8. Programming

Data types

A classification of data into groups according to the kind of data they represent.

- Allows data to be stored in an appropriate way (as numbers/characters)
- Allows data to be manipulated effectively (numbers with arithmetic operators/ characters with concatenation)
- Allows automatic validation (in some cases)

1. **Integer:** used to represent whole numbers, either positive or negative.
   Examples: 10, -5, 0
2. **Real:** used to represent numbers with a fractional part, either positive or negative.
   Examples: 3.14, -2.5, 0.0
3. **Char:** used to represent a single character such as a letter, digit or symbol
   Examples: 'a', 'B', '5', '$'
4. **String:** to represent a sequence of characters; may have no characters (empty string)
   Examples: "Hello World", "1234", "@#$%, “ “
5. **Boolean**: used to represent true or false values.
   Examples: True, False

Variables & Constants

- Used to store a single item of data in a program.
- This can be accessed through an identifier.
- Variables can be changed during program execution.
- Constants remain the same/ do not change during program execution.

Explain how variables and constants should be used when creating and running a program.

- variables and constants should have meaningful identifiers
- …so that programmers/future programmers are able to understand their purpose
- they are both used for data storage
- constants store values that never change during the execution of a program // by example
- variables contain values that have been calculated within the program / can change during the execution of the program // by example

Example - Declaring variables

When there’s no value	When there is a value

`DECLARE name : STRING`
`DECLARE age : INTEGER`
	`name ← “Naomi”`
	`age ← 16`

Example - Declaring constants

```
const PI ← 3.14
```

*Name of the constant to be written in caps

Input & Output

Input: the process of providing data or information to a program.

- User Input: data/information entered by user during execution; requires a prompt.

Output: the process of displaying or saving the results of a program to the user.

Example

```
OUTPUT "Enter your name"
INPUT name
OUTPUT "Hello ", name
```

Sequence

Executing instructions in a particular order/ step-by-step.

Example

```
PRINT “Hello world”
x ← 5
y ← 10
z ← x + y
PRINT z
```

Selection

Allows execution of different sets of instructions based on certain conditions.

IF statement: Used to execute a set of instructions if a condition is true.

```
IF <condition>
  THEN
    <instructions>
ENDIF
```

IF ELSE statement: Used to execute one set of instructions if a condition is true and a different set of instructions if the condition is false.

```
IF <condition>
  THEN
    <instructions>
  ELSE
    <instructions>
ENDIF
```

Examples

```
IF Found   //Boolean variable
   THEN
      OUTPUT "Search was successful"
   ELSE
      OUTPUT "Search was unsuccessful"
ENDIF
```

```
IF ((Height > 1) OR (Weight > 20)) AND (Age < 70) AND (Age > 5)
   THEN
      OUTPUT "you can ride"
   ELSE
      OUTPUT "too small, too young or too old to ride"
ENDIF
```

IF ELSE IF statement (nested IF): used to test multiple conditions and execute different statements for each condition.

```
IF <condition>
  THEN
     <instructions>
  ELSE IF <condition>
     THEN
        <instructions>
  ELSE
     <instruction>
ENDIF
```

Examples

```
x ← 5
IF x > 0
   THEN
      PRINT "x is positive"
   ELSE IF x < 0
      THEN
         PRINT "x is negative"
   ELSE
      PRINT "x is zero"
ENDIF
```

```
OUTPUT "Please enter mark"
INPUT PercentageMark
IF PercentageMark < 0 OR PercentageMark > 100
     THEN
```

```
          OUTPUT “invalid mark”
     ELSE
          IF PercentageMark > 49
               THEN
                  OUTPUT “pass”
               ELSE
                  OUTPUT “Fail”
          ENDIF
ENDIF
```

CASE statement: To execute different sets of instructions based on the value of a variable.

```
CASE OF <variable>
  <value1> : <instructions>
  <value2> : <instructions>
  …
  OTHERWISE <instructions>
ENDCASE
```

Examples

```
CASE OF Grade
   “A” : OUTPUT “Excellent”
   “B” : OUTPUT “Good”
   “C” : OUTPUT “Average”
   OTHERWISE OUTPUT “Improvement needed”
ENDCASE
```

```
CASE OF OpValue
   “+” : Answer ← Num1 + Num2
   “-” : Answer ← Num1 - Num2
   “/” : Answer ← Num1 / Num2
   “*” : Answer ← Num1 * Num2
   OTHERWISE OUTPUT “please enter valid choice”
ENDCASE
```

Iteration

The process of repeating a set of instructions for a specific number of times or until a specific condition is met.

Count-controlled loops

- Definite loop
- Used when number of iterations is known beforehand
- Counter variable is used, which is incremented/decremented after each iteration
- Loop continues until the counter reaches a specific value

Example

```
FOR Count ← 1 TO 10
      OUTPUT "Enter student name "
      INPUT StudentName[Count]
NEXT Count
```

Precondition loops

- Indefinite loop
- Used when number of iterations is not known beforehand and is dependent on a condition being true
- Condition is checked at the beginning; the loop will continue to execute while the condition is true and stops once the condition becomes false
- May not have iterations

Examples

```
Total ← 0
OUTPUT "Enter value for mark, -1 to finish "
INPUT Mark
WHILE Mark <> -1 DO
      Total ← Total + Mark
      OUTPUT "Enter value for mark, -1 to finish
      INPUT Mark
ENDWHILE
```

```
INPUT temperature
WHILE temperature > 37 DO
      OUTPUT "Patient has a fever"
      INPUT temperature
END WHILE
```

Postcondition loops

- Used when number of iterations is not known beforehand and when the loop must execute at least once, even if the condition is false from the start
- The condition is checked at the end of the loop
- Has at least 1 iteration

