7. Algorithm Design & Problem Solving

The program development life cycle

1. Analysis

Involves using tools like abstraction and decomposition to identify the specific requirements for the program.

a. <u>Abstraction</u>

- simplifying the problem
- removing unnecessary details from problem // selecting elements required
- filtering out irrelevant characteristics from those elements

b. <u>Decomposition:</u> Breaking down a large problem into smaller, clear, manageable and understandable sub-parts/sub-systems that are easier to solve and create.

Component parts when a problem has been decomposed at the analysis stage

- <u>Inputs:</u> data entered into the system // what is put into the system
- <u>Processes:</u> subroutines and algorithms that turn inputs and stored data into outputs // actions taken to achieve a result
- <u>Outputs:</u> data that is produced by the system (eg. printed information) // what is taken out of the system
- <u>Storage:</u> data that is stored in files/ on physical device (eg. hard drive) // what needs to be kept for future use

c. <u>Identification of the problem:</u>

- Before tackling the problem, it needs to be clearly understood by everyone working on it.
- The overall goal of the solution needs to be agreed.
- Constraints (limited resources / requiring platform specific solution) need to be agreed.

d. <u>Requirements specification</u>

- A requirements document is created to define the problem and break it down into clear, manageable, understandable parts by using abstraction and decomposition.
- Requirements document labels each requirement, gives it a description and success criteria - states how we know when the requirement has been achieved.

2. Design

Details of solution set out

Decomposition

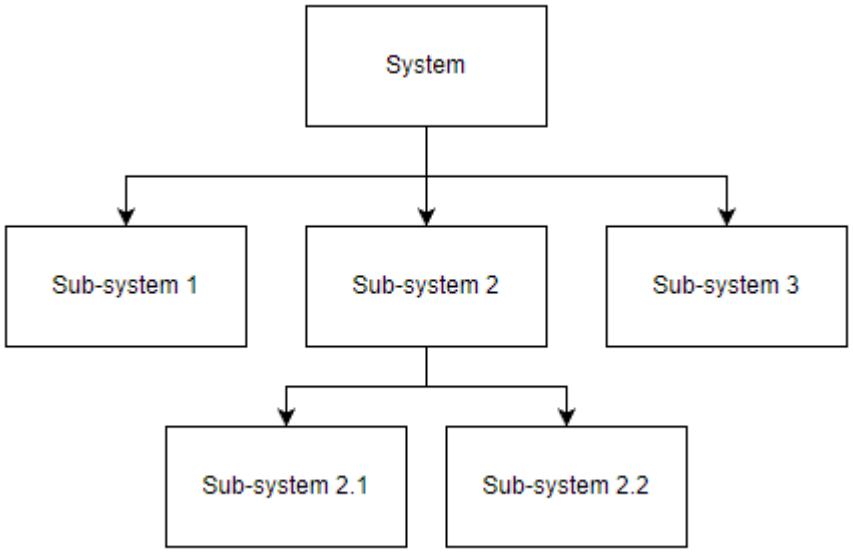
Top-down design: decomposition of a computer system into subsystems, which can further be divided into more sub-systems, until each subsystem performs a single action.
Stepwise refinement: the process of breaking down a problem into sub-systems.

- Each sub-system can be assigned to a developer / group of developers who create subroutines from these sub-systems.
- Sub-system can be created/tested simultaneously, reducing development & testing time.

Methods to design & construct solutions to problems

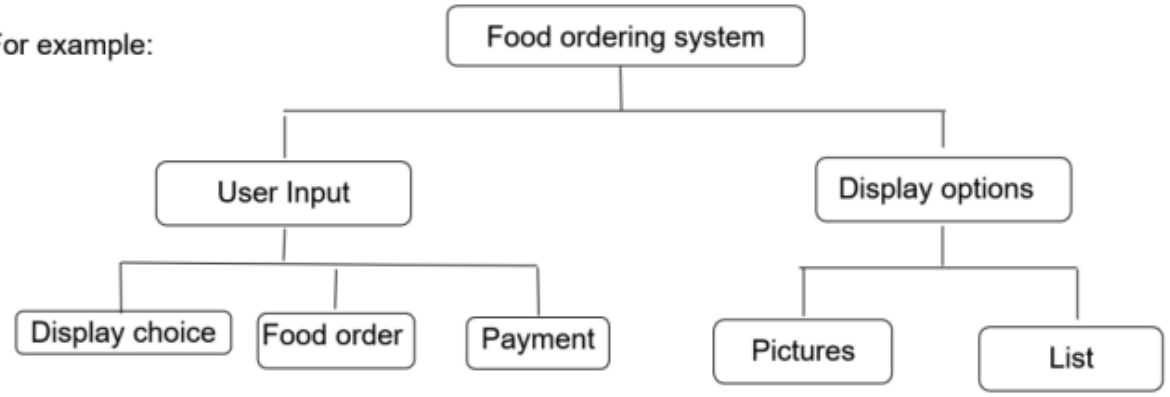
Structure diagrams

- They show the breakdown of tasks/systems into subtasks/subsystems
- Hierarchical & top-down design



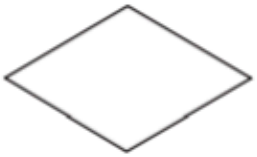
A food ordering system is an example of a computer system that is made up of sub-systems. The food ordering system:
• allows the user to enter details of the food they want to order and to pay for the order
• displays food available as pictures or as a list.
Complete the structure diagram for the given parts of the food ordering system.

For example:



Flowcharts

- Diagrammatically shows the steps required to complete a task, in order
- Visual representation of algorithm.

