TEMASEK
JUNIOR COLLEGE

Temasek Junior College
JC H2 Computing
Problem Solving & Algorithm Design 4 – Pseudocoding

1 What is Pseudocoding?

Pseudocoding is a method of describing an algorithm or program design.

It uses control structures and keywords similar to those found in programming languages, but without the strict rules of programming languages.

2 Comments

In writing pseudocode, comments are preceded by two forward slashes `//`.
The comment continues until the end of the line. For multi-line comments, each line is preceded by `//`.
Normally, the comment is on a separate line before, and at the same level of indentation as, the code it refers to.
Occasionally, however, a short comment that refers to a single line may be at the end of the line to which it refers.
Comments should include name of programmer, date written, program description and version for book-keeping and/or control e.g.

```
// Name of programmer: Fong KK
// Date written/updated: 02/01/2020
// Program description and version 1.0.0
// This procedure swaps values of X and Y
```

Comments are used to make the algorithm/program/code easier to understand or help (other) designers/programmers understand the algorithm/program/code.

3 Variables, Constants and Data Types

3.1 Atomic Type Names

The table below gives the keywords used to designate atomic data types.

Keyword	Data Type
`INTEGER`	A whole number
`REAL`	A real number (capable of containing a fractional part)
`CHAR`	A single character
`STRING`	A sequence of zero or more characters
`BOOLEAN`	The logical values TRUE and FALSE
`DATE`	A valid calendar date.

3.2 Literals

Literals of the above atomic data types are written as shown in the following table.

Atomic Data Type	Written Format
`INTEGER`	Written as per normal following the denary number system. e.g. 5, -3
`REAL`	Always written with at least one digit on either side of the decimal point, zeros being added if necessary. e.g. 4.7, 0.3, -4.0, 0.0
`CHAR`	A single character delimited by single quotes e.g. 'x', 'C', '@'
`STRING`	Delimited by double quotes. A string may contain no characters (i.e. the empty string). e.g. "This is a string", "" (i.e. the empty string)
`BOOLEAN`	TRUE, FALSE
`DATE`	Normally be written in the format dd/mm/yyyy. Nevertheless, it is good practice to state explicitly that this value is of data type DATE and to explain the format (as the convention for representing dates varies across the world).

3.3 Identifiers

Identifiers (the names given to variables, constants, procedures and functions) are written in **mixed case**.

The following rules apply when writing identifiers:

- Can only contain letters (`A-Z`, `a-z`),
- Digits (`0-9`)
- Underscore "`_`".
- Must start with a letter
- **Cannot** start with a digit.
- Accented letters (e.g. `á`, `ü`) and other characters (e.g. `$`, `&`) should not be used.

Examples of valid identifiers are `CTGroup`, `Age`, `Height`, `AgeOfJC2`, `New_Batch`.

In writing pseudocode, identifiers should be assumed to be **case insensitive**, for example, `ExamQuestions` and `Examquestions` should not be used as separate variables. This is slightly different from actual programming where case sensitivity matters.

As in programming, it is good practice to **use meaningful identifier names** that describe the variable, procedure or function they refer to (**why**?).

Single letters may be used where these are conventional (such as `i` and `j` when dealing with array indices, or `X` and `Y` when dealing with coordinates) as these are made clear by the convention.

Keywords identified should never be used as variables e.g. `INTEGER`, `PRINT`, `WRITE`.

3.4 Variable Declaration

It is good practice to declare variables explicitly in pseudocode (why?).
Declarations are made as follows:

```
DECLARE <identifier> : <data type>
```

e.g.

```
DECLARE Age : INTEGER
DECLARE School_Fee : REAL
DECLARE Promotional_Status : BOOLEAN
DECLARE Date_Of_Birth : DATE
```

When we input data for a process, individual values need to be stored in the memory. We will hence need to be able to refer to a specific memory location so that we can write statements of what to do with the value stored there. We refer to these named memory locations as variables.
You can imagine these variables like boxes with name labels on them. When a value is input, it is stored in the box with the specified name (identifier) on it (**see Fig 1**).