Examples

```
Total ← 0
Mark ← 0
REPEAT
      Total ← Total + Mark
      OUTPUT "Enter value for mark, -1 to finish"
```

```
    INPUT Mark
UNTIL Mark = -1
```

```
REPEAT
    INPUT guess
UNTIL guess = 42
```

Question: state the type of pseudocode statement

`IF Number = 20:` selection

`PRINT Number:` output

`Number ← Number + 1:` counting

`X ← X + Y:` totalling

String Handling

- String: sequence of characters, such as letters, numbers, and symbols
- Used to represent/store text in a computer program
- Characters in string can be labelled by position number; first character can either be in position 0 or 1
- String handling: operations that can be performed on strings

```
MyString ← "save my exams"
```

Length

Length of string = number of characters it contains.

```
LENGTH("Computer Science")      //output: 16
LENGTH(MyString)                //output: 13
```

Substring

A portion of a string.

`SUBSTRING(<string>, a, b)`

- a = where the substring should start from
- b = number of characters to be included in the substring

```
SUBSTRING("Computer Science", 10, 7)  //output: Science
SUBSTRING(MyString, 9, 5)             //output: exams
```

Upper

The `UCASE()` method converts all characters of a string to uppercase.

```
UCASE("Computer Science")       //output: COMPUTER SCIENCE
```

Lower

The `LCASE()` method converts all characters of a string to lowercase.

```
LCASE("Computer Science")        //output: computer science
```

Arithmetic, Comparison & Logical/ Boolean Operators

Arithmetic operators

addition	+
subtraction	-
division	/
multiplication	*
exponentiation	^
modulo (returns remainder of division operation)	MOD
integer division (returns whole number of division operation)	DIV

Comparison operators

Greater than	>
Less than	<
Equal	=
Greater than or equal	>=
Less than or equal	<=
Not equal	<>

Boolean/ logical operators

Returns True if both conditions are True	AND
Returns True if one or both conditions are True	OR
Returns the opposite of the condition (True if False, False if True)	NOT

Subroutines - Procedures & Functions

Subroutines

- Sequence of instructions that perform a specific task or set of tasks.
- Used to simplify a program by breaking it into smaller, more manageable parts:

- They can avoid duplicating code & can be reused throughout a program.
- They can improve readability and maintainability of code.
- They can perform calculations, retrieve data, or make decisions based on input.

Parameters

Values that are passed into a subroutine; Subroutines can have multiple parameters.

- They can be used to customise the behaviour of a subroutine
- They have default values, which are used if no value is passed in
- They can be of different types
- They can be passed by value or by reference:
    - When a parameter is passed by value, a copy of the parameter is made and used within the subroutine.
    - When it's passed by reference, the original parameter is used within the subroutine, and any changes made to the parameter will be reflected outside the subroutine.

Defining & calling a subroutine

- A subroutine only needs to be defined once to set it up.
- After setting up the subroutine, it can be called as many times as required.

Procedure

- A type of subroutine that doesn't return a value.
- A set of programming statements grouped together under a single name.
- Defined using the `PROCEDURE` keyword in pseudocode, and can be called from other parts of the program using its name.

Example - procedure without parameters

<u>Procedure definition</u>

```
PROCEDURE Stars
      OUTPUT "******"
ENDPROCEDURE
```

<u>Procedure call</u>

```
CALL Stars
```

Examples - procedure with parameters

<u>Procedure definition</u>

```
PROCEDURE Stars(Number : INTEGER)
      DECLARE Counter: INTEGER
      FOR Counter ← 1 TO Number
            OUTPUT "*"
      NEXT Counter
ENDPROCEDURE
```

Procedure call
```
CALL Stars (7)
```

Procedure definition
```
PROCEDURE Area(Length : INTEGER, Width : INTEGER)
    Area ← Length * Width
    OUTPUT "The area is ", Area
ENDPROCEDURE
```

Procedure call
```
CALL Area (5,3)
```

Explain why a programmer would use procedures and parameters when writing a program

Procedures

- to enable the programmer to write a collection of programming statements under a single identifier
- to allow modular programs to be created // to allow procedures to be re-used within the program or in other programs
- to make program creation faster because procedures can be re-used // to enable different programmers to work on different procedures in the same project
- to make programs shorter (than using the repeated code) / using less duplication of code // to make programs easier to maintain due to being shorter.

Parameters

- 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.

Function

- A type of subroutine that returns a value.
- A set of programming statements grouped together under a single name.
- Defined using the `FUNCTION` keyword in pseudocode, and can be called from other parts of the program using its name.
- Returns a value using the `RETURN` keyword in pseudocode, and the return value can be stored in a variable.

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 // the function is called using its identifier

- Parameters are / may be passed (from the main program) to the function (to be used within the function)
- The function performs its task ...
- ... and returns a value / values to the main program

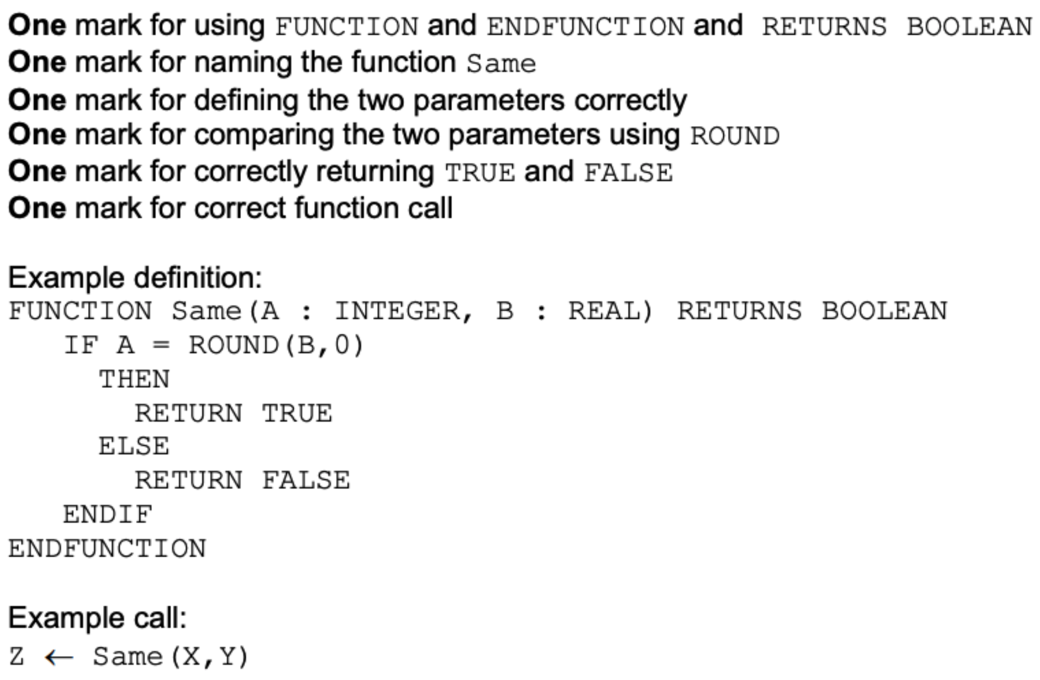
State the difference between defining and calling a function.

- Defining the function = setting up the function; Calling the function = using the function
- Function is defined only once; Function can be called many times

The function Same(A,B) returns TRUE if the value of A is the same as the value of B when B is rounded to the nearest whole number and FALSE otherwise. Write pseudocode statements to:

• define the function

• call the function with X and Y and store the return value in Z



One mark for using `FUNCTION` and `ENDFUNCTION` and `RETURNS BOOLEAN`
One mark for naming the function `Same`
One mark for defining the two parameters correctly
One mark for comparing the two parameters using `ROUND`
One mark for correctly returning `TRUE` and `FALSE`
One mark for correct function call

Example definition:

```
FUNCTION Same(A : INTEGER, B : REAL) RETURNS BOOLEAN
   IF A = ROUND(B,0)
     THEN
       RETURN TRUE
     ELSE
       RETURN FALSE
   ENDIF
ENDFUNCTION
```

Example call:

```
Z ← Same(X,Y)
```

Examples

Function definition

```
FUNCTION Celsius(Temperature : REAL) RETURNS REAL
     RETURN (Temperature - 32) / 1.8
ENDFUNCTION
```

Function call

```
MyTemp ← Celsius(MyTemp)
OUTPUT(MyTemp)
```

OR

```
OUTPUT(Celsius(98))
```

NOTE: Function calls are not standalone, and are usually made on the RHS of an expression
NOTE: Procedures and functions may have up to two parameters (in IGCSE)

Local & Global Variables

Local variable

- Variable that is declared inside a subroutine
- Can only be accessed from within that subroutine; its scope is restricted to subroutine

Global variable

- Variable that is declared outside of a subroutine
- Can be accessed from anywhere in the program; its scope covers the whole program
- Used to pass data between different subroutines
- Overuse of global variables make programs difficult to understand & debug

Describe two differences between local variables and global variables.

- local variables - scope is a defined block of code/subroutine/procedure/function
- ... global variables – scope is the whole program
- local variables - value cannot be changed elsewhere in the program
- ... global variables – value can be changed anywhere in the program

Example

```
DECLARE Num1, Num2, Ans : INTEGER     //global variables
PROCEDURE Test
     DECLARE Num3, Ans : INTEGER          //local variables
     Num1 ← 10
     Num2 ← 20
     Num3 ← 30
     Ans ← Num1 + Num2
     OUTPUT "Num1 is now ", Num1
     OUTPUT "Num2 is now ", Num2
     OUTPUT "Answer is now ", Ans
ENDPROCEDURE

Num1 ← 50
Num2 ← 100
Ans ← Num1 + Num2

OUTPUT "Num1 is ", Num1
OUTPUT "Num2 is ", Num2
OUTPUT "Answer is ", Ans

CALL Test
```

```
OUTPUT "Num1 is still ", Num1
OUTPUT "Num2 is still ", Num2
OUTPUT "Answer is still ", Ans
OUTPUT "Num3 is ", Num3
```

Output

```
Num1 is 50
Num2 is 100
Answer is 150

Num1 is now 10
Num2 is now 20
Answer is now 30

Num1 is still 50
Num2 is still 100
Answer is still  150
ERROR - Num3 is undefined
```

Library Routines

Pre-written code that can be used in programs to perform specific tasks.

- Using library routines saves time: no need to write code from scratch
- Library routines are fully tested and proven to work: errors are less likely
- Libraries can be included in programs by importing them
- Each language has specific syntax for importing libraries
- It is important to check the documentation for a library to understand its usage and available functions

Common library routines

name	**function**	**syntax**
`MOD`	A function that returns the remainder of division operation.	`x MOD y`
`DIV`	A function that returns the quotient of a division operation as a whole number.	`x DIV y`
`ROUND`	A function that rounds a number to a specified number of decimal places	`ROUND(x,n)`
`RANDOM`	A function that generates a random number between x & n	`RANDOM(x,n)`

Examples

```
Value1 ← MOD(10,3)                //Value1 = 1
```

```
Value2 ← DIV(10,3)              //Value2 = 3
Value3 ← ROUND(6.97354, 2)      //Value3 = 6.97
Value4 ← RANDOM()               //Value4 = number between 0 & incl
```

Explain the purpose of the library routine MOD
- Performs (integer) division, where one number is divided by another
- Finds/ returns the remainder
- Eg. 7 MOD 2 = 1

Explain the purpose of the library routine RANDOM
- To generate random numbers
- Within a specified range
- Eg. RANDOM() * 10 returns a random number between 0 and 10

Explain the purpose of the library routine DIV
- To perform integer division
- Meaning only the whole number part of the answer is retained
- Example of DIV For example DIV(9,4) = 2

Explain the purpose of the library routine 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.

Arrays

- A data structure containing several elements of the same data type, in a uniformly accessible manner.
- They are used to store & manage multiple values of the same data type efficiently.
- These elements can be accessed using the same identifier name.
- Position of each element in an array is defined using the array's index.
- Variables are used as indexes in arrays.
- First element of an array can have an index of 0 or 1.

Use of variables in arrays
- They are used as indexes in arrays
- They are used to access & modify elements by repeatedly checking every element
- This is useful when iterating through the array using loops/ performing calculations

Explain how indexing could be used to search for a value stored in a one-dimensional array

- using a counter to index the array
- so that the same code can be repeatedly used to check every element // every element can be checked in a loop

One-dimensional array

List; collection of items of the same data type

Declaring a 1D array

```
DECLARE <ArrayName> : ARRAY[x:y] OF <DataType>
```

x = first index value

y = last index value

```
DECLARE MyList : ARRAY[0:9] OF INTEGER
```

Populating a 1D array

```
MyList[4] ← 27
```

Using a loop:

```
OUTPUT "Enter these 10 values in order: 27, 19, 36, 42, 16, 89, 21,
16, 55, 72"
FOR Counter ← 0 TO 9
      OUTPUT "Enter next value "
      INPUT MyList[Counter]
NEXT Counter

OUTPUT MyList[1]           //output: 19
```

NOTE: Arrays can also be populated as they are declared.

Two-dimensional array

Table/grid with rows and columns; an array of arrays.

Declaring a 2D array

```
DECLARE <ArrayName> : ARRAY[x:y, m:n] OF <DataType>
```

x = first index value for rows

y = last index value for rows

m = first index value for columns

n = last index value for columns

Example

27	31	17
19	67	48
36	98	29
42	22	95
16	35	61
89	46	47
21	71	28
16	23	13
55	11	77
34	76	21

```
DECLARE MyList : ARRAY[0:9, 0:2] OF INTEGER
```

NOTE: First element located at 0,0

Populating a 2D array

Using a loop:

```
OUTPUT “Enter these 10 values in order: 27, 19, 36, 42, 16, 89, 21,
16, 55, 34”
OUTPUT “Enter these 10 values in order: 31, 67, 98, 22, 35, 46, 71,
23, 11, 76
OUTPUT “Enter these 10 values in order: 17, 48, 29, 95, 61, 47, 28,
13, 77, 21
FOR ColumnCounter ← 0 TO 2
    FOR RowCounter ← 0 TO 9
        OUTPUT “Enter next value ”
        INPUT MyTable[RowCounter, ColumnCounter]
    NEXT RowCounter
NEXT ColumnCounter

OUTPUT MyList[2,1]    //Output: 98
```

File Handling

Purpose of storing data in a file to be used by a program

- data is stored permanently // data is not lost when the computer is switched off
- data can be moved to another computer // can be transported from one system to another

    - another copy of data can be made and stored/ accessed elsewhere // backup copy; data can be backed up or archived, ensuring data isnt lost when program/ computer shuts down
    - data can be used by more than one program or reused when a program is run again

    - They allow for large amounts of data to be stored and managed efficiently.
    - Files allow for organised storage of data, making it easier to find, access, and update.
    - They enable sharing of data between different programs & users.

File operations
- Opening: creates a connection between the file and the program.
- Reading: retrieves data from a file.
- Writing: saves new data to a file, overwriting existing content.
- Appending: adds new data to the end of a file without overwriting existing content.
- Closing: ends the connection between the file and program, releasing system resources.

NOTE:
- Single items of data can be numbers, characters, or other simple data types.
- A line of text typically ends with a newline character or an end-of-line marker.

```
DECLARE TextLine : STRING
DECLARE MyFile : STRING
MyFile ← "MyText.txt"

//writing line of text into file
OPEN MyFile FOR WRITE
     OUTPUT "Enter a line of text "
     INPUT TextLine
     WRITEFILE, TextLine
CLOSEFILE(MyFile)

//reading line of text from file
OUTPUT "The file contains this line of text: "
OPEN MyFile FOR READ
     READFILE, TextLine
     OUTPUT TextLine
CLOSEFILE(MyFile)
```

Write the pseudocode statements to:
• allow a string to be input to the variable Saying
• store the content of the variable Saying in a text file named "Quotations.txt"

• **make sure the text file is closed at the end of the algorithm.**

```
INPUT Saying
OPENFILE "Quotations.txt" FOR WRITE
WRITEFILE "Quotations.txt", Saying
CLOSEFILE "Quotations.txt"
```

A program halted unexpectedly with the error message 'File not found' whilst trying to read data from a file. Outline the actions that the program needs to take to prevent this error occurring

- before trying to open the file
- check the file exists
- if the file does not exist, then output a suitable error message.

Maintaining Programs

Why it is important to create a maintainable program

- Improves program quality (easier to understand & modify, so fewer bugs)
- Reduces development time and costs (requires less time and effort to modify)
- Enables collaboration: (multiple developers can work together on the same project)
- Increases program lifespan: (more likely to be updated and maintained over time)
- Adapt to changing requirements

How to create a well maintained program

- Use of meaningful identifiers
- Use of comments
- Use of procedures/functions
- Use of white space

- Use meaningful identifiers for variables, constants, arrays, procedures and functions.
    - Use descriptive & meaningful identifiers so code is easier to understand & maintain.
    - Avoid using single letters / abbreviations that may not be clear to others.
- Use procedures & functions:
    - They are reusable blocks of code that perform specific tasks.
    - This allows to modularise code & make it easier to understand, debug & maintain.
    - They should have descriptive names that clearly indicate what they do
- Use the commenting feature provided by the programming language:
    - Comments add descriptions to code to help readers understand what it is doing.
    - Use comments to explain purpose of variables, constants, procedures, functions, etc.
    - Use comments to document any assumptions or limitations of the code.
- Relevant and appropriate commenting of syntax:
    - Commenting of syntax: used to explain the purpose of individual lines / blocks of code within the program

    - Use it to describe complex algorithms/ to provide additional information on the purpose or behaviour of code

State three different features of a high-level programming language that a programmer could use to make sure that their program will be easier to understand by another programmer. Give an example for each feature.

- Feature 1: ensuring that all identifiers have meaningful names
- Example: using `Total` to store a running total

- Feature 2: using comments to explain how the program works
- Example: `//all values are zeroed before the next calculation`

- Feature 3: using procedures and functions for the tasks within a program
- Example: `CalculateInterest(Deposit, Rate)`

State two features that should be included to create a maintainable program. Give a reason why each feature should be used.

- Meaningful identifiers: to enable the programmer (or future programmers) to easily recognize the purpose of a variable / array / constant // to enable easy tracking of a variable / constant / array through the program
- Use of comments: to annotate each section of a program so that a programmer can find specific sections / so that the programmer knows the purpose of that section of code
- Procedures and functions: to make programs modular and easier to update / add functionality

Programming examples in pseudocode

1. **Program to calculate & output total marks, average marks & number of marks entered**

```
DECLARE Total, Average : REAL
DECLARE Mark, Counter : INTEGER
Total ← 0
Mark ← 0
Counter ← -1  //counter starts at -1 so that extra mark of 999 not counted

OUTPUT "Enter marks, 999 to finish"
REPEAT
   INPUT Mark
   Total ← Total + Mark
   Counter ← Counter + 1
UNTIL Mark = 999
```

```
OUTPUT “Total mark is ”, Total
Average ← Total / Counter
OUTPUT “Average mark is ”, Average
OUTPUT “Number of marks is ”, Counter
```

2. **Program to output largest of 3 numbers input by user**

```
OUTPUT “Enter 3 numbers”
INPUT Num1, Num2, Num3
IF Num1 > Num2 AND Num1 > Num3
   THEN
     OUTPUT Num1, “ is largest”
   ELSE IF Num2 > Num1 AND Num2 > Num3
     THEN
        OUTPUT Num2, “ is largest”
   ELSE
     PRINT Num3, “ is largest”
ENDIF
```

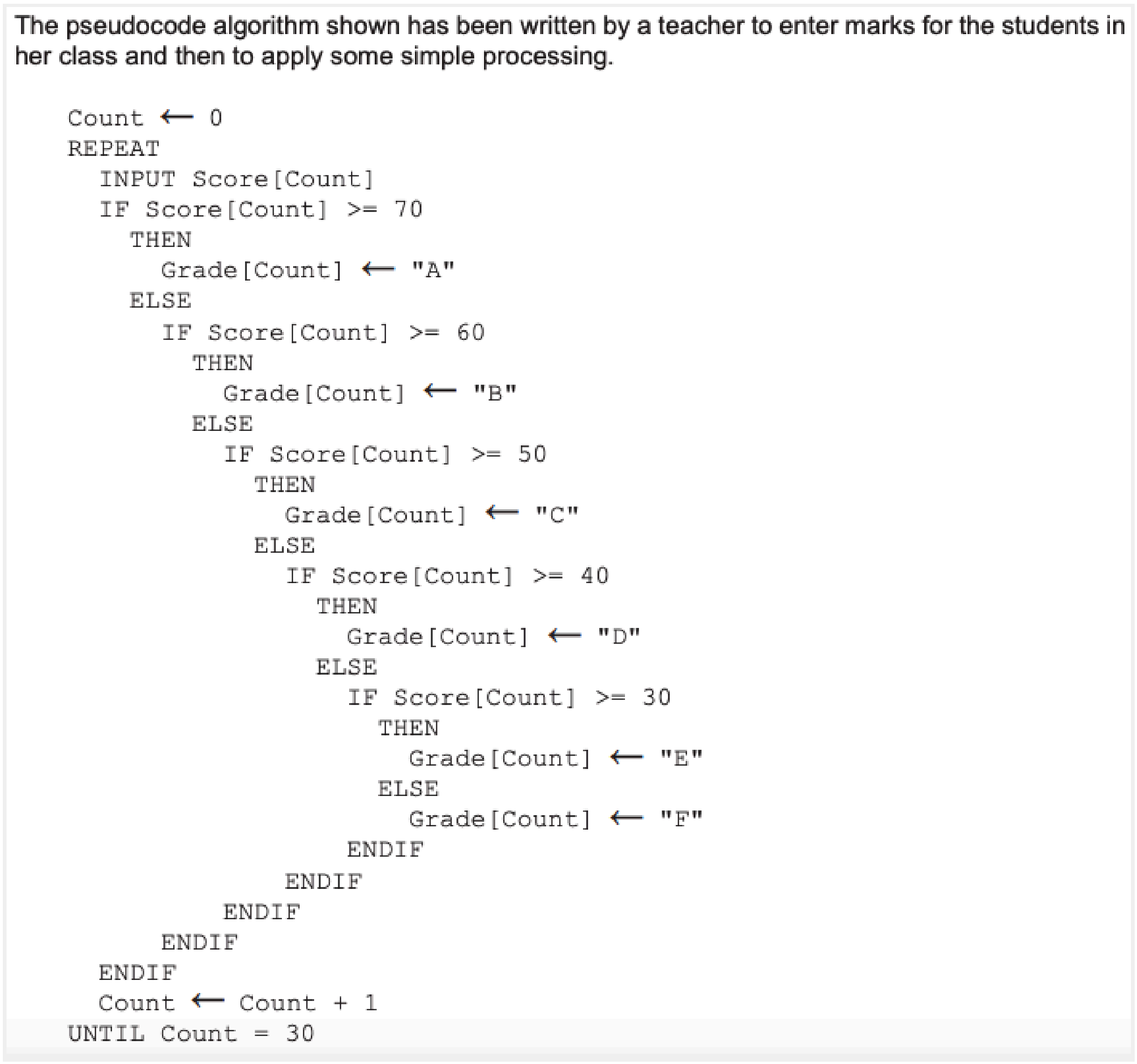
3. **Program to:**
   a. **Calculate & output highest, lowest and average marks for a class of 20 students**
   b. **Calculate & output largest, highest, lowest and average marks for each student**
   c. **Calculate & output largest, highest, lowest and average marks for each of the 6 subjects studied by the student - each subject has 5 tests**

   (Assume that all marks input are whole numbers between 0-100)

```
DECLARE ClassAverage, StudentAverage, SubjectAverage : REAL
DECLARE Student, Subject, Test : INTEGER
DECLARE ClassHigh, ClassLow, ClassTotal : INTEGER
DECLARE StudentHigh, StudentLow, StudentTotal : INTEGER
DECLARE SubjectHigh, SubjectLow, SubjectTotal : INTEGER

CONSTANT NumberOfTests = 5
CONSTANT NumberOfSubjects = 6
CONSTANT ClassSize = 20

ClassHigh ← 0
ClassLow ← 100
ClassTotal ← 0

FOR Student ← 1 TO ClassSize
   StudentHigh ← 0
   StudentLow ← 100
   StudentTotal ← 0

   FOR Subject ← 1 TO NumberOfSubjects
        SubjectHigh ← 0
        SubjectLow ← 100
        SubjectTotal ← 0

        FOR Test ← 1 TO NumberOfTests
             OUTPUT “Please enter mark ”
             INPUT Mark
             IF Mark > SubjectHigh
             THEN
                  SubjectHigh ← Mark
             ENDIF
             IF Mark < SubjectLow
             THEN
                  SubjectLow ← Mark
             ENDIF
             SubjectTotal ← SubjectTotal + Mark
        NEXT Test

        SubjectAverage ← SubjectTotal / NumberOfTests
        OUTPUT “Average for ”, Subject, “ is ”, SubjectAverage
        OUTPUT “Highest mark for ”, Subject, “ is ”, SubjectHigh
        OUTPUT “Lowest mark for ”, Subject, “ is ”, SubjectLow

        IF SubjectHigh > StudentHigh
        THEN
             StudentHigh ← SubjectHigh
        ENDIF
        IF SubjectLow < StudentLow
        THEN
             StudentLow ← SubjectLow
        ENDIF
        StudentTotal ← StudentTotal + SubjectTotal
   NEXT Subject

   StudentAverage ←
StudentTotal/(NumberOfTests*NumberOfSubjects)
   OUTPUT “Average mark for ”, Student, “ is ”, StudentAverage
   OUTPUT “Highest mark for ”, Student, “ is ”, StudentHigh
   OUTPUT “Lowest mark for ”, Student, “ is ”, StudentLow

   IF StudentHigh > ClassHigh
   THEN
        ClassHigh ← StudentHigh
   ENDIF
   IF StudenLow < ClassLow
   THEN
        ClassLow ← StudentLow
   ENDIF
   ClassTotal ← ClassTotal + StudentTotal
```

```
NEXT Student

ClassAverage ←
ClassTotal/(NumberOfTests*NumberOfSubjects*ClassSize)
OUTPUT "Average mark for clas is ", ClassAverage
OUTPUT "Highest mark for clas is ", ClassHigh
OUTPUT "Lowest mark for clas is ", ClassLow
```

4. **Describe what happens in this algorithm.**



The pseudocode algorithm shown has been written by a teacher to enter marks for the students in her class and then to apply some simple processing.

```
Count ← 0
REPEAT
  INPUT Score[Count]
  IF Score[Count] >= 70
    THEN
      Grade[Count] ← "A"
    ELSE
      IF Score[Count] >= 60
        THEN
          Grade[Count] ← "B"
        ELSE
          IF Score[Count] >= 50
            THEN
              Grade[Count] ← "C"
            ELSE
              IF Score[Count] >= 40
                THEN
                  Grade[Count] ← "D"
                ELSE
                  IF Score[Count] >= 30
                    THEN
                      Grade[Count] ← "E"
                    ELSE
                      Grade[Count] ← "F"
                  ENDIF
              ENDIF
          ENDIF
      ENDIF
  ENDIF
  Count ← Count + 1
UNTIL Count = 30
```

- Marks are input and stored in the array Score[]
- Marks are checked against a range of boundaries
- … and a matching grade is assigned to each mark that has been input
- … then stored in the array Grade[]…
- … at the same index as the mark input
- Algorithm finishes after 30 marks have been input // allows 30 scores to be entered

Write the pseudocode to output the contents of the arrays Score[] and Grade[] along with suitable messages.

```
FOR Count ← 0 TO 29
     PRINT "Student: ", Count, "Mark: ", Score[Count], "Grade: ",
Grade[Count]
NEXT Count
```

Describe how to change the algorithm to allow teachers to use it with any size of class.

- Add an input facility to allow teachers to enter the class size
- Add a variable to store the input class size
- Use the class size variable as the terminating condition for the loop
- Make sure arrays are sufficiently large to accommodate the largest possible class size

5. **Write an algorithm in pseudocode to input 500 positive whole numbers. Each number input must be less than 1000. Find and output the largest number input, the smallest number input and the range (difference between largest number and smallest).**

```
Large ← 0
Small ← 1000

FOR Count ← 1 TO 500
  REPEAT
    OUTPUT "Enter a whole number between 1 and 999"
    INPUT Number
  UNTIL Number >= 1 AND Number < 1000 AND Number = Number DIV 1

  IF Number < Small
    THEN
    Small ← Number
  ENDIF

  IF Number > Large
    THEN
    Large ← Number
  ENDIF
NEXT

Range ← Large - Small
OUTPUT "Largest number is ", Large, " Smallest number is ",
Small, "Range of numbers is", Range
```

Describe how the algorithm could be changed to make testing less time-consuming.

- Reduce the amount of numbers entered
- By decreasing the final value of the loop

OR

- Remove the need to input values

- By using random numbers / a previously populated array

6. **A one-dimensional array** `dataArray[1:20]` **needs each element set to zero.**
   a. **Write a pseudocode routine that sets each element to zero. Use the most suitable loop structure.**

```
FOR Count ← 1 TO 20
        dataArray[Count] ← 0
NEXT Count
```

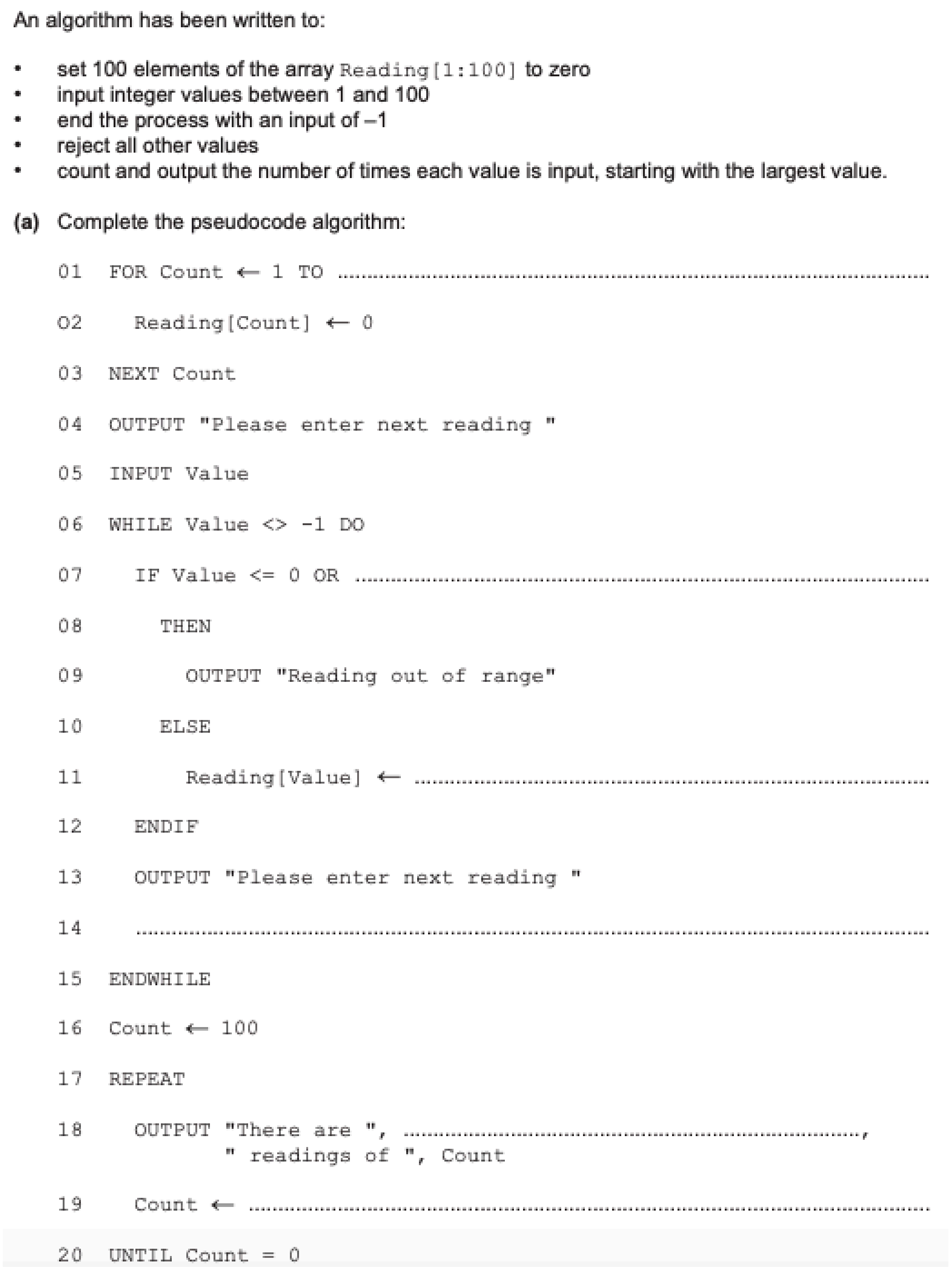
   b. **Explain why you chose this loop structure**
      - (A FOR loop has) a fixed number of repetitions //
      - No need to manage the loop counter //
      - No need to use another variable for the array index

7.

```
1   Count ← 0
2   REPEAT
3     FullScore ← 20
4     INPUT Number
5     FOR StoreLoop ← 1 TO Number
6       INPUT Score
7       FullScore ← FullScore
8     UNTIL StoreLoop = Number
9     OUTPUT "The full score is ", FullScore
10    OUTPUT "Another set of scores (Y or N)?"
11    OUTPUT Another
12    IF Another = "N"
13      THEN
14        Count ← 1
15    ENDIF
16  UNTIL Count = 1
```

Show how you could change the algorithm to store the individual scores in the array `ScoreArray[]`**, then find and print average score once scores have all been entered.**
- After line 6 // replace line 6:
- INPUT ScoreArray[StoreLoop]
- Between lines 8 and 10:
- AverageScore ← FullScore/Number
- OUTPUT "The average score is ", AverageScore

8.



An algorithm has been written to:

- set 100 elements of the array `Reading[1:100]` to zero
- input integer values between 1 and 100
- end the process with an input of –1
- reject all other values
- count and output the number of times each value is input, starting with the largest value.

(a) Complete the pseudocode algorithm:

```
01 FOR Count ← 1 TO ..............................................
02   Reading[Count] ← 0
03 NEXT Count
04 OUTPUT "Please enter next reading "
05 INPUT Value
06 WHILE Value <> -1 DO
07   IF Value <= 0 OR ............................................
08     THEN
09       OUTPUT "Reading out of range"
10     ELSE
11       Reading[Value] ← ........................................
12   ENDIF
13   OUTPUT "Please enter next reading "
14   .............................................................
15 ENDWHILE
16 Count ← 100
17 REPEAT
18   OUTPUT "There are ", ......................................,
            " readings of ", Count
19   Count ← ....................................................
20 UNTIL Count = 0
```

- Line 1 100
- Line 7 Value > 100 // Value >= 101
- Line 11 Reading[Value] + 1
- Line 14 INPUT Value
- Line 18 Reading[Count]
- Line 19 Count – 1

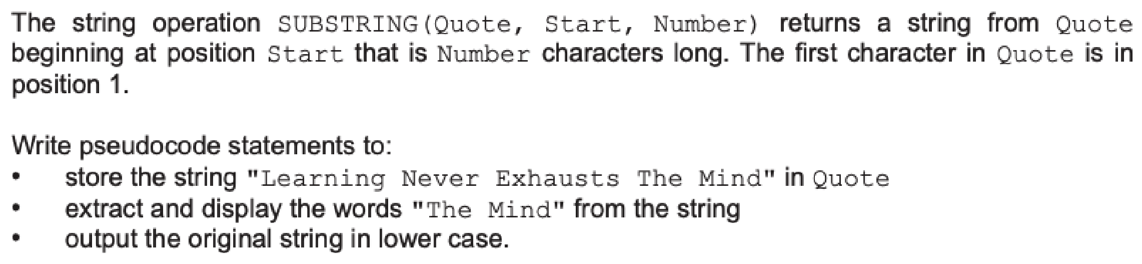
Describe how the algorithm could be changed so that it does not output any counts of zero.
• use an IF/conditional statement
• to check if Reading[Count] not equal to zero
• before outputting the value // between statements 17 and 18 // code sample showing position

```
IF Reading[Count] <> 0
    THEN
         OUTPUT
ENDIF
```

9. **Write a pseudocode algorithm to:**
• convert P to upper case
• find the position of Q in the string P (the first character in this string is in position 1)
• store the position of Q in the variable Position

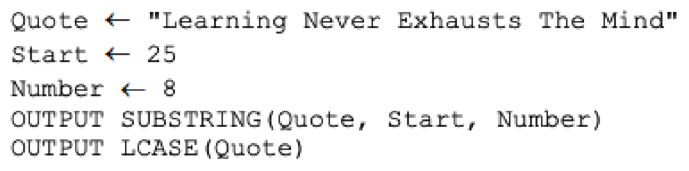
```
P ← UCASE(P)
Counter ← 1
Position ← 0
REPEAT
   IF SUBSTRING(P, Counter, 1) = Q
   THEN
        Position ← Counter
   ENDIF
Counter ← Counter + 1
UNTIL Position <> 0 OR Counter = LENGTH(P)
```

10.



The string operation `SUBSTRING(Quote, Start, Number)` returns a string from `Quote` beginning at position `Start` that is `Number` characters long. The first character in `Quote` is in position 1.

Write pseudocode statements to:
• store the string `"Learning Never Exhausts The Mind"` in `Quote`
• extract and display the words `"The Mind"` from the string
• output the original string in lower case.



```
Quote ← "Learning Never Exhausts The Mind"
Start ← 25
Number ← 8
OUTPUT SUBSTRING(Quote, Start, Number)
OUTPUT LCASE(Quote)
```

11. **A programmer is writing a data entry program for booking theatre seats. The programmer needs the program to accept only whole numbers that are greater than**

or equal to one and less than or equal to six. Complete this pseudocode to perform the 2 validation checks:

```
OUTPUT "Please enter the number of seats you want to book "

INPUT Seats
```

One mark for each point (max five)

- use of loop for check
- checking for whole number
- checking for number greater than or equal to one
- … and less than or equal to six
- Appropriate error/reinput message
- ability to reinput value

Example:

```
WHILE Seats < 1 OR Seats > 6 OR Seats <> ROUND(Seats, 0) DO
    OUTPUT "Please enter a valid number of seats "
    INPUT Seats
ENDWHILE
```

Give one item of test data to use when testing this program. State the reason for your choice of test data.

- Test data: 7
- Reason for choice: abnormal data to show that this value would be rejected