	Flow line	An arrow represents control passing between the connected shapes.
	**Process**	This shape represents something being performed or done.
	**Subroutine**	This shape represents a subroutine call that will relate to a separate, non-linked flowchart.
	**Input/Output**	This shape represents the input or output of something into or out of the flowchart.
	**Decision**	This shape represents a decision (Yes/No or True/False) that results in two lines representing the different possible outcomes.
	**Terminator**	This shape represents the 'Start' and 'Stop' of the process.

Pseudocode

- Uses English-like keywords, statements & identifiers to represent instructions, similar to programming code
- Not bound by strict syntax rules
- Allows developers to understand how to create a program & implement a solution
- Uses a consistent style:
    - Non-proportional font is used throughout
    - Keywords are written in capital letters
    - Variable and subroutine names start with capital letters
    - Indentation is used for iteration and selection

three different ways that the design of a solution to a problem can be presented.

- Structure diagram
- Flowchart
- Pseucocode

3. Coding

The program is developed.

- **Writing program code:** Developers write each module of the program using suitable programming language to provide an overall solution to the problem.
- **Iterative testing:** Each module is tested & debugged thoroughly to ensure it interacts correctly with other modules and accepts data without crashing or causing error.

4. Testing

The program is tested for errors

The completed program/set of programs is tested using test data

Standard methods of solution

Totalling

Keeping a running total that values are added to (often used in loops)

- A total variable can be initialised to 0 and then updated within a loop

Examples

1. Keeping running total of marks awarded to each student in a class

```
Total ← 0
FOR Counter ← 1 TO ClassSize
     Total ← Total + StudentMark[Counter]
NEXT Counter
```

2. Totalling a receipt for purchases made at a shop

```
Total ← 0
FOR Count ← 1 TO ReceiptLength
      INPUT ItemValue
      Total ← Total + ItemValue
NEXT Count
OUTPUT Total
```

Counting

Keeps count of the number of times an action has been performed.

- A count variable can be initialised to 0 and then updated within a loop.

Examples

1. Counting the number of students that were awarded a pass mark

```
      PassCount ← 0
      FOR Counter ← 1 TO ClassSize
            INPUT StudentMark
            IF StudentMark > 50
                  THEN
                        PassCount ← PassCount + 1
      NEXT Counter
```

2. Counting down - checking number of items in stock

```
      :
      NumberInStock ← NumberInStock - 1
      IF NumberInStock < 20
            THEN
                  CALL Reorder()
      :
```

Maximum, minimum, average

Examples

1. Finding highest & lowest mark awarded to a class of students

```
      MaximumMark ← 0          //initialising maximum to lowest mark
possible
      MinimumMark ← 100  //initialising minimum to highest mark possible
      FOR Counter ← 1 TO ClassSize
            IF StudentMark[Counter] > MaximumMark
                  THEN
                        MaximumMark ← StudentMark[Counter]
            ENDIF
            IF StudentMark[Counter] < MinimumMark
                  THEN
                        MinimumMark ← StudentMark[Counter]
```

```
            ENDIF
      NEXT Counter
```

2. Finding highest & lowest mark awarded to a class of students - when maximum & minimum marks are not known

```
MaximumMark ← StudentMark[1]
MinimumMark ← StudentMark[1]
FOR Counter ← 2 TO ClassSize //loop starts at 2nd position in list
      IF StudentMark[Counter] > MaximumMark
            THEN
                  MaximumMark ← StudentMark[Counter]
      ENDIF
      IF StudentMark[Counter] < MinimumMark
            THEN
                  MinimumMark ← StudentMark[Counter]
      ENDIF
NEXT Counter
```

- In the above algorithm, the `MaximumMark` and `MinimumMark` are set to the first value in the list and the `FOR` loop starts at the second element instead of the first as the first value is the benchmark to compare all other items to.
- The algorithm loops over each element asking whether the current value is larger than `MaximumMark` and whether it is smaller than `MinimumMark`.

3. Calculating average (mean) mark for class of students

```
Total ← 0
FOR Counter ← 1 TO ClassSize
      Total ← Total + StudentMark[Counter]
NEXT Counter
Average ← Total / ClassSize
```

Linear search

- Used to find elements in an unordered list.
- The list is searched sequentially and systematically from start to end, one element at a time, comparing each element to the value being searched for.
- If the value is found the algorithm outputs where it was found in the list.
- If the value is not found it outputs a message stating it is not in the list.

Examples

1. Searching for a name in a class list of student names

```
OUTPUT "Please enter name to find"
INPUT Name
Found ← FALSE
Counter ← 1
```

```
REPEAT
      IF Name = StudentName[Counter]
         THEN
            Found ← TRUE
       ELSE
           Counter ← Counter + 1
      ENDIF
UNTIL Found OR Counter > ClassSize
IF Found
      THEN
            OUTPUT Name, " found at position ", Counter, " in
the list."
      ELSE
            OUTPUT Name, " not found."
ENDIF
```

2. Searching for a number in a list

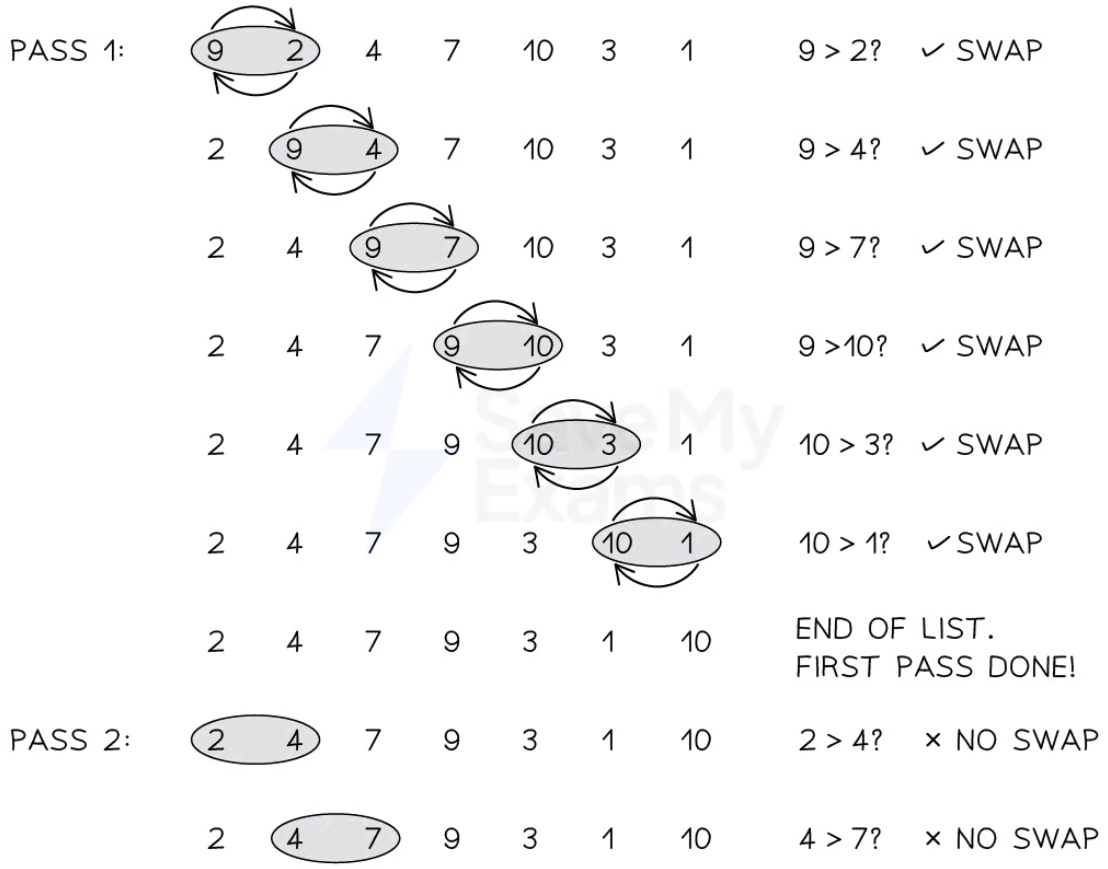
```
OUTPUT "Enter a value to find"
INPUT Number
Found ← FALSE
Index ← 1
REPEAT
      IF Number = Mylist[Index]
            THEN
                  Found ← TRUE
            ELSE
                  Counter ← Counter + 1
      ENDIF
UNTIL Found = TRUE OR Counter > LENGTH(Mylist)

IF Found = TRUE
      THEN
            OUTPUT Number, " found at position ", Index
      ELSE
            OUTPUT Number, " not found"
ENDIF
```

3. Checking for how many people chose ice cream as their favourite dessert

```
      ChoiceCount ← 0
      FOR Counter ← 1 TO Length
      IF "ice cream" = Dessert[Counter]
            THEN
                  ChoiceCount ← ChoiceCount + 1
      NEXT Counter
      OUTPUT ChoiceCount, " chose ice cream as their favourite
dessert."
```

Bubble sort

- Sorts items into order, smallest to largest, by comparing pairs of elements and swapping them if they are out of order.
- The first element is compared to the second, the second to the third, the third to the fourth and so on, until the second to last is compared to the last.
- Swaps occur if a comparison is out of order. This overall process is called a pass.
- Once the end of the list has been reached, the value at the top of the list is now in order and the sort resets back to the start of the list. The next largest value is then sorted to the top of the list.
- More passes are completed until all elements are in the correct order.
- A final pass checks all elements and if no swaps are made then the sort is complete.



Examples

1. To arrange a list of numbers in ascending order

```
Mylist ← [5, 9, 4, 2, 6, 7, 1, 2, 4, 3]
FirstElement ← 1
LastElement ← LENGTH(Mylist)

REPEAT
  Swap ← FALSE

  FOR Index ← FirstElement TO LastElement - 1
```

```
      IF Mylist[Index] > Mylist[Index + 1]
          THEN
            Temp ← Mylist[Index]
            Mylist[Index] ← Mylist[Index + 1]
            Mylist[Index + 1] ← Temp
            Swap ← TRUE
      ENDIF
    NEXT Index
    LastElement ← LastElement - 1

  UNTIL Swap = FALSE OR LastElement = 1
  OUTPUT "Your sorted list is:", Mylist
```

2. To sort a list of 10 temperatures in an array into ascending order

```
    First ← 1
    Last ← 10
    REPEAT
        Swap ← FALSE
        FOR Index ← First TO Last - 1
            IF Temperature[Index] > Temperature[Index+1]
                THEN
                    Temp ← Temperature[Index]
                    Temperature[Index] ←
Temperature[Index+1]
                    Temperature[Index+1] ← Temp
                    Swap ← TRUE
            ENDIF
        NEXT Index
        Last ← Last - 1
    UNTIL (NOT Swap) OR Last = 1
```

Bubble sort for 2D array:

```
REPEAT
    Flag ← FALSE
    FOR Count ← 1 TO CurrentSize - 1
        IF Contacts[Count,1] > Contacts[Count+1, 1] THEN
            Temp1 ← Contacts[Count,1]
            Temp2 ← Contacts[Count,2]
            Contacts[Count,1] ← Contacts[Count+1, 1]
            Contacts[Count,2] ← Contacts[Count+1, 2]
            Contacts[Count+1,1] ← Temp1
            Contacts[Count+1,2] ← Temp2
            Flag ← TRUE
        ENDIF
    NEXT Count
    CurrentSize = CurrentSize - 1
UNTIL NOT Flag OR CurrentSize = 1
```

Validation & Verification

Used to ensure input data is correct, reasonable and accurate.

Validation

To check if the data entered is possible / reasonable / sensible.

Describe what is meant by data validation.

- Validation is an automated check carried out by a computer
- ... to make sure the data entered is sensible/acceptable/reasonable

Types of validation checks

- Range checks
- Length checks
- Type checks
- Presence checks
- Format checks
- Check digits

Range check

Makes sure that input data lies within specified values/ given parameters.

Example: Checking that percentage marks are between 0 and 100 inclusive

```
OUTPUT "Enter percentage marks"
REPEAT
    INPUT Mark
    IF Mark < 0 OR Mark > 100
       THEN
         OUTPUT "Marks is not between 0 and 100, please try
   again"
   ENDIF
UNTIL Mark >= 0 AND Mark <= 100
```

Length check

Checks if:

- Input data contains an exact number of characters OR
- Input data is a reasonable number of characters/ lies within a user-specified number range of characters

Examples

1. Exact number of characters

```
      OUTPUT "Enter password of 8 characters"
      REPEAT
            INPUT Password
            IF LENGTH(Password) <> 8
                  THEN
                        OUTPUT "Password must be exactly 8
characters. Please try again"
            ENDIF
      UNTIL LENGTH(Password) = 8
```

2. Specified number range of characters

```
      OUTPUT "Enter family name"
      REPEAT
            INPUT FamilyName
            IF LENGTH(FamilyName) > 30 OR LENGTH(FamilyName) < 2
                  THEN
                        OUTPUT "Too short or too long, re-enter"
            ENDIF
      UNTIL LENGTH(FamilyName) <= 30 AND LENGTH(FamilyName) >= 2
```

Type check

Checks that data entered is of a given data type // Checks the type of data entered to make sure no numbers (eg) are present

Example: Checks if value entered is an integer

```
OUTPUT "How many brothers do you have?"
REPEAT
      INPUT Num
      IF Num <> DIV(Num,1)
            THEN
                  OUTPUT "Must be a whole number, please re-enter"
      ENDIF
UNTIL Num = DIV(Num,1)
```

Presence check

Enures/checks that some data has been entered & input box (or field in a database) has not been left blank.

Example: Presence check for login system

```
OUTPUT "Enter your username"
REPEAT
      INPUT Username
      IF Username = ""
            THEN
                  OUTPUT "No username entered, please try again"
```

```
      ENDIF
UNTIL Username <> “”
```

Format check

Makes sure that input data is of a predefined pattern/ format.

- Done using pattern matching and string handling

Example: to check if a six digit identification number follows the format “XX9999” where X is an uppercase alphabetical letter and 9999 is a four digit number

- The first two characters are checked against a list of approved characters. The first character is compared one at a time to each valid character in the ValidChars array. If it finds a match it stops looping and sets ValidChar to True. The second character is then compared one at a time to each valid character in the ValidChars array. If it finds a match then it also stops looping and sets ValidChar to True.
- Casting is used on the digits to turn the digit characters into numbers. Once the digits are considered a proper integer they can be checked to see if they are in the appropriate range of 0-9999.
- If any of these checks fail then an appropriate message is output.

```
INPUT IDNumber
IF LENGTH(IDNumber) <> 6
  THEN
    OUTPUT “ID number must be 6 characters long”
ENDIF

ValidChars ← “ABCDEFGHIJKLMNOPQRSTUVWXYZ”
FirstChar ← SUBSTRING(IDNumber, 1, 1)
Valid ← False
Index ← 1

WHILE Index <= LENGTH(ValidChars) AND Valid = False DO
      IF FirstChar = ValidChars[Index]
         THEN
            Valid ← True
      ENDIF
      Index ← Index + 1
ENDWHILE

IF Valid = False
  THEN
      OUTPUT “First character is not a valid uppercase alphabetic
character”
ENDIF
```

```
SecondChar ← SUBSTRING(IDNumber, 2, 2)
Valid ← False
Index ← 1

WHILE Index <= LENGTH(ValidChars) AND Valid = False DO
      IF SecondChar = ValidChars[Index]
        THEN
            Valid ← True
      ENDIF
      Index ← Index + 1
ENDWHILE

IF Valid = False
  THEN
      OUTPUT "Second character is not a valid uppercase
alphabetical character"
ENDIF

Digits ← INT(SUBSTRING(IDNumber, 3, 6))
IF Digits < 0000 OR Digits > 9999
  THEN
      OUTPUT "Digits invalid. Enter four valid digits in the range
0000-9999"
ENDIF
```

Check digit

Numerical values that are the final digit included in a code. It is calculated by applying an algorithm to the other digits in the code and then attached to the overall code.

They help to identify errors in data entry (mistyping, miscanning, misspeaking) such as:

- Incorrect digits
- Missing/extra digits
- Transposition errors (where 2 numbers have swapped)
- Phonetic errors (eg. 13 instead of 30)

Example

```
Barcode ← "9780201379624"
Total ← 0

FOR Index ← 1 to LENGTH(Barcode) - 1
      IF Index MOD 2 = 0
            THEN
                  Total ← Total + CAST_TO_INT(Barcode[Index])*3
            ELSE
                  Total ← Total + CAST_TO_INT(Barcode[Index])*1
      ENDIF
```

```
NEXT Index

CheckDigit ← 10 - Total MOD 10

IF CheckDigit = Barcode[LENGTH(Barcode)]
  THEN
      OUTPUT "Valid check digit"
  ELSE
      OUTPUT "Invalid check digit"
ENDIF
```

Verification

Why it is required: To check that values are entered as intended.

Explain why verification checks are used when data is input

- To ensure that data has been accurately copied // to ensure that changes have not been made to the values originally intended when data is copied
- ... from one source to another

Types of verification methods

- Double entry
- Screen/visual check

Double entry check

- Ask the user to enter the value twice and compare the values
- the two entries are compared and if they do not match, a re-entry is requested // only accept the value if both entries are identical

Example

```
REPEAT
      OUTPUT "Enter your password"
      INPUT Password
      OUTPUT "Please confirm your password"
      INPUT ConfirmPassword
      IF Password <> ConfirmPassword
            THEN
                  OUTPUT "Passwords do not match, please try again"
      ENDIF
UNTIL Password = ConfirmPassword
```

Visual check

- Displaying the value as it is entered
- the user looks through the data that has been entered and confirms that no changes have been made.

Example

```
REPEAT
      OUTPUT "Enter your name"
      INPUT Name
      OUTPUT "Your name is: ", Name, ". Is this correct? (y/n)"
      INPUT Answer
UNTIL Answer = "y"
```

Statement	Validation (✓)	Verification (✓)	Neither (✓)
a check where data is re-entered to make sure no errors have been introduced during data entry		✓	
an automatic check to make sure the data entered has the correct number of characters	✓		
a check to make sure the data entered is sensible	✓		
a check to make sure the data entered is correct			✓

Statement	Validation	Verification	Both
Entering the data twice to check if both entries are the same.		✓	
Automatically checking that only numeric data has been entered.	✓		
Checking data entered into a computer system before it is stored or processed.			✓
Visually checking that no errors have been introduced during data entry.		✓	

Test data

- Before a system is used, each sub-system is tested to ensure it works correctly and interacts correctly with other sub-systems.
- Programs are tested by running them on a computing device; pseudocode & flowcharts are tested manually (dry-run)
- Testing requires different sets of suitable test data.
- The outputs are compared to expected output to check if algorithm works as intended.

Types of test data

1. **Normal:** Data that should be accepted by the algorithm and processed correctly
2. **Abnormal (erroneous):** Data that should be rejected by algorithm; is expected to fail.
3. **Extreme:** The largest/smallest acceptable value. Eg. 0 and 100 for percentage.
4. **Boundary:** The largest/smallest acceptable value and the corresponding smallest/largest rejected value. Eg. 0 & -1 (for 0); 100 & 101 (for 100), for percentage.

Describe the purpose of test data.

- Checks that the program works as expected
- Checks for logic/ runtime errors
- Checks that the program accepts only reasonable data
- Checks that the program rejects any invalid data that is input

Reasons for your choice of test data (when asked to give examples of test data)

- Normal: to test that normal data is accepted & processed correctly // data is within range and should be accepted
- Erroneous: to test that erroneous data is rejected // data is outside range and should be rejected
- Extreme: Data at the maximum / minimum end of the range and should be accepted

A program has been written to check the value of a measurement. The measurement must be a positive number and given to three decimal places, for example, 3.982. Explain why two pieces of boundary test data are required for this program. Give an example of each piece of boundary test data.

- to test that the highest possible non-positive number is rejected and the lowest possible positive number is accepted
- Sample 1: 0.000 (rejected)
- Sample 2: 0.001 (accepted)

Trace Tables

Dry run: Manually tracing/working through an algorithm/flowchart with test data.

- Used to follow algorithms and make sure they perform the required task correctly.
- They record the state of the algorithm at each step / iteration. The state includes all variables that impact the output.
- A trace table is composed of columns where each variable and the output is a column.
- Whenever a value changes or an output is produced the relevant column and row is updated to reflect the change.

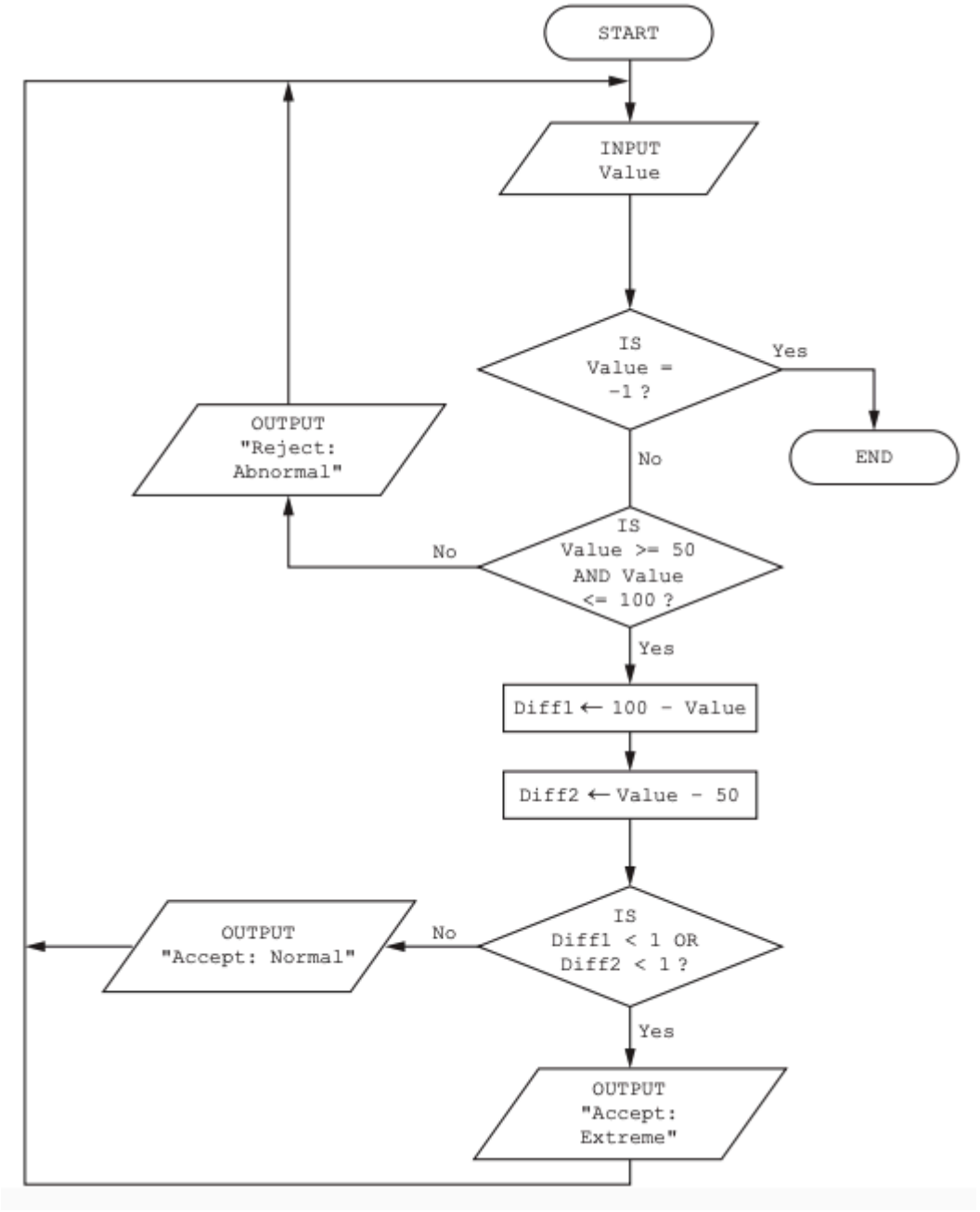
Identifying errors

2 types of error:

1. Syntax error
2. Logical error

Practice using trace tables

Question 1



Complete the trace table for the input data:

50, 75, 99, 28, 82, 150, –1, 672, 80

Value	Diff1	Diff2	OUTPUT

Value	Diff1	Diff2	OUTPUT
50	50	0	Accept: Extreme
75	25	25	Accept: Normal
99	1	49	Accept: Normal
28			Reject: Abnormal
82	18	32	Accept: Normal
150			Reject: Abnormal
–1			

Describe the purpose of the algorithm.

- To output the type of test data
- by performing a range check // by checking if numbers are within the range 50 and 100 (inclusive) or not

Question 2

This algorithm checks passwords.

- Each password must be 8 or more characters in length; the predefined function `Length` returns the number of characters.
- Each password is entered twice, and the two entries must match.
- Either `Accept` or `Reject` is output.
- An input of 999 stops the process.

```
REPEAT
  OUTPUT "Please enter password"
  INPUT Password
  IF Length(Password) >= 8
    THEN
      INPUT PasswordRepeat
      IF Password <> PasswordRepeat
        THEN
          OUTPUT "Reject"
        ELSE
          OUTPUT "Accept"
      ENDIF
    ELSE
      OUTPUT "Reject"
  ENDIF
UNTIL Password = 999
```

(a) Complete the trace table for the algorithm using this input data:
Secret, Secret, VerySecret, VerySecret, Pa55word, Pa55word, 999, 888

Password	PasswordRepeat	OUTPUT

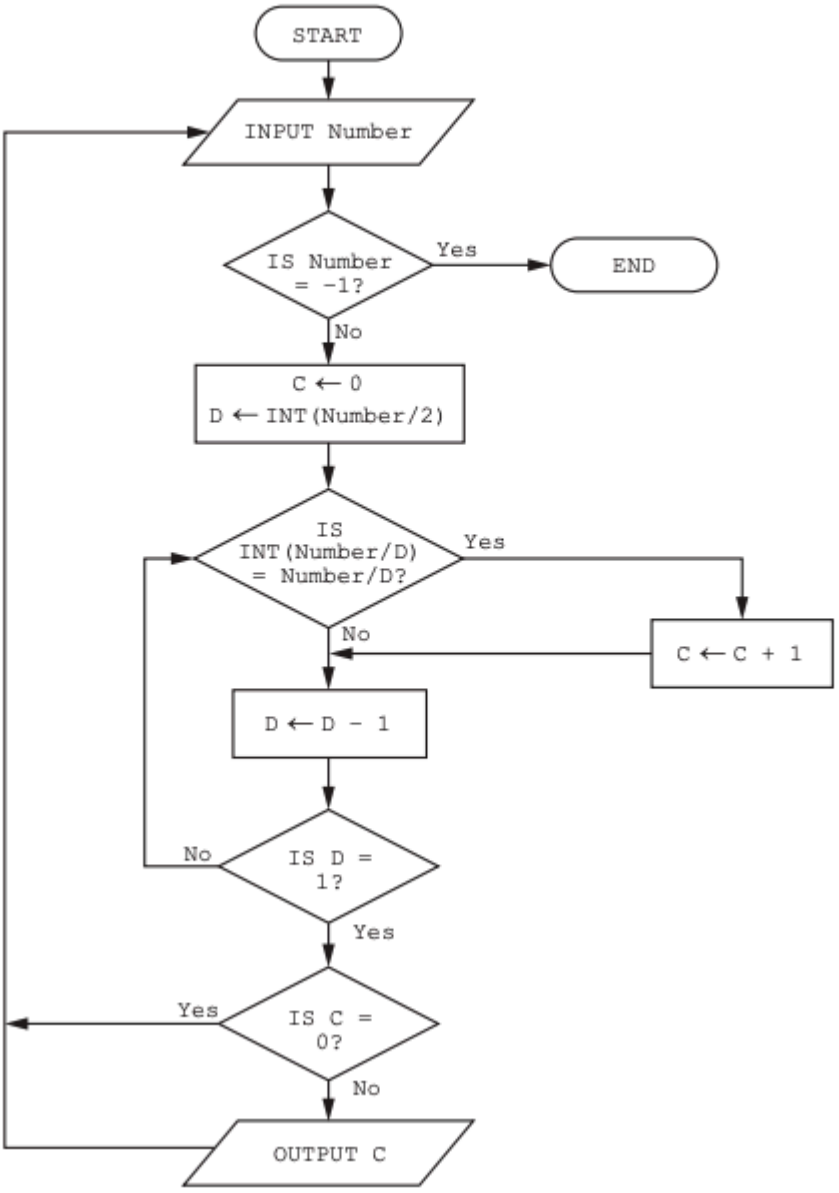
Password	PasswordRepeat	OUTPUT
		(Please enter password)
Secret		Reject
		(Please enter password)
Secret		Reject
		(Please enter password)
VerySecret	VerySecret	Accept
		(Please enter password)
Pa55word	Pa55word	Accept
		(Please enter password)
999		Reject

NOTE: REMEMBER to add the output line “Please enter password”. Look out for these in all trace tables!

Question 3

This flowchart inputs a whole number. The function `INT` returns the integer value of a number. For example, `INT (7.5)` is `7`

An input of `-1` ends the routine.



Trace table:

Number	C	D	OUTPUT
7	0	3	
		2	
		1	
6	0	3	
	1	2	
	2	1	2
5	0	2	
		1	
4	0	2	
	1	1	1
-1			

Describe the purpose of this algorithm.

- Counts the number of factors a number has, apart from 1 and itself
- Outputs this number of factors

Describe the problem that occurs if a whole number smaller than 4 and not equal to –1 is input.

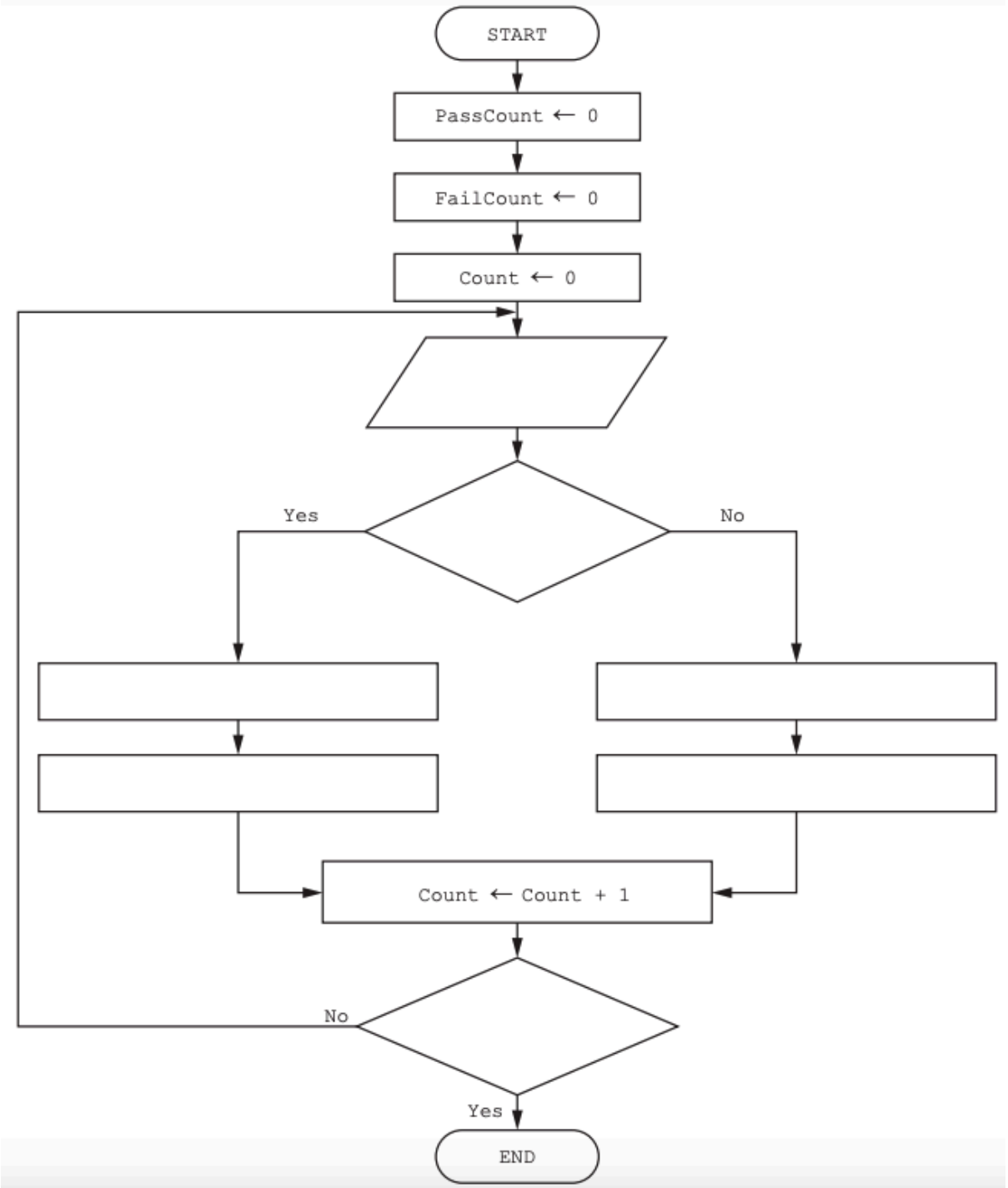
- The value of D becomes 0
- This causes an endless loop
- Due to division by zero error

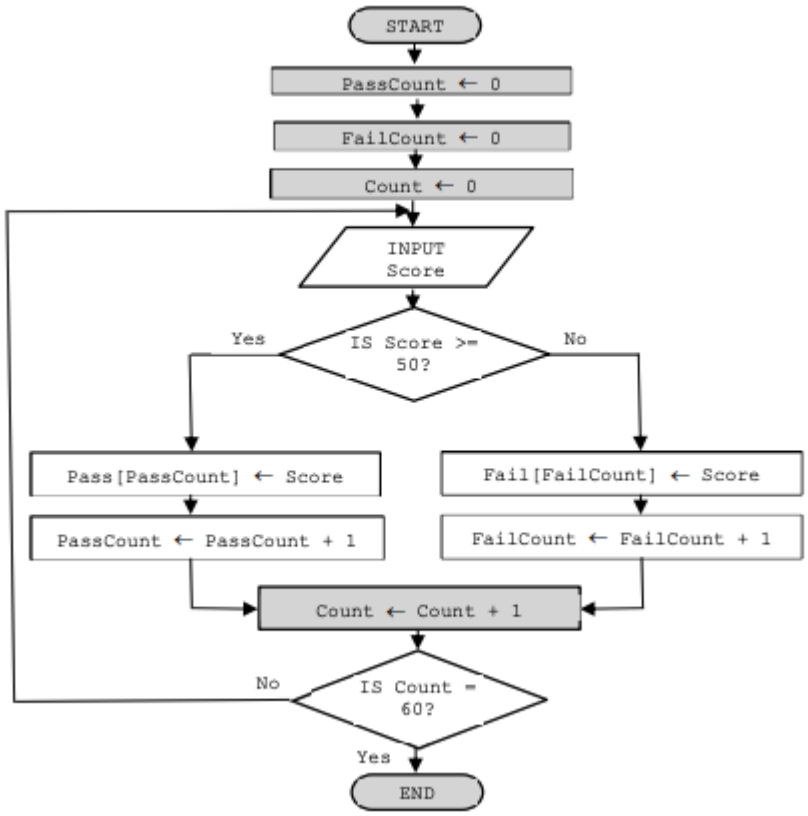
Explain how to change the flowchart to prevent this problem occurring.

- after the decision box to test if the number is -1
- insert another decision box to test if the number is less than 4 / less than or equal to 3
- return to INPUT Number if true

Question 4

The flowchart shows an algorithm that should allow 60 test results to be entered into the variable Score. Each test result is checked to see if it is 50 or more. If it is, the test result is assigned to the Pass array. Otherwise, it is assigned to the Fail array.


START
PassCount ← 0
FailCount ← 0
Count ← 0
Yes
No
Count ← Count + 1
No
Yes
END


START
PassCount ← 0
FailCount ← 0
Count ← 0
INPUT Score
IS Score >= 50?
Yes
No
Pass[PassCount] ← Score
PassCount ← PassCount + 1
Fail[FailCount] ← Score
FailCount ← FailCount + 1
Count ← Count + 1
IS Count = 60?
No
Yes
END

Question 5

The pseudocode represents an algorithm.

The pre-defined function `DIV` gives the value of the result of integer division.
For example, `Y = 9 DIV 4` gives the value `Y = 2`

The pre-defined function `MOD` gives the value of the remainder of integer division.
For example, `R = 9 MOD 4` gives the value `R = 1`

```
First ← 0
Last ← 0
INPUT Limit
FOR Counter ← 1 TO Limit
  INPUT Value
  IF Value >= 100
    THEN
      IF Value < 1000
        THEN
          First ← Value DIV 100
          Last ← Value MOD 10
            IF First = Last
              THEN
                OUTPUT Value
            ENDIF
      ENDIF
  ENDIF
NEXT Counter
```

Complete the trace table for the algorithm using this input data:

8, 66, 606, 6226, 8448, 642, 747, 77, 121

Counter	Value	First	Last	Limit	OUTPUT

Counter	Value	First	Last	Limit	OUTPUT
		0	0	8	
1	66				
2	606	6	6		606
3	6226				
4	8448				
5	642	6	2		
6	747	7	7		747
7	77				
8	121	1	1		121

NOTE: Remember to update the counter variable each time; fill up this column!

Describe the purpose of the algorithm.

- checks for / outputs 3-digit numbