[Count + 1] <- TempMarks
            // Swap the name
            TempName <- Students[Count]
            Students[Count] <- Students[Count + 1]
            Students[Count + 1] <- TempName
        ENDIF
    NEXT Count
NEXT Element
FOR Count <- 1 TO 10
    OUTPUT Students[Count]
    OUTPUT Marks[Count]
NEXT Count
```

Example 7.2.4:

The 2D Array `Contacts[]` stores the full name and phone number of 10 different people.

The array is already setup and the data is stored

Write a program that would sort the names in ascending order of names.

```
DECLARE TempName, TempPhone : STRING
FOR Element <- 1 TO 10
    FOR Index <- 1 TO 9
        IF Contacts[Index, 1] > Contacts[Index + 1, 1] THEN
            TempName <- Contacts[Index, 1]
            Contacts[Index, 1] <- Contacts[Index + 1, 1]
            Contacts[Index + 1, 1] <- TempName

            TempPhone <- Contacts[Index, 2]
            Contacts[Index, 2] <- Contacts[Index + 1, 2]
            Contacts[Index + 1, 2] <- TempPhone
        ENDIF
    NEXT Index
NEXT Element
```

8

Functions & Procedures

8. Functions and Procedures

8.1. Library Routines

1. ROUND(REAL,INTEGER)

Purpose:

- To return a value rounded to a specified number of digits / decimal places
- The result will either be rounded to the next highest or the next lowest value
- … depending on whether the value of the preceding digit is >=5 or <5
- Example of `ROUND` for example, `ROUND(4.56, 1) = 4.6`

Example 8.1.1:

Write a program that reads a number from the user and rounds it to the nearest whole number.

```
DECLARE Number : REAL
OUTPUT "Enter a number"
INPUT Number
OUTPUT "Rounded to the nearest whole number:", ROUND(Number, 0)
```

2. INT(REAL)

Purpose:

Returns the value of a number discarding the fractional part.

Example 8.1.2:

Write a program that reads a number from the user and output only the integer part.

```
DECLARE Number : REAL
OUTPUT "Enter a number"
INPUT Number
OUTPUT "No decimal:", INT(Number)
```

3. RANDOM()

Purpose:

- To generate (pseudo) random numbers
- ...(usually) within a specified range
- Allow example e.g. `RANDOM() * 10` returns a random number between `0` and `10`.

Example 8.1.3:

Write a program that displays a random number between 0 and 100

```
OUTPUT RANDOM() * 100
```

4. DIV(Number1,Number2)

Purpose:

- To perform integer division of two numbers..
- .. with the fractional part discarded
- Example of `DIV` For example `DIV(9,4) = 2`

Important:

DIV can be used in 2 ways:

- `Result <- Number1 DIV Number2`
- `Result <- DIV(Number1,Number2)`

Example 8.1.4:

Write a program that reads 2 numbers and displays the integer division of the first number divided by the second number

```
DECLARE Number1,Number2 : REAL
OUTPUT "Enter two numbers"
INPUT Number1, Number2
Result <- DIV(Number1,Number2)
OUTPUT "Integer division = ", Result
```

5. MOD(Number1,Number2)

Purpose:

- To perform (integer) division when one number is divided by another
- .. and find the remainder
- Example: `7 MOD 2 = 1`

Important:

MOD can be used in 2 ways:

- `Result <- Number1 MOD Number2`
- `Result <- MOD(Number1,Number2)`

Example 8.1.5:

Write a program that reads 2 numbers and displays the remainder of the division of the first number divided by the second number

```
DECLARE Number1,Number2 : REAL
OUTPUT "Enter two numbers"
INPUT Number1, Number2
Result <- MOD(Number1,Number2)
OUTPUT "Remainder= ", Result
```

Procedure

A subroutine that does not return a value

Calling a procedure

```
CALL Display(5)
CALL Display(10)
```

Function

Calling a function

```
X <- CALL Multiply(10)
Y <- CALL Multiply(34)
```

OR

```
OUTPUT Multiply(10)
OUTPUT CALL Multiply(10)
```

Differences between function and procedure:

- Procedures cannot **return** a value
- While functions must **return** a value

Example 1:

Description	Function/Procedure
The routine `INT(X)` takes a real number `X` and **returns** the whole number part of the number.	Function
The routine `RangeCheck(X,Y,Z)` takes a number `X` to validate, the lower bound `Y` and the upper bound `Z` and **performs the range check validation**	Procedure
The routine `LengthCheck(X,Y)` takes a string `X` and a maximum number of letters `Y` and performs a length check validation.	Procedure
The routine `ROUND(X,Y)` returns the value of `X` rounded to a number of decimal places `Y`.	Function

Example 2:

The procedure `DisplayLength(X)` outputs the length of a string `X`.

Write a program that would:

- input a name as input
- output the number of characters in the name

Solution:

```
DECLARE Name : STRING
OUTPUT "Enter your name"
INPUT Name
DisplayLength(Name)
```

Example 3:

The function `LENGTH(X)` finds the length of a string `X`.

Write a program that would:

- input a name as input
- output the number of characters in the name

```
DECLARE Name : STRING
OUTPUT "Enter your name"
INPUT Name
OUTPUT LENGTH(Name)
```

Function/Procedure Definition

Definition:

- Setting-up the function/procedure
- Includes defining the:
- Function/Procedure name
    - Parameter names and data types
    - In case of a function, the return data type
- Done only once

How to define a procedure:

```
PROCEDURE Name(Params)
    // Code to perform when the procedure is called
ENDPROCEDURE
```

Example 4:

The procedure `RangeCheck(Value,LowerBound,UpperBound)` takes a number `Value` to validate, the lower bound `LowerBound` and the upper bound `UpperBound`.

The procedure checks that `Value` is between `LowerBound` and `UpperBound` and if not repeatedly display **Invalid** and ask the user to re-enter the number `Value`.

Write a program that would:

- define the procedure `RangeCheck(Value,LowerBound,UpperBound)`

```
PROCEDURE RangeCheck(Value : REAL, LowerBound : REAL, UpperBound : REAL)
    WHILE Value < LowerBound OR Value > UpperBound DO
        OUTPUT "Invalid value, try again"
        INPUT Value
    ENDWHILE
ENDPROCEDURE
```

Example 5:

The procedure `Display(Message: STRING, Number : INTEGER)` that would output a message a number of times equivalent
Then the program should:

- define the procedure `Display(Message: STRING, Number : INTEGER)`
- output the message `"Hello World"` 5 times
- output the message `"Welcome to CSTeam"` 2 times
- output the message `"Bye Bye"` 4 times

```
PROCEDURE Display(Message : STRING, Number: INTEGER)
    FOR Count <- 1 TO Number
        OUTPUT Message
    NEXT
ENDPROCEDURE
Display("Hello World", 5)
Display("Welcome to CSTeam", 2)
Display("Bye Bye", 4)
```

Example 5:

The function `Min(A,B)` takes 2 whole numbers and **returns** the smaller number.

(a) Write pseudocode statements to define the function

```
FUNCTION Min(A : INTEGER, B: INTEGER) RETURNS INTEGER
    IF A < B THEN
        RETURN A
    ELSE
        RETURN B
    ENDIF
ENDFUNCTION
```

(b) Call the function with call the function with 5, 8 and store the return value in Z

```
Z <- Min(5,8)
```

Example 6:

The function `Equals(A : REAL, B : REAL)` returns `"Equal"` if `A` is equal to `B` and returns `"Different"` otherwise

Write a program that would:

- define the function `Equals(...)`
- input two numbers
- call the function `Equals(...)` with those numbers and display the return value

```
FUNCTION Equals(A : REAL, B: REAL) : REAL
    IF A = B THEN
        RETURN "Equal"
    ELSE
        RETURN "Different"
    ENDIF
ENDFUNCTION
DECLARE Num1, Num2 : REAL
OUTPUT "Enter two numbers"
INPUT Num1, Num2
OUTPUT CALL Equals(Num1, Num2)
```

Function/Procedure Call

Definition:

- Using/executing the function code with parameters values
- Can be done multiple times

Difference between function/procedure definition and function/procedure call

- Defining is done once, calling can be done multiple times
- Defining is setting up the function/procedure and calling is using the function

What happens when a function is called during the execution of a program

- A call statement is used in order to make use of a function
- Parameters are be passed from the main program to the function
- The function performs its task..
- ..and returns a value to the main program

Local Variable

Definition:

- A variable that is defined inside the scope of a subroutine (function/procedure)
- Its scope is restricted to only the part of the program where it was defined
- Can be accessed only within the scope of the subroutine (function/procedure) where it was defined

Global Variable

Definition:

- A variable that is defined outside the scope of a subroutine (function/procedure)
- Its scope covers the whole program
- Can be accessed from outside any subroutine and from inside the scope of a subroutine

9

File Handling

9. File Handling

Purpose of using files

- Data is not lost when the computer is switched off (data is stored permanently)
- Data can be used by more than one program or reused when a program is run again
- Data can be transported from one computer to another.

Using files

To use file, we go through 3 steps

1. Open the file (with a mode of operation)
2. Read from or write to the file
3. Close the file

OPENFILE

Opens the file to prepare it either for reading or writing

Syntax for opening a file

```
OPENFILE [FILE_NAME] FOR [MODE OF OPERATION]
```

Modes of operation:

- **READ** – for data to be read from the file
- **WRITE** - for data to be written to the file. A new file will be created and any existing data in the file will be lost

READFILE

Reads a **line** from the file and assigns to a variable

Syntax for reading from a file

```
READFILE [FILE_NAME], [VARIABLE_NAME]
```

WRITEFILE

Writes a **line** (string) to the file

Syntax for writing to a file

```
WRITEFILE [FILE_NAME], [VARIABLE_NAME]
```

CLOSEFILE

Closes a file

Syntax for closing a file

```
CLOSEFILE [FILE_NAME]
```

Example 1:

Write a program that would copy a line of text from FileA.txt to FileB.txt

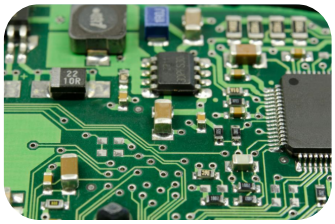
```
DECLARE LineOfText : STRING
OPENFILE "FileA.txt" FOR READ
OPENFILE "FileB.txt" FOR WRITE
READFILE "FileA.txt", LineOfText
WRITEFILE "FileB.txt", LineOfText
CLOSEFILE "FileA.txt"
```

10

TEAM

Digital Logic

Introduction



Why is data stored as binary in computers?

- Computers use logic gates
- Logic gates use one of two values: 0 or 1

Definition of a Logic gate:

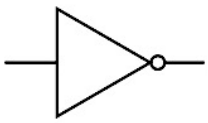
A small physical device that controls the flow of electrical signals in a pre-determinant way.

Purpose of a logic gates:

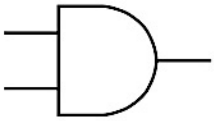
- To carry out a logical operation
- To control the flow of electricity through a logic circuit
- To alter the output from given inputs

1. Types of logic gates

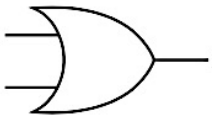
Six different logic gates are considered:



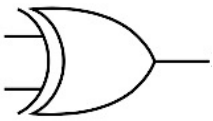
NOT gate



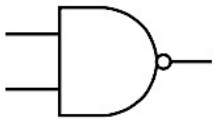
AND gate



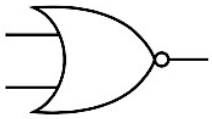
OR gate



XOR gate



NAND gate



NOR gate

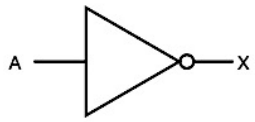
The behavior of each logic gate can be presented by constructing a truth table.

KEY TERM

Truth table – A table is used to trace the output from a logic gate or a logic circuit considering all possible combinations of 0s and 1s that can be input.

2.1 NOT gate

How to draw it:



Truth table:

Input	Output
A	X
0	1
1	0

How to write it:

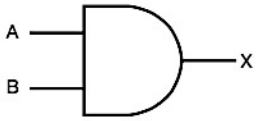
X = NOT A

Explanation:

- It has one input
- Output will be 1 if the input is 0
- Output will be 0 if the input is 1

2.2 AND gate

How to draw it:



Truth table:

Input	Input	Output
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

How to write it:

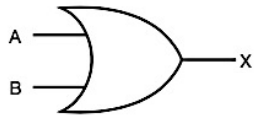
X = A AND B

Explanation:

- It has two inputs
- Output will be 1 if both inputs are 1
- Output will be 0 if either input is 0

2.3 OR gate

How to draw it:



Truth table:

Input	Input	Output
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

How to write it:

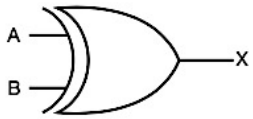
X = A OR B

Explanation:

- It has two inputs
- Output will be 1 if either input is 1
- Output will be 0 if both inputs are 0

2.4 XOR gate

How to draw it:



Truth table:

Input	Input	Output
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

How to write it:

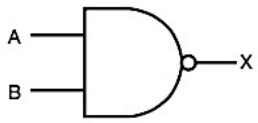
X = A XOR B

Explanation:

- It has two inputs
- Output will be 1 if both inputs are different
- Output will be 0 if both inputs are 1
- Output will be 0 if both inputs are 0

2.5 NAND gate

How to draw it:



How to write it:

X = A NAND B

Explanation:

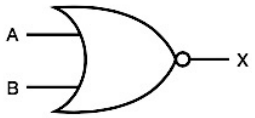
- It has two inputs
- Output will be 1 if either input is 0
- Output will be 0 if both inputs are 1

Truth table:

Input	Input	Output
A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

2.6 NOR gate

How to draw it:



How to write it:

X = A NOR B

Explanation:

- It has two inputs
- Output will be 1 if both inputs are 0
- Output will be 0 if either input is 1

Truth table:

Input	Input	Output
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

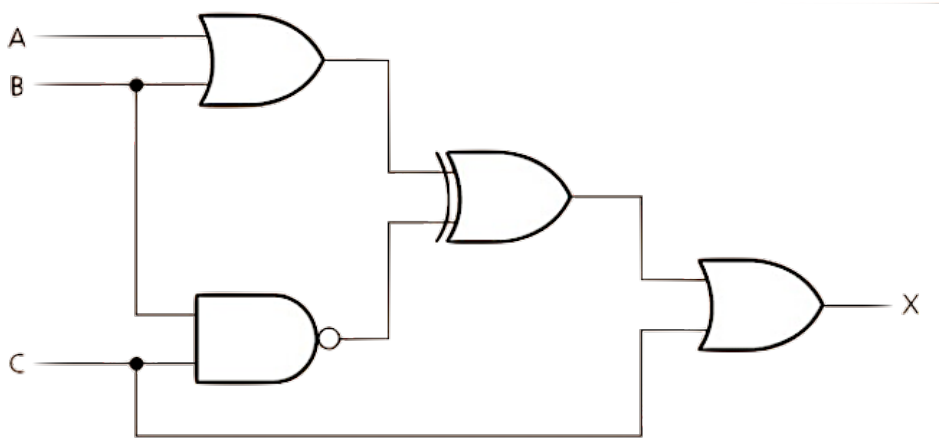
2. Logic circuits

Definition:

A combination of logic gates to carry out a particular function.

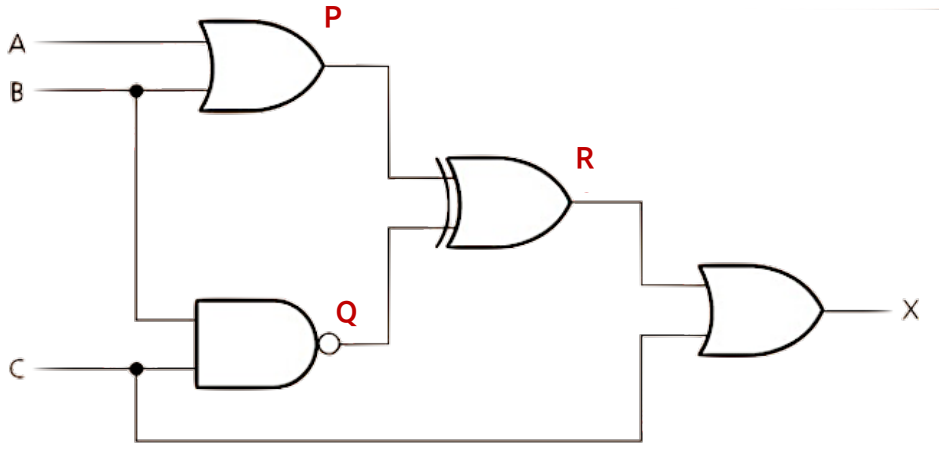
3.1 Produce truth table for a logic circuit

Produce the truth table for the following logic circuit:



Solution:

1 Name all intermediate logic gates



2 Write down the logic statements for all intermediate gates in addition to the output:

P = A **OR** B

Q= B **NAND** C

R = P **XOR** Q

X (Output) = R **OR** C

3 Draw the truth table (this logic circuit has 3 inputs which means that there are $2^3 = 8$ possibilities):

Input			Working Space	Output
A	**B**	**C**		**X**
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

4 Insert the logic statements of the intermediate gates in the truth table:

Input			Working Space			Output
A	B	C	P = (A OR B)	Q = (B NAND C)	R = (P XOR Q)	X
0	0	0	0	1	1	
0	0	1	0	1	1	
0	1	0	1	1	0	
0	1	1	1	0	1	
1	0	0	1	1	0	
1	0	1	1	1	0	
1	1	0	1	1	0	
1	1	1	1	0	1	

5 Complete the truth table by filling the output X column:

Input			Working Space			Output
A	B	C	P = (A OR B)	Q = (B NAND C)	R = (P XOR Q)	X = (R OR C)
0	0	0	0	1	1	1
0	0	1	0	1	1	1
0	1	0	1	1	0	0
0	1	1	1	0	1	1
1	0	0	1	1	0	0
1	0	1	1	1	0	1
1	1	0	1	1	0	0
1	1	1	1	0	1	1

Write the logic statement for the logic circuit:

1 Recall that the logic statements for all intermediate gates are:

P = A **OR** B

Q= B **NAND** C

R = P **XOR** Q

X (Output) = R **OR** C

2 Write the output (X) in terms of the input values (A, B and C):

- Expand R to P XOR Q

  X = (P **XOR** Q) **OR** C

- Expand both P and Q

  X = ((A **OR** B) **XOR** (B **NAND** C)) **OR** C

3.2 Design a logic circuit from a logic statement

Draw the logic circuit and the truth table for the following logic statement:

X = 1 if ((A is 1 OR B is 1) AND (A is 1 AND B is 1)) OR (C is NOT 1)

Solution:

Scan the following QR code to watch an elaborative video of the answer:

3.3 Wording Problems

Example:

A wind turbine has a safety system, which uses three inputs to a logic circuit. A certain combination of conditions results in an output, X, from the logic circuit being equal to 1. When the value X=1 then the wind turbine is shut down.

The following table shows which parameters are being monitored and form the three inputs to the logic circuit:

Parameter description	Parameter	Binary value	Description of condition
turbine speed	S	0	<= 1000 rpm
		1	> 1000 rpm
bearing temperature	T	0	<= 80°C
		1	> 80°C
wind velocity	W	0	<= 120 kph
		1	> 120 kph

The output X, will have the value of 1 if any of the following combination of conditions occur:

- either turbine speed <=1000 rpm and bearing temperature > 80°C
- or turbine speed > 1000 rpm and wind velocity > 120 kph
- or bearing temperature <=80°C and wind velocity > 120 kph

Design the logic circuit and complete the truth table to produce a value of X=1 when any of the three conditions above occur.

Solution:

Scan the following QR code to watch an elaborative video of the answer:

3.4 Design a logic circuit from a truth table

Example:

Use the truth table below to draw the logic circuit

Input		Output
A	**B**	**X**
0	0	0
0	1	0
1	0	1
1	1	0

1 Look for the rows where the output is **1**

Input		Output
A	**B**	**X**
0	0	0
0	1	0
1	**0**	**1**
1	1	0

This means X = 1 if (A = 1 AND B = 0)

2 Turn this into a logic statement

X = (A AND NOT B)

Example:

Use the truth table below to draw the logic circuit

Input			Output
A	**B**	**C**	**X**
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

1 Look for the rows where the output is **1**

Input			Output
A	**B**	**C**	**X**
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	1	1	0

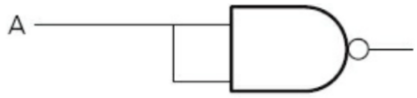
X = 1 if (A = 0 AND B = 0 AND C = 0)
or (A = 1 AND B = 0 AND C = 0)
or (A = 1 AND B = 1 AND C = 0)

2 Turn this into a logic statement

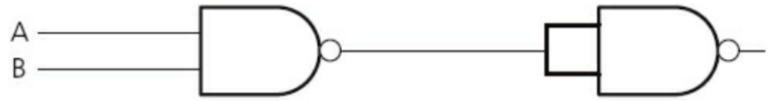
X = (NOT A AND NOT B AND NOT C)
or (A AND NOT B AND NOT C)
or (A AND B AND NOT C)

3.5 Reducing a logic circuit to a single logic gate

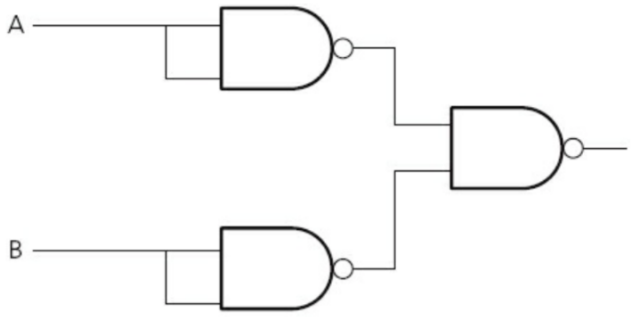
The following circuit behaves the same as NOT gate:



The following circuit behaves the same as AND gate:



The following circuit behaves the same as OR gate:



11

Databases

1. Data types used in database

Data Type	Examples
Integer	12, 45, 1274, 653, -35
Real	12.45, -0.01, 999.00, 500
Character	A,b,0,-,+,?
Text	DOG, ABC123, enquiries@bbc.co.uk
Boolean	TRUE/FALSE and YES/NO
Date	25/10/2007, 12 Mar 2008, 10-06-08

2. Selecting Data Types

When we input data to a database, we must analyze it and select appropriate data types for each field:

Student Name:	Ben Smith	**Text**
Student Number:	1234	**Integer**
Date of Birth:	10 July 1998	**Date**
Year Group:	6	**Integer**
Special Diet:	Yes	**Boolean**
Height:	1.67	**Real**
Fees Paid:	$ 1500	**Real**

3. Definition of Database

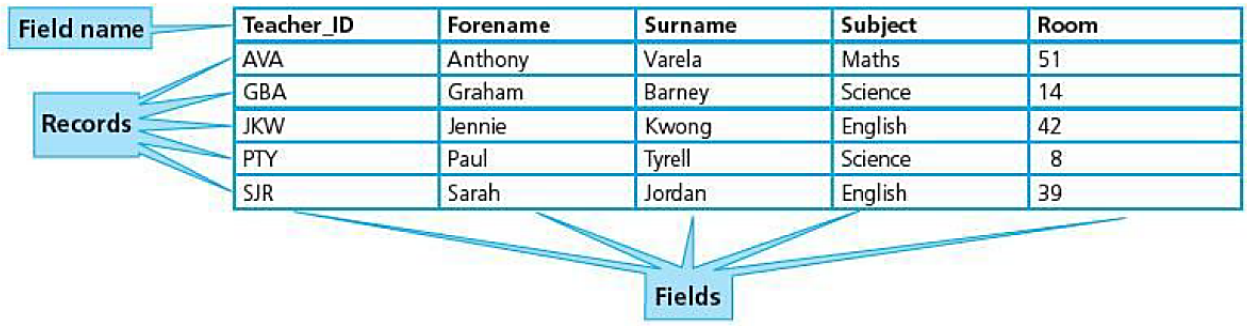
A structured collection of related data that allows people to extract information in a way that meets their needs.

Databases are very useful in preventing data problems occurring because:

- Data is only stored once - no data duplication
- If any changes are made it only has to be done once - the data is consistent
- Same data is used by everyone

4. The structure of a database

Inside a database, data is stored in **tables**, which consists of many **fields** and **records**.



Teacher_ID	Forename	Surname	Subject	Room
AVA	Anthony	Varela	Maths	51
GBA	Graham	Barney	Science	14
JKW	Jennie	Kwong	English	42
PTY	Paul	Tyrell	Science	8
SJR	Sarah	Jordan	English	39

The following table shows some database elements and their definitions:

Term	Definition	Examples
Table	A collection of related records	Table of students
Field	A column that contains one specific piece of information and has one data type	For a student the fields could include: • Student name • Student ID • Age • Address • Gender
Field Name	Title given to each field	
Record	A row within a table that contains data about a single item, person or event	Details (all information) of one student
Primary Key	A field that contains unique data and can't have duplicates	Student ID field

5. Query using Standard Query Language (SQL)

In order to select data with some specific criteria from a database, we use **queries**.

Let's consider the following example:

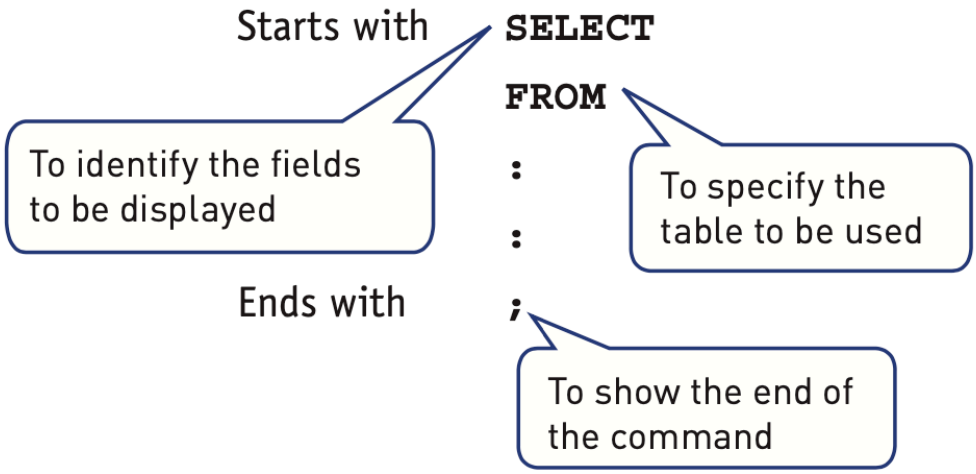
A database table, `2018MOV`, is used to keep record of movie details

CatNo	Title	Genre1	Genre2	Blu-ray	DVD	Streaming
18m01	Power Rangers	Adventure	Fantasy	Yes	No	Yes
18m02	Baywatch	Comedy	Drama	Yes	No	Yes
18m03	Table 19	Comedy	Drama	Yes	Yes	No
18m04	Wonder Woman	Action	Fantasy	Yes	No	Yes
18m05	Justice League	Action	Fantasy	Yes	Yes	Yes
18m06	Twilight	Thriller	Action	Yes	Yes	No
18m07	Ant Man	Action	Fantasy	No	Yes	No
18m08	Venice Beach	Action	History	No	Yes	No
18m12	Fast Five	Action	Thriller	No	Yes	No
18m15	King Kong	Adventure	Fantasy	No	Yes	No
18m16	Transformers: The Last Knight	Action	Sci-Fi	Yes	Yes	Yes
18m17	The Dark Tower	Fantasy	Sci-Fi	Yes	Yes	No
18m19	Beauty and the Beast	Fantasy	Romance	Yes	Yes	Yes
18m21	The Mummy	Action	Fantasy	No	No	Yes
18m22	Star Wars: Episode VIII	Sci-Fi	Action	Yes	No	Yes
18m23	Guardians of the Galaxy	Action	Sci-Fi	Yes	Yes	Yes
18m26	Thor	Action	Sci-Fi	No	Yes	Yes
18m27	Twilight	Fantasy	Sci-Fi	No	No	Yes
18m30	Beneath	Action	Fantasy	Yes	No	No
18m31	Despicable Me	Animation	Action	Yes	Yes	No

SQL Statements

- `SELECT`
  - Specifies the fields to return from the query (queries always begin with **`SELECT`**)
- `FROM`
  - Specifies the table to use
- `WHERE`
  - Includes only records (rows) in a query that match a given condition.
- `ORDER BY`
  - Sorts the results from a query by a given column either alphabetically or numerically (DESC for sorting the data descendingly)
- `COUNT`
  - Counts the number of records for a certain field
- `SUM`
  - Totals the values for a certain field

SQL Statement structure:



Example 1:

Complete the structured query language (SQL) to return the category number and title for all Action movies sorted in ascending order of the title.

```
SELECT CatNo, Title

..... 2018MOV

WHERE Genre1 = .....

ORDER BY .....;
```

Answer:

```
SELECT CatNo, Title

FROM 2018MOV

WHERE Genre1 = "Action"

ORDER BY Title;
```

IMPORTANT

SELECT * returns all the fields in the table

Example 2:

Complete the structured query language (SQL) to count the number of Action Movies.

```
SELECT .....

..... 2018MOV

WHERE Genre1 = .....;
```

Answer:

```
SELECT Count(*)

FROM 2018MOV

WHERE Genre1 = "Action";
```

Example 3:

Change the previous query to count the number of action movies that are available for streaming

Answer:

```
SELECT Count(*)

FROM 2018MOV

WHERE Genre1 = "Action" AND Streaming = Yes;
```

6. Validation

Definition:

The automated checking by a program that data to be entered is **sensible**.

Examples of validation checks:

- Type
- Presence
- Range
- Length
- Format
- Check digit

6.1 Type Check

Definition:

Checks that the data entered has the appropriate data type.

Example:

A person's age should be a number not text.

6.2 Presence Check

Definition:

Checks that some data has been entered and the value has not been left blank.

Example:

An email address must be given for an online transaction.

6.3 Range Check

Definition:

It checks that only numbers within a specified range are accepted.

Example:

Exam marks between 0 and 100 inclusive.

6.4 Length Check

Definition

It checks that data contains an **exact number of characters**

Example:

A password with exactly 8 characters in length. If the password has fewer or more than 8 characters, it will be rejected.

6.5 Format Check

Definition:

It checks that the characters entered conform to a pre-defined pattern.

Example:

A company with IDs that start with MS then three numbers would have a format of MS###

6.6 Check Digit

Definition:

- A digit that it is calculated from all the other digits and then appended to the number
- Used to check whether data is correct **when entered into the system**

7. Data Verification

Definition:

Checking that the data has not changed during input to a computer or during transfer between computers.

Purpose:

To ensure the accuracy of transcription.

Verification methods include:

- Screen/Visual check
- Double entry

Screen/Visual Check

Definition:

- Data is compared **visually** with the source document by a user
- The user is asked to confirm that the data entered is same as original
- If user finds a difference, the data is re-entered

Double Data Entry

Definition:

- Data is entered twice (sometimes by two different people)
- A computer checks that both entries are equal
- If they are not equal, an error message requesting to re-enter the data is displayed

IMPORTANT

Double data entry is different from **Proofreading**.
Proofreading is reading through the document, without referring to the original source document, to identify grammatical and spelling mistakes.

Pseudocode for each validation check

1. Presence Check

```
OUTPUT "Enter a word"
INPUT Word
// Validate using presence check using WHILE Loop
// Check if the word is an empty string
WHILE Word = "" DO
    OUTPUT "Invalid, word has to contain a value, try again"
    INPUT Word
ENDWHILE
```

2. Type Check

```
// A number is an integer
// An algorithm to validate the type of a number
DECLARE Number : REAL
OUTPUT "Enter a number"
INPUT Number
WHILE Number <> INT(Number) DO
    OUTPUT "Invalid number, it must be a whole number"
    INPUT Number
ENDWHILE
```

3. Length Check

```
// An algorithm to validate that a password is at least 8 characters
DECLARE Password :STRING
OUTPUT "Enter a password:"
INPUT Password
WHILE LENGTH(Password) < 8 DO
    OUTPUT "Invalid password, try again:"
    INPUT Password
ENDWHILE
```

4. Range Check

```
// An algorithm to validate that a number is between 1 and 100 inclusive
DECLARE Number: REAL
OUTPUT "Enter a number"
INPUT Number
WHILE Number < 1 OR Number > 100 DO
    OUTPUT "Invalid number, try again:"
    INPUT Number
ENDWHILE
```

5. Format Check

```
// An algorithm to perform a format check
// Making sure that a username starts with N23-
// followed by any 3 characters
// Format : N23-CCC
DECLARE UserName : STRING
OUTPUT "Enter a username:"
INPUT UserName
WHILE SUBSTRING(UserName, 1, 4) <> "N23-" OR LENGTH(UserName) <> 7 DO
    OUTPUT "Invalid username, try again:"
    INPUT UserName
ENDWHILE
```

6. Check Digit

Example 1:

Write an algorithm that calculates the check-digit of a 4-digit number

```
// An algorithm to calculate the check-digit
// Multiply each digit by its position
// Add those results
// Divide them by 10
// The remainder would be the check-digit
DECLARE Number[1:5] : REAL
DECLARE Total : REAL
Total <- 0
FOR Count <- 1 TO 4
    OUTPUT "Enter the digit:"
    INPUT Number[Count]
    WHILE Number[Count] < 0 OR Number[Count] > 9 DO
        OUTPUT "This is not a single digit, try again:"
        INPUT Number[Count]
    ENDWHILE
    Total <- Total + Number[Count] * Count
NEXT
Number[5] <- Total MOD 10
FOR Count <- 1 TO 5
    OUTPUT Number[Count]
NEXT
```

Example 2:

Write an algorithm to make sure that check-digit is correct

```
// An algorithm to check if the data entry is valid
// using check-digit
DECLARE Number[1:5] : REAL
DECLARE Total : REAL
DECLARE CalculatedCheckDigit : INTEGER
Total <- 0
FOR Count <- 1 TO 5
    OUTPUT "Enter the digit at ", Count
    INPUT Number[Count]
    WHILE Number[Count] < 0 OR Number[Count] > 9 DO
        OUTPUT "Invalid, this is not a single digit, try again"
        INPUT Number[Count]
    ENDWHILE
NEXT
FOR Count <- 1 to 4
    Total <- Total + Number[Count] * Count
NEXT
CalculatedCheckDigit <- Total MOD 10
IF CalculatedCheckDigit = Number[5] THEN
    OUTPUT "Check digit is valid"
ELSE
    OUTPUT "Check digit is not valid"
ENDIF
```

12

Structure Diagrams

12. Structure Diagrams

Top-down design

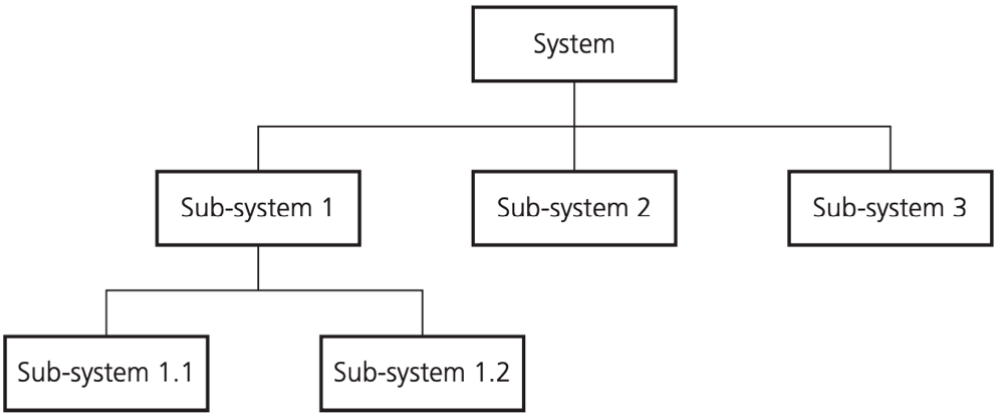
Definition:

The decomposition of a computer system into a set of sub-systems, then breaking each subsystem down into a set of smaller sub-systems, until each sub-system just performs a single action

Structure Diagram

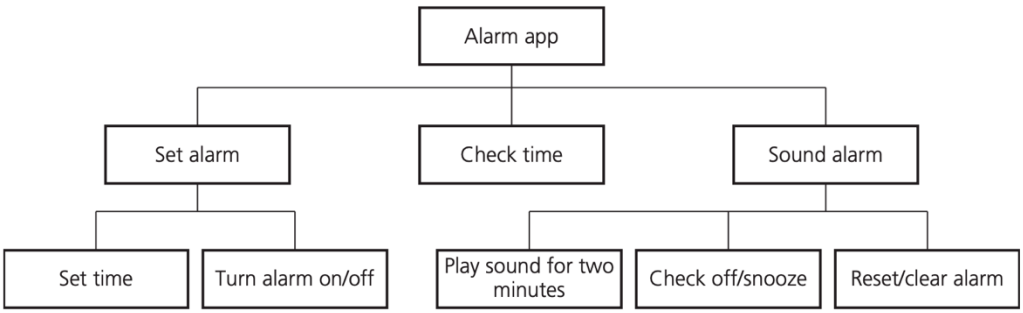
Definition:

A diagram that shows the design of a computer system in a hierarchical way, with each level giving a more detailed breakdown of the system into subsystems



Example 1:

Draw a structure diagram for an alarm app that allows the user to set an alarm by setting the time and turn the set alarm on/off. The alarm should check the time every now and then. If the time of the alarm has been reached the app sounds an alarm by playing a sound for two minutes and check off the snooze and clear the alarm



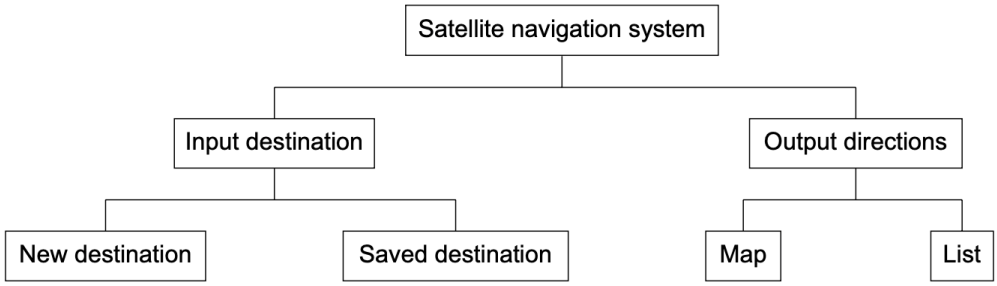
Example 2:

A satellite navigation system is an example of a computer system that is made up of sub-systems.

Part of a satellite navigation system:

- Allows the user to enter details for a new destination or select a previously saved destination
- displays directions in the form of a visual map or as a list.

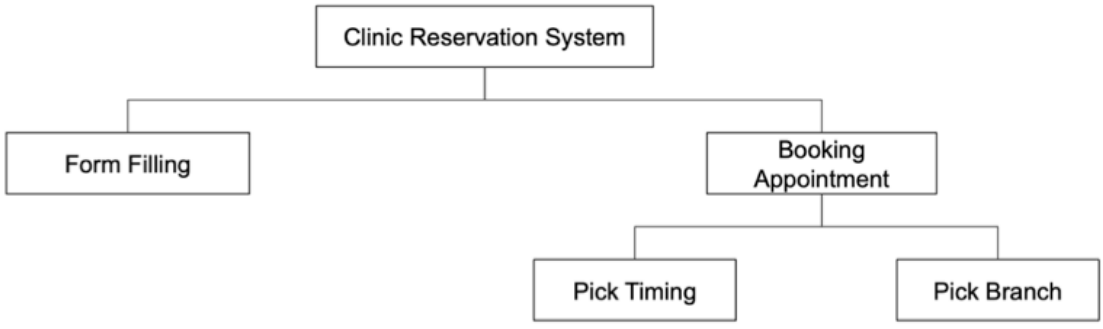
Draw a structure diagram for this part of the satellite navigation system.



Example 3:

A dentist creates a system for his clinic. He requires the patients to register their information by filling a form. The patient can then book an appointment by choosing the time and branch for the clinic.

Draw the structure diagram of this system.



13

Definitions

General Programming Terms

1. Program

Definition:

- A list of instructions that does a specific task
- It can be written in high-level language or low-level language

2. Pseudocode

Definition:

Method that uses English words and mathematical notations to design the steps of a program.

3. Flowcharts

Definition:

Diagram that designs the steps of a program by using a standard set of symbols joined by lines to show the direction of flow.

4. Trace Table

Definition:

- Table that can be used to record the results from each step in an algorithm
- It is used to record the value of a variable each time it changes
- Used to document the dry run of program

5. Tracing (Dry Run)

The manual exercise of working through an algorithm step by step

6. Variable

Definition:

A named data store that contains a value that can change during the execution of a program.

7. Constant

Definition:

A named data store that contains a value that does not change during the execution of a program.

8. Comments

Definition:

Blocks of text in the pseudocode that are not executed used to explain the code.

9. Features of a maintainable program

- Use comments …
- … to explain the purpose of each section of code
- Use meaningful identifier names to …
- … clearly identify the purpose of variables, constants, arrays, procedures and functions
- Use procedures and functions …
- … to avoid repeated code
- … and to simplify logic
- Use indentation and white space …
- … to make the program readable

10. Component parts after decomposition

- **inputs** – the data that needs to be entered to the system
- **processes** – the tasks that need to be performed using the input data and any previously stored data
- **outputs –** information that needs to be displayed or printed for the users
- **storage –** data that needs to be stored in files for use in the future

11. Top-Down design

Definition:

The breaking down of a computer system into set of sub systems and then breaking each subsystem into set of smaller ones until each sub system performs a single action.

Stepwise refinement definition:

Process of breaking down into smaller sub-systems

Module definition:

An individual section of code that can be used by other programs

12. Structure Chart // Structure Diagram

Definition:

Diagram that shows the design of a computer system in a hierarchal way, with each level giving a more detailed breakdown of the system.

Purpose:

- To provide an overview of the system hierarchy
- To provide overview of how a problem is broken down

13. Ways to represent an algorithm

- Flowcharts
- Pseudocode
- Structure Diagrams

14. Basic concepts (constructs) to use when developing a program:

- **Data use** – variables, constants and arrays
- **Sequence** – order of steps in a task
- **Selection** – choosing a path through a program
- **Iteration** – repetition of a sequence of steps in a program
- **Operator use** – arithemtic operations for calculation and logical and boolean operators for decisions

Program Development Lifecycle

15. Stages of Program Develeopment Lifecycle

Analysis

- Identification of the problem and requirements
- May include breaking down the problem into smaller parts
- Includes abstractions (removal of details)

Design

- Drawing flowcharts to solve the problem
- Writing pseudocode statement to solve the problem
- Decomposing the problem into smaller parts using structure diagrams

Coding

- Writing program code solutions (using a programming language)
- May include early testing

Testing

- Testing the program code using test data

Tasks performed in analysis:

- abstraction
- decomposition
- identification of the problem and requirements (requirement specification)

Abstraction Definition:

- simplifying the problem
- selecting elements required
- filtering out irrelevant characteristics from those elements

Decomposition Definition:

- breaking down a complex problem into smaller parts
- which can then be divided into even smaller parts that can be solved easily

Tasks performed in design:

- decomposition
- structure diagrams
- flowcharts
- pseudocode

Tasks performed in coding:

- writing program code
- iterative testing

Iterative Testing Definition:

- tests are conducted to each sub-system of the program
- errors are corrected
- tests are repeated until the sub-system performs as required

Tasks performed in testing:

- testing program code…
- … using test data

Testing Definition:

- The completed program or set of programs is run many times…
- …with different sets of test data
- …to ensure all the completed tasks work together as specified in the program design

16. Operators

Operators definition and purpose:

A special character or word in programming that identifies an action to be performed

Arithmetic Operators Definition and purpose:

Operators that are used to perform calculations.

Logical Operators Definition and purpose:

Operators that allow the computer to decide whether an expression is true or false

Boolean Operators Definition and purpose:

Operators that are used that is used with logical operators to form a more complex expression.

17. Program data types:

Data type	Definition	Examples
Integer	A positive or negative whole number be used in mathematical operations	150 -100
Real	A positive or negative number that has a fractional part that can be used in mathematical operations	100.5 -15.2
Char	A single character from the keyboard	H
String	A sequence of characters	Hello World A312_@odq
Boolean	Data with two possible values	TRUE/FALSE

18. Sequential statements

Definition:

Statements are executed one after another according to their order.

19. Assignment Statement

- A statement that changes the value of a variable or an element in an array
- Example: `Value <- 39` sets the value of the variable `Value` to `39`

20. Prompt Message

Definition:

An output message that is displayed before the user input stating the input required

Purpose:

Helps the user to know what they are expected to input

21. Conditional/Selection statements

Definition:

Selective statement(s) are executed depending on meeting certain criteria.

Purpose:

To allow different paths through a program depending on meeting certain criteria.

22. IF .. THEN .. ELSE .. ENDIF

Definition:

A conditional structure with different outcomes for true and false.

Purpose:

It allows for checking complex conditions.

23. CASE .. OF .. OTHERWISE .. ENDCASE

Definition:

A conditional structure that allows selection to be made based on value of a variable.

Purpose:

Used to test for a large number of discrete values.

24. Iterative/Repetition statements

Definition:

One or more statements are repeated till a test condition is met or whilst a test condition is TRUE.

Purpose:

To repeat same or similar code a number of times, which allows for shorter code.

Types:

- Count-controlled loops
- Pre-condition loops
- Post-condition loops

25. FOR .. TO .. NEXT (Count-controlled loop)

Definition:

A loop structure that iterates a set number of times.

Purpose:

used when a **set** (known) number of iterations are required

26. WHILE .. DO .. ENDWHILE (Pre-condition Loops)

Definition:

- A loop that iterates whilst a specified condition exists
- Criteria is pre-tested (It has criteria check at the start)
- It may never run

27. REPEAT .. UNTIL (Post-condition Loops)

Definition:

- A loop that iterates until a condition is met
- Criteria is post-tested (It has criteria check at the end)
- It iterates at least once

Differences between REPEAT ... UNTIL and WHILE ... DO ... ENDWHILE:

- In WHILE the criteria is pre-tested but in REPEAT..UNTIL the criteria is post-tested
- WHILE may never run while REPEAT.. UNTIL runs at least once
- WHILE iterates whilst a condition is TRUE but REPEAT UNTIL iterates till a condition is TRUE

28. Standard methods of solution

- Totalling
- Counting
- Finding **maximum**/**minimum** value
- Finding **Average** (mean) value
- Searching using **Linear Search**
- Sorting using a **bubble sort**

29. Totaling

Definition:

The process of summing a list of number.

Example:

```
Sum ← Sum + Num
```

30. Counting

Definition:

The process of finding how many items are in a list.

Example:

```
Count ← Count + 1
```

Arrays

31. Array

Definition:

A data structure that holds a number of elements of the same data type

Purpose:

- To store multiple values under the same identifier making the code shorter
- Allows for simpler programming

Index definition

The position of an element in an array

Dimension definition

The number of indexes required to access an element in an array

32. One-dimensional (1-D) Array

Definition:

- a list / one column of items
- … of the same data type
- … stored under a **single** name
- … with a **single** index to identify each element

Purpose of using arrays:

- Makes the program easier to design / code / test / understand
- Code will be more compact
- Code will be more organized
- Less likely to make errors

Declaring a 1D array:

```
DECLARE Numbers : ARRAY[1:10] OF INTEGER
```

OR

```
DECLARE Numbers[1:10] : INTEGER
```

33. Two-dimensional (2-D) Array

Definition:

- a table-like collection of items
- … of the same data type
- ⋯ has row and columns
- … stored under a **single** name
- … with **two** indexes to identify each element

Declaring a 2D array:

```
DECLARE Marks: ARRAY[1:10, 1:3] OF INTEGER
```

OR

```
DECLARE Marks[1:10, 1:3] : INTEGER
```

.

Linear Search & Bubble Sort

34. Linear Search

Definition:

The process of finding an value within a list of elements

Steps:

- Loop over a list of items starting with the first element and finishing with the last
- Compares each element with a search value⋯
- ⋯ if it is equal to the search value, set a flag to TRUE
- ⋯ it is not move to the next element
- After iterating over all the elements the flag shows whether the element is found or not

35. Bubble Sort

Definition:

The process of sorting a list of items either in alphabetical order or using numeric values

Steps:

- Loops over a list of items starting with the first and finishing with next-to-last element
- Each element is compared with the next element
- ⋯ swaps the elements if the elements are in the wrong order
- ⋯ after reaching the last element the last element is now in the correct position
- This is repeated a number of times equivalent to the number of elements in the list⋯
- ⋯to ensure that all elements are in the correct position

File Handling

36. Purpose of using files:

- Data is not lost when the computer is switched off (data is stored permanently)
- Data can be used by more than one program or reused when a program is run again
- Data can be transported from one computer to another

Functions / Procedures

37. Subroutine

Definition:

- Sub-program
- Used to perform a frequently used operation within a program
- Given a name and can be called when needed
- Can be reused by another program
- Written in High Level Language

Purpose of using subroutines (functions / procedures):

- to enable the programmer to write a collection of programming statements under a single identifier
- to allow procedures to be re-used within the program or in other programs
- to enable different programmers to work on different procedures in the same project
- to make programs easier to maintain due to program code being shorter

38. Function

Definition:

A subroutine that always returns a value.

39. Procedure

Definition:

A subroutine that doesn't return a value.

40. Function/Procedure Definition

Definition:

- Setting-up the function/procedure
- Includes defining the:
- Function/Procedure name
    - Parameter names and data types
    - In case of a function, the return data type
- Done only once

41. Function/Procedure Call

Definition:

- Using/executing the function code with parameters values
- Can be done multiple times

Difference between function/procedure definition and function/procedure call

- Defining is done once, calling can be done multiple times
- Defining is setting up the function/procedure and calling is using the function

Describe what happens when a function is called during the execution of a program

- A call statement is used in order to make use of a function
- Parameters are be passed from the main program to the function
- The function performs its task..
- ..and returns a value to the main program

42. Local Variable

Definition:

- A variable that is defined inside the scope of a subroutine (function/procedure)
- Its scope is restricted to only the part of the program where it was defined
- Can be accessed only within the scope of the subroutine (function/procedure) where it was defined

43. Global Variable

Definition:

- A variable that is defined outside the scope of a subroutine (function/procedure)
- Its scope covers the whole program
- Can be accessed from outside any function and from inside the scope of any function

Differences between local and global variables:

- for local variables the scope is a defined block of code (procedure/function)…
- …while global variables the scope is the whole program
- For local variables value **cannot** be changed elsewhere in the program…
- … while global variables the value can be changed anywhere in the program

44. Parameter

Definition:

A variable that is used to pass values (from the main program) to a subroutine (function/procedure)

Purpose:

- to pass values from the main program to a procedure / function
- …so that they can be used in the procedure / function
- allow the procedure / function to be **re-used** with different data

45. Return value

Definition:

A value that is passed from inside a function to the main program

46. Library Routines

Definition:

It is a list of programming instructions that is:

- Given a name
- Already available for use
- Pre-tested

Benefits of using library routines:

- Library routines make writing programs easier and faster as the code is already written
- They make testing the program easier as the code has already been tested and debugged
- Immediately available for use // They are readily available / speed up development time
- Can perform a function that you may not be able to program yourself

Purpose of `ROUND`:

- To return a value rounded to a specified number of digits / decimal places
- The result will either be rounded to the next highest or the next lowest value
- … depending on whether the value of the preceding digit is >=5 or <5
- Example of `ROUND` for example, `ROUND(4.56, 1) = 4.6`

Purpose of `RANDOM`:

- To generate (pseudo) random numbers
- …(usually) within a specified range
- Allow example e.g. `RANDOM() * 10` returns a random number between `0` and `10`.

Purpose of `DIV`:

- To perform integer division of two numbers..
- .. with the fractional part discarded
- Example of `DIV` For example `DIV(9,4) = 2`

Purpose of `MOD`:

- To perform (integer) division when one number is divided by another
- .. and find the remainder
- Example: `7 MOD 2 = 1`

Databases

47. Database

Definition:

A structured collection of related data that allows people to extract information in a way that meets their needs.

48. Purpose of SQL keywords

Purpose of `SELECT`:

An SQL command that identifies the fields to be displayed

Purpose of `FROM`:

An SQL command that identifies the table to use

Purpose of `WHERE`:

An SQL command that identifies the search criteria // An SQL command to include only the records that match a given condition.

Purpose of `ORDERBY`:

An SQL command that sorts the result of the query by a given column

Purpose of `SUM` keyword in SQL:

An SQL command that returns the sum of all values in a field

Purpose of `COUNT` keyword in SQL:

An SQL command that counts the number of records returned by the query

49. Database terms:

Term	Definition	Examples
Table	A collection of related records	Table of students
Field	A column that contains one specific piece of information and has one data type	For a student the fields could include: • Student name • Student ID • Age • Address
Field Name	Title given to each field	
Record	A row within a table that contains data about a single item, person or event	Details (all information) of one student
Primary Key	A field that contains unique data and can't have duplicates	Student ID field

Validation & Verification

50. Validation

Definition:

The automated checking by a program that data to be entered is sensible.

51. Examples of validation checks:

Validation check	Definition	Examples
Type Check	Checks that the data entered has the appropriate data type	A person's age should be a number not text.
Presence Check	Checks that some data has been entered and the value has not been left blank	An email address must be given for an online transaction
Range Check	It checks that only numbers within a specified range are accepted	Exam marks between 0 and 100 inclusive
Length Check	It checks that data contains an exact number of characters	A password with exactly 8 characters in length
Format Check	It checks that the characters entered conform to a pre-defined pattern	A company with IDs that start with MS then three numbers would have a format of MS###
Check Digit	A digit that it is calculated from all the other digits and appended to the number	Used in barcodes

52. Verification

Definition:

Checking that the data has not changed during input to a computer or during transfer between computers.

Purpose:

To ensure the accuracy of transcription.

Types:

- Screen/Visual Check
- Double Data Entry

53. Screen/Visual Check

Definition:

- Data is compared visually with the source document by a user
- The user is asked to confirm that the data entered is same as original
- If user finds a difference, the data is re-entered

54. Double Data Entry

Definition:

- Data is entered twice (sometimes by two different people)
- A computer checks that both entries are equal
- If they are not equal, an error message requesting to re-enter the data is displayed

Testing data

Definition:

Data used to check that a computer responds correctly to correct and incorrect values.

Purpose:

- check that the program works as expected
- check for errors
- check that the program **rejects any invalid data** that is input
- check that the program **only accepts** reasonable data

55. Types of test data:

Type	Definition	Examples
Normal Data	Correct data that should be accepted	The month can be any whole number in the range of 1 to 12
Abnormal Data	Data outside the limits of acceptability, or wrong type of data, and should be rejected	All the following values are not allowed as inputs for the month: • Negative numbers • Any value greater than 12 • Letters or non-numeric data • Non-integer values (e.g., 3.5,10.75, etc.)
Extreme Data	Largest/smallest acceptable value and should be accepted	The extreme values of month can be either 1 or 12
Boundary Data	The largest/smallest acceptable value and the corresponding smallest/largest rejected value	The extreme values of month can be either 1 or 12 The (boundary) rejected values of month can be 0 and 13

Digital Logic

56. Logic gate

Definition:

A small physical device that controls the flow of electrical signals in a pre-determinant way.

Purpose:

- To carry out a logical operation
- To control the flow of electricity through a logic circuit
- To alter the output from given inputs

Why data is stored as binary in computers?

- Computer use logic gates
- Logic gates use one of two values: 0 or 1

Types of logic gates:

- NOT
- AND
- OR
- XOR
- NAND
- NOR

Purpose (Explanation) of NOT gate:

- It has one input
- Output will be 1 if the input is 0
- Output will be 0 if the input is 1

Purpose (Explanation) of AND gate:

- It has two inputs
- Output will be 1 if both inputs are 1
- Output will be 0 if either input is 0

Purpose (Explanation) of OR gate:

- It has two inputs
- Output will be 1 if either input is 1
- Output will be 0 if both inputs are 0

Purpose (Explanation) of XOR gate:

- It has two inputs
- Output will be 1 if both inputs are different
- Output will be 0 if both inputs are 0
- Output will be 0 if both inputs are 1

Purpose (Explanation) of NAND gate:

- It has two inputs
- Output will be 1 if either input is 0
- Output will be 0 if both inputs are 1

Purpose (Explanation) of NOR gate:

- It has two inputs
- Output will be 1 if both inputs are 0
- Output will be 0 if either input is 1

57. Logic circuit

Definition:

A combination of logic gates to carry out a particular function

58. Truth Table

Definition and purpose:

A table is used to trace the output from a logic gate or a logic circuit considering all possible combinations of 0s and 1s that can be input.