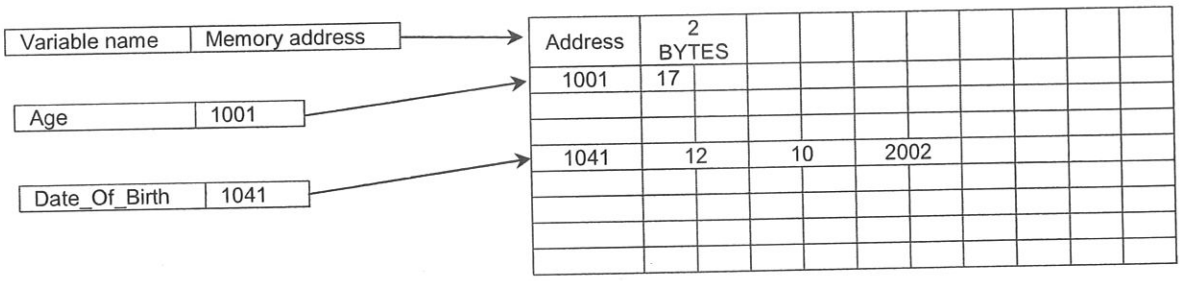


Fig 1: Diagrammatic representation how data is stored in the memory

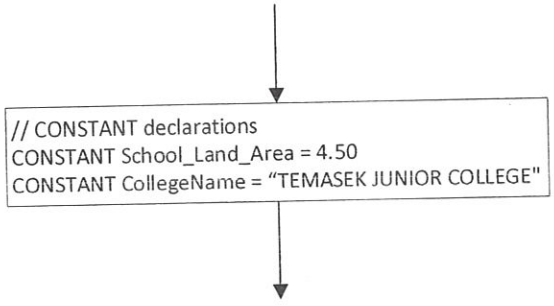
3.5 Constants

It is good practice to use constants if this makes the pseudocode more readable, as an identifier is more meaningful in many cases than a literal. It also makes the pseudocode easier to update if the value of the constant changes.
Constants are normally declared at the beginning of a piece of pseudocode (unless it is desirable to restrict the scope of the constant).
Constants are declared by stating the identifier and the literal value in the following format:

```
CONSTANT <identifier> = <value>
```

e.g.

```
CONSTANT School_Land_Area = 4.50
CONSTANT CollegeName = "TEMASEK JUNIOR COLLEGE"
```



Only literals can be used as the value of a constant. A variable, another constant or an expression must never be used.

3.6 Solution Clarity

The clarity of the programming solution may be enhanced through **comments**, **indentations** and use of **meaningful identifiers**. Hence in writing pseudocode it is good practice to

- Declare meaningful parameters and procedure names.

  e.g.
  ```
  // Constant Declarations
  CONSTANT CollegeName = "TEMASEK JUNIOR COLLEGE"
  ```

  e.g.
  ```
  FUNCTION Deposit(Loan_Amount: REAL) RETURN REAL
  ```

- Declare meaning variable identifiers that are self-documenting whenever possible to ensure the purpose is clear. The use of the same identifier for the same variable in different programs enhances clarity.

  e.g.
  ```
  //Variable Declarations
  DECLARE Age : INTEGER
  DECLARE School_Fee : REAL
  ```

- Have different sections of code clearly identified through the use of blank lines, indentation and comments.

  e.g.
  ```
  //The local time will change to International time
  //Extract the local time form the system clock …
  ```

  e.g.
  ```
  IF Age > 22 THEN   //JC student cannot be older than 22
      Registration ← False
  ELSE
  …
  ENDIF
  ```

3.7 Assignments

The assignment operator is the ← symbol.

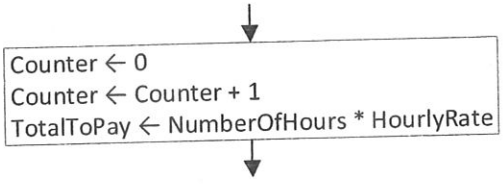
Assignments should be made in the following format:

```
<identifier> ← <value>
```

The identifier must refer to a variable (this can be an individual element in a data structure such as an array or an abstract data type). The value may also be any expression that evaluates to a value of the same data type as the variable.

e.g.

```
Counter ← 0
Counter ← Counter + 1
TotalToPay ← umberOfHours * HourlyRate
```



4 Operators

(A) Arithmetic Operations

Standard arithmetic operator symbols are used to represent arithmetic operations.

+	Addition
-	Subtraction
*	Multiplication
/	Division

(B) Comparison Operations

The following comparison operators are used to write statements involving comparison of values.

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

(C) Logical Operations

The following logical operators are used to write statements involving logical operations.

```
AND
OR
NOT
```

5 Basic Computer Operations

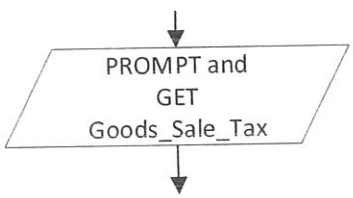
5.1 A Computer can Receive Information (INPUT)

When a computer is required to receive information or input from a particular source, whether it be a terminal, a disk or any other device, the verbs `READ` and `GET` are used in pseudocode.

- `READ` is usually used when the algorithm is to receive input from a record on a file.
- `GET` is used when the algorithm is to receive input from the keyboard.

e.g.

```
READ student_name
GET system_date
READ number_1, number_2
GET Goods_Sale_Tax
```



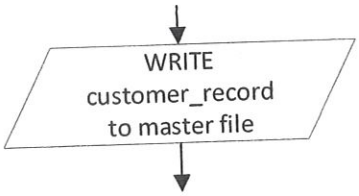
5.2 A Computer can Generate Information (OUTPUT)

When a computer is required to supply information or output to a device, the verbs `PRINT`, `WRITE`, `OUTPUT` or `DISPLAY` are used in pseudocode.

- `PRINT` is usually used when the output is to be sent to printer.
- `WRITE` is used when the output is to be written to a file.
- `PUT`, `OUTPUT` or `DISPLAY` is used when the output is to be written to the screen.

e.g.

```
PRINT "Program Completed"
WRITE customer_record to master file
PUT out name, address and postcode
DISPLAY "End of file"
```



5.3 A Computer can Assign (←) a Value to a Variable or Memory Location

Three cases:

(A) To give data an initial value in pseudo code.

Example

```
Initialise total_accumulators to zero
Set student_count to 0
```

(B) To assign a value as a result of some processing.

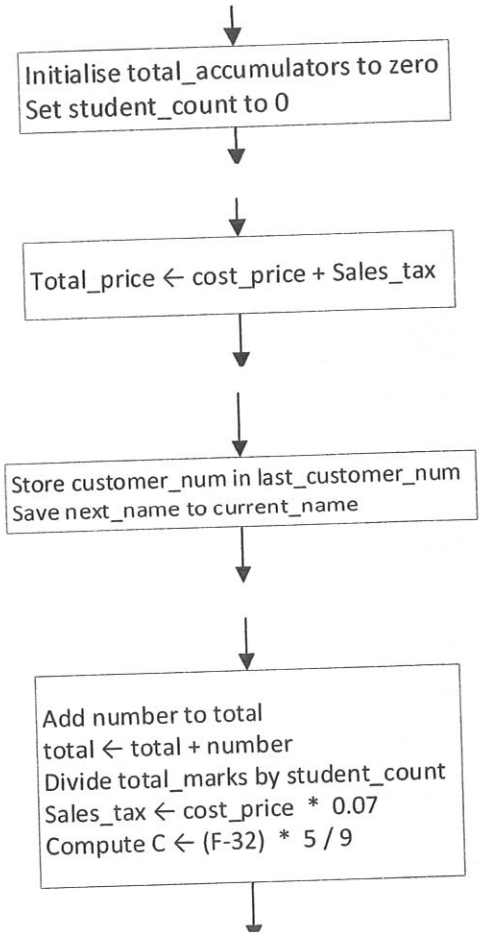
Example

```
Total_price ← cost_price + Sales_tax
```

(C) To keep a piece of information for later use.

Example

```
Store customer_num in last_customer_num
Save next_name to current_name
```



5.4 A Computer can Perform Arithmetic

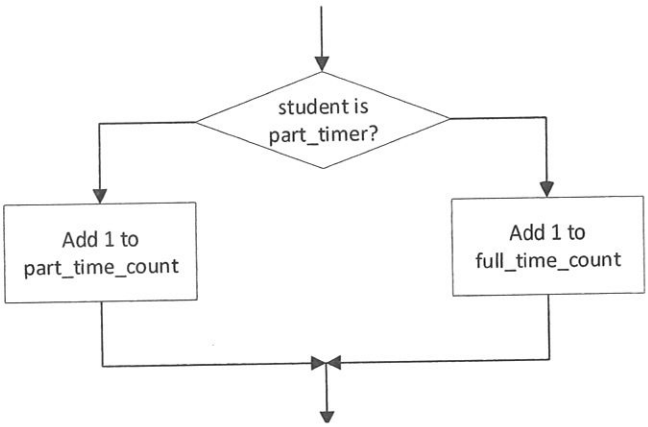
Example

```
Add number to total
total ← total + number
Divide total_marks by student_count
Sales_tax ← cost_price  *  0.07
Compute C ← (F-32)  *  5 / 9
```

5.5 A Computer can Compare Two Variables and Select One of Two Alternative Actions

Example

```
IF student is part_timer
    THEN
        Add 1 to part_time_count
    ELSE
        Add 1 to full_time_count
ENDIF
```



5.6 A Computer can Repeat a Group of Actions

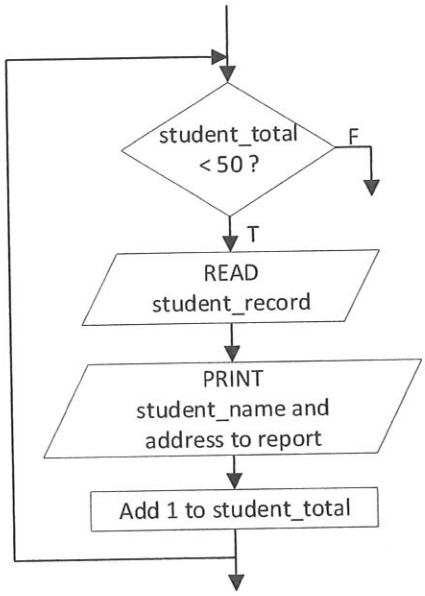
Example [WHILE-ENDWHILE]

```
WHILE student_total < 50
    READ student_record
    PRINT student_name, address to report
    Add 1 to student_total
ENDWHILE
```

Example [DOWHILE-ENDO]

```
DOWHILE student_total < 50
    READ student_record
    PRINT student_name, address to report
    Add 1 to student_total
ENDDO
```

Both the WHILE-ENDWHILE and DOWHILE-ENDO loops can be represented using the flowchart shown in on the next page.



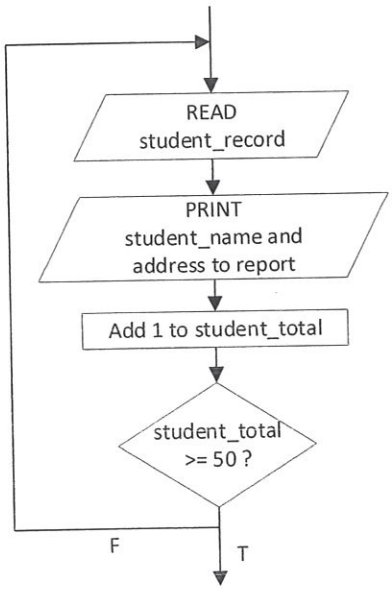
Example [REPEAT-UNTIL]

```
REPEAT
    READ student record
    PRINT student name, address to report
    Add 1 to student_total
UNTIL student_total >= 50
```

Example [FOR-ENDFOR]

```
FOR student_total ← 1 TO 50 STEP 1
    READ student record
    PRINT student name, address to report
ENDFOR
```

Both the REPEAT-UNTIL and FOR-ENDFOR loops can be represented using the flowchart shown below

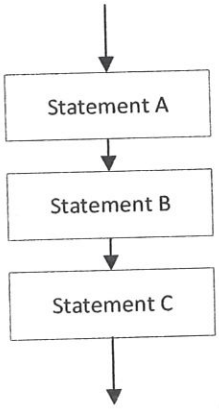


6 The Structure Theorem (Structured Algorithm/Program)

- A **structure** is a basic unit of programming logic; each structure is a **sequence, selection (conditional)** or **iteration/repetition (loop)**.
- Any program, no matter how complicated, can be constructed using these **three** constructs.
- In a **structured algorithm/program**, only three basic control structures are used;
  - ***Sequence*** (one instruction after another)
  - ***Selection*** (e.g. `IF-ELSE` statements) a
  - ***Iteration*** (e.g. `WHILE-ENDWHILE` loops).
- The Structure Theorem forms the basic framework for **structured programming**.

6.1 Sequence

The **sequence control structure** is the straightforward execution of one processing step after another.



Statement A
Statement B
Statement C

Example
Exchanging the values of two variables (Swapping)
Given two variables, A and B, exchange the values assigned to them.

[Solution]
Algorithm description:
(i) Save the original value of A in Temp
(ii) Assign to A the original value of B
(iii) Assign to B the I value of Temp

Pseudocode:

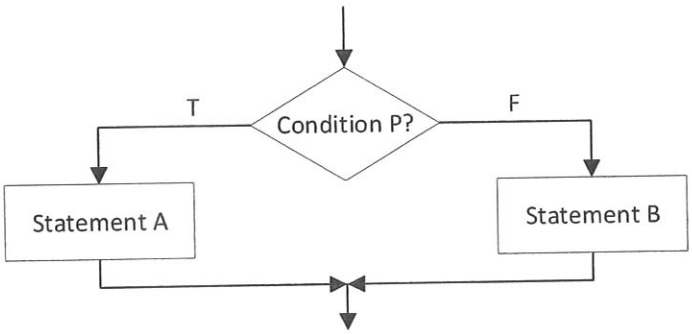
6.2 Selection

The **selection control structure** is the representation of a condition and the choice between two actions. The choice depends on whether the condition is true or false.

(A) Simple Selection (Simple `IF` Structure)

Simple selection occurs when a choice is made between two alternative paths, depending on the result of a condition being true or false. The structure is represented in pseudocode using the keywords `IF`, `THEN`, `ELSE` and `ENDIF`.

```
IF condition p is true
  THEN
    Statement(s) in true case
  ELSE
    Statement(s) in false case
ENDIF
```



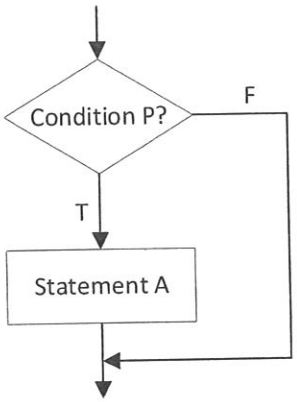
Example

```
IF account_balance < $500
    THEN
        service_charge = $5.00
    ELSE
        service_charge = $2.00
ENDIF
```

(B) Simple Selection with Null False Branch (Null `ELSE` Structure)

The null `ELSE` structure is a variation of the simple `IF` structure. It is used when a task is performed only when a particular condition is true. If the condition is false, then no processing will take place and the `IF` statement will be bypassed.

```
IF condition p is true
    THEN
        Statement(s) in true case
ENDIF
```



Example

```
IF student_attendance = part_time
    THEN
        add 1 to part_time_count
ENDIF
```

(C) Combined Selection (Combined IF Structure)

A combined IF structure is one that contains multiple conditions, each connected with the logical operators AND or OR.

(1) The AND Case

If the conditions are combined using the connector **AND**, **both conditions must be true** for the combined condition to be true e.g.

```
IF student_attendance = part__time AND student_gender = female
    THEN
        add 1 to fem_part_time_count
ENDIF
```

In this case, each student record will undergo two tests. Only those students who are female and who attend part time will be selected, and the variable fem_part_time_count will be incremented. If either condition is found to be false, the counter will remain unchanged.

(2) The OR Case

If the connector OR is used to combine any two conditions, **at least one of the conditions needs to be true** for the combined condition to be considered true. If neither condition is true, the combined condition is considered false.

```
IF student_attendance = part__time OR student_gender = female
    THEN
        add 1 to fem_part_time_count
ENDIF
```

In this example, if either or both conditions is found to be true, the combined condition will be considered true. That is, the counter will be incremented:

- if the student is part time, regardless of gender; **OR**
- if the student is female, regardless of attendance pattern.

Only students who are not female and not part time will be ignored. The fem_part_time_count will contain the total count of

- female part-time students,
- male part-time students, and
- female full-time students.

As a result, fem_part_time_count is no longer a meaningful name for this variable.

As can be seen, changing the AND operator to the OR operator dramatically changes the outcome from the processing of the IF structure. **Hence you must fully understand the processing that takes place when combining conditions with the AND or OR logical operators.**

More than two conditions can be linked together with the AND or OR operators. However, if both operators are used in one IF structure, **parentheses must be used to avoid ambiguity**.

Consider the following example:

```
IF record_code = '23' OR update_code = delete AND account_balance = zero
    THEN
        delete customer record
ENDIF
```

The above pseudocode is **ambiguous**. It is uncertain whether the first two conditions should be grouped together and operated on first, or the second and third conditions should be grouped together and operated on first.

Pseudocode algorithms should never be ambiguous.

While there are rules of precedence for logical operators in most programming languages, there are no rules of precedence for logical operators in pseudocoding. Therefore **parentheses must be used** in pseudocode to avoid ambiguity as to the meaning intended e.g.

```
IF (record_code = '23' OR update_code = delete) AND account_balance =
zero
    THEN
        delete customer record
ENDIF
```

The pseudocode is now no longer ambiguous, and it is clear as to what conditions are necessary for the customer record to be deleted: the record will only be deleted if the `account_balance = zero` and either the `record_code = 23` or the `update code = delete`.

(3) The NOT Case

The NOT operator can also be used for the logical negation of a condition, as follows:

```
IF NOT (record_code = '23')
    THEN
        update customer record
ENDIF
```

Here, the IF structure will be executed for all recordcodes other than code '23', i.e. for record codes not equal to '23'.

Note that the AND and OR operators can also be used with the NOT operator, but great care must be taken and **parentheses must be used to avoid ambiguity**, as follows:

```
IF NOT (record_code = '23' AND updote_code = delete)
    THEN
        update customer record
ENDIF
```

Here, the customer record will only be updated if the record code is not equal to '23' and the update code is not equal to delete.

(D) Nested Selection (Nested `IF` Structure)

Nested selection occurs when the word `IF` appears more than once within an `IF` statement. Nested `IF` statements can be classified as linear or non-linear.

(1) Linear nested `IF` statements

The linear nested `IF` statement is used when a field is being tested for various values and a different action is to be taken for each value.

This form of nested `IF` is called linear because each `ELSE` condition immediately follows the `IF` condition to which it corresponds. Comparisons are made until a true condition is encountered, and the specified action is executed until the next `ELSE` statement is reached.

Linear nested `IF` statements should be indented for readability, with each `IF` and corresponding `ENDIF` aligned.

Example

```
IF record_code = 'A'
    THEN
        increment counter_A
    ELSE
        IF record_code = 'B'
            THEN
                increment counter_B
            ELSE
                IF record_code = 'C'
                    THEN
                        increment counter_C
                    ELSE
                        increment error_counter
                ENDIF
        ENDIF
ENDIF
```

Note that there are an equal number of `IF`, `ELSE` and `ENDIF` statements, and that the correct indentation makes it easy to read and understand.

(2) Non-linear nested IF statements

A non-linear nested IF occurs when a number of different conditions need to be satisfied before a particular action can occur.

It is termed non-linear because the ELSE statement may be separated from the IF statement with which it is paired. Indentation is once again important when expressing this form of selection in pseudocode. Each ELSE statement should be aligned with the IF condition to which it corresponds. For instance:

```
IF student_attendance = part_time
    THEN
        IF student_gender = female
            THEN
                IF student_age > 21
                    THEN
                        add 1 to mature_fem_pt_students
                    ELSE
                        add 1 to young_fem_pt_studenfs
                ENDIF
            ELSE
                add 1 to male_pt_students
        ENDIF
    ELSE
        add 1 to full_time_students
ENDIF
```

Note that there is an equal number of IF conditions as ELSE and ENDIF conditions. Using correct indentation helps to see which pair of IF and ELSE conditions match.

Non-linear nested IF statements may contain logic errors that could be difficult to correct. They should hence be used sparingly in pseudocode. If possible, replace a series of non-linear nested IF statements with a combined IF statement. This is possible in pseudocode as two consecutive IF statements act like a combined IF statement using the AND operator.

Consider the following non-linear nested IF structure:

```
IF student_attendance = part_time
    THEN
        IF student_age > 21
            THEN
            increment mature_pt_student
        ENDIF
ENDIF
```

This can be written as a combined IF structure:

```
IF student_attendance = part_time AND student_age > 21
    THEN
        increment mature_pt_student
ENDIF
```

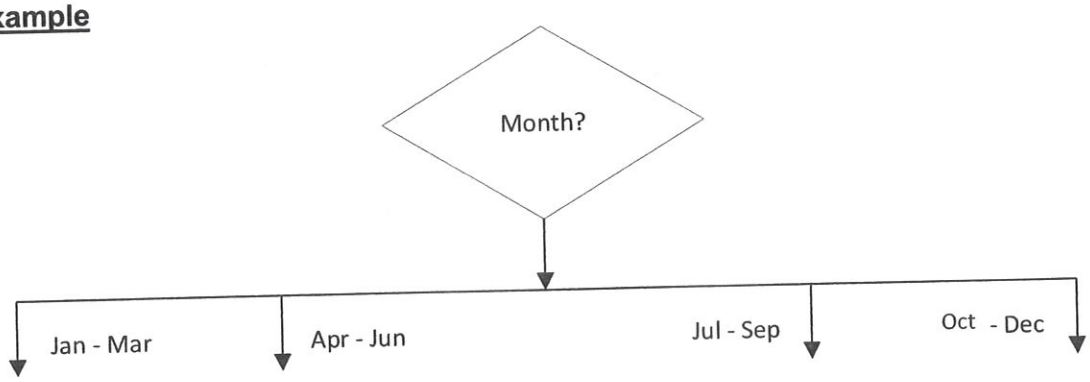
The same outcome will occur for both pseudocode expressions, but the format of the latter is preferred, if the logic allows it, simply because it is easier to understand.

(E) CASE Statements

CASE statements allow one out of several branches of code to be executed, depending on the value of a variable. CASE statements are written as follows:

```
CASE OF <identifier>
    <value 1> : <statement>
    <value 2> : <statement>
    ...
ENDCASE
```

Example



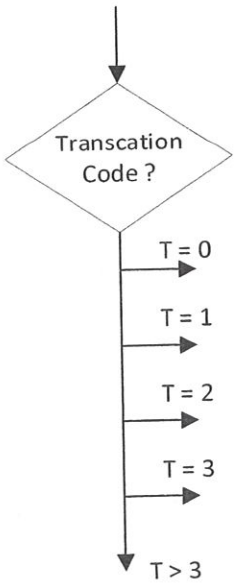

```
CASE OF Month
    Jan-Mar : <statement>
    Apr-Jun : <statement>
    Jul-Sep : <statement>
    Oct-Dec : <statement>
ENDCASE
```

An OTHERWISE clause can be the last case:

```
CASE OF <identifier>
    <value 1> : <statement>
    <value 2> : <statement>
    ...
    OTHERWISE <statement>
ENDCASE
```

Example

```
CASE OF TransactionCode
    0 : <statement>
    1 : <statement>
    2 : <statement>
    3 : <statement>
    OTHERWISE <statement>
ENDCASE
```



6.3 Iteration/Repetition

The **iteration/repetition control structure** can be defined as the representation of a set of instructions to be performed repeatedly, as long as condition is true.

(A) Pre-condition loops (`DOWHILE-ENDDO / WHILE-ENDWHILE`),

As the name suggests, a pre-condition loop involves evaluating the condition before the statements within the loop are executed.

A pre-condition loop will execute the statements within the loop as long as the condition evaluates to True.

When the condition evaluates to False, execution will go to the next statement after the loop.

Note that any variable used in the condition must not be undefined when the loop structure is first encountered.

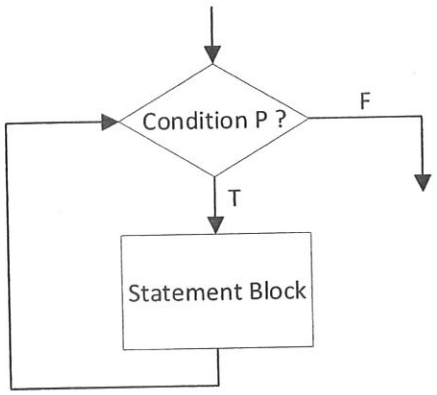
When coding a pre-condition loop, you must ensure that there is a statement within the loop that will at some point change the value of the controlling condition. Otherwise the loop will execute forever (infinite loop).

`DOWHILE-ENDDO` Loop

```
DOWHILE condition p is true
    Statement block
ENDDO
```

`WHILE-ENDWHILE` Loop

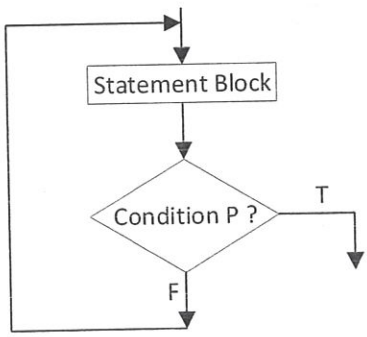
```
WHILE condition p is true
    Statement Block
ENDWHILE
```



(B) Post-condition loops (`REPEAT-UNTIL`),

```
REPEAT
    Statement block
UNTIL condition p is true
```

The condition must be an expression that evaluates to a Boolean.



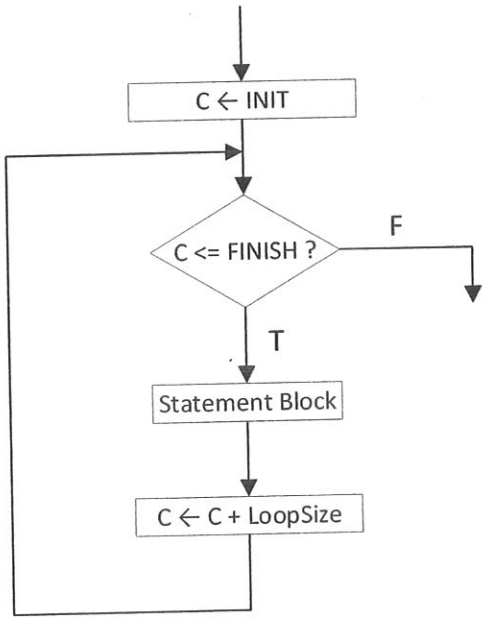
(C) Count-controlled loops (FOR-ENDFOR),

The **FOR** loop provides you with three actions in one compact statement. The statement uses a loop control variable that automatically:

- Initializes
- Evaluates
- Increments

The control variable starts with value `INIT`, increments by value `LoopSize` each time round the loop and finishes when the control variable reaches the value `FINISH`.

```
FOR C ← INIT TO FINISH [STEP LoopSize]
    Statement block
ENDFOR or NEXT or NEXT C
```



A `FOR` Loop is a definite loop when you know how many times a loop will repeat.

Writing pseudocodes for the following questions

Tutorial 2.4 Q – 1, 2, 3, 6, 7, 9, 10
Tutorial 2.5 Q – 1, 2, 4, 5

Tutorial 4 Q – 1, 2, 3, 4, 5

Tutorial 4

1* Below is a segment of an algorithm that **attempts** to determine the least number of coins that can be used to make up the number `AmountInCent` which is input by the user. This algorithm will not produce the correct output in every case

```
input AmountInCent
AmountLeft ← AmountInCent

if AmountLeft >= 50
    then
        output " 50¢ "
        AmountLeft ← AmountLeft - 50
endif

if AmountLeft >= 20
    then
        output " 20¢ "
        AmountLeft ← AmountLeft - 20
endif

if AmountLeft >= 10
    then
        output " 10¢ "
        AmountLeft ← AmountLeft - 10
endif

if AmountLeft >= 5
    then
        output " 5¢ "
        AmountLeft ← AmountLeft - 5
endif

if AmountLeft >= 1
    then
        output " 1¢ "
        AmountLeft ← AmountLeft - 1
Endif
```

(a) Using the algorithm above, write down all the outputs, produced for the following test data:

Test data: AmountInCent = 36
Test data: AmountInCent = 97
Test data: AmountInCent = 44

[3]

(b) Briefly describe why the algorithm does not produce the correct output in every case [1]

[ ]

2* A program is to be written to calculate the discount given on purchases.

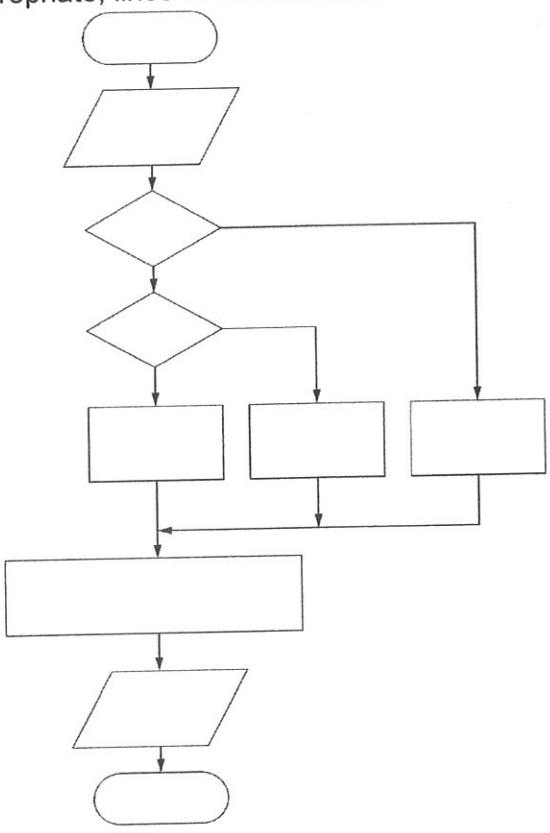
A purchase may qualify for a discount depending on the amount spent. The purchase price (Purchase), the discount rate (`DiscountRate`) and amount paid (`Paid`) is calculated as shown in the following pseudocode algorithm.

```
INPUT Purchase

IF Purchase > 1000
    THEN
        DiscountRate ← 0.10
    ELSE
        IF Purchase > 500
            THEN
                DiscountRate ← 0.05
            ELSE
                DiscountRate ← 0
        ENDIF
ENDIF

Paid ← Purchase * (1 - DiscountRate)
OUTPUT Paid
```

The algorithm is also to be documented with a program flowchart.

Copy and complete the flowchart by:

- filling in the flowchart boxes
- labelling, where appropriate, lines of the flowchart



3* [FOR-ENDFOR Loops]
The following test scores are given: 65, 71, 82, 63, 90, 58, 66, 67, and 68. Using a trace table to find the largest and smallest of test scores.

```
CONSTANT ScoreSize = 9
DECLARE Count, Score, CurrentLargest, CurrentSmaller : INTEGER

CurrentLargest ← -1 // Dummy value -1, why? Any other alternatives?
CurrentSmaller ← -1

FOR Count ← 1 TO ScoreSize

    PRINT "Enter score of number " & Count & ":"
    READ Score

    IF Score > CurrentLargest
        THEN
            CurrentLargest ← Score
    ENDIF

    IF Score < CurrentSmaller
        THEN
            CurrentSmaller ← Score
    ENDIF

ENDFOR

PRINT "The smallest test scores: ", CurrentSmaller
PRINT "The largest test scores: ", CurrentLargest
```

4* The loop construct (also known as repetition or iteration) appears in many algorithms.

Use **pseudocode** to write a **post-condition loop** to output all the odd numbers between 100 and 200.

5* Below is an algorithm.

```
Algorithm FindTotal
HighNum is integer {input by user}
Total is integer
i is integer

startmainprog

    input HighNum
    set Total = 0

    if HighNum < 1
        then
            output "Number not valid"
        else
            for i = 1 to HighNum
                set Total = Total + i
                output Total
            endfor
    endif

endmainprog
```

Write down all the outputs in the correct order produced by the algorithm for the inputs given below.

(a) Input is 0

(b) Input is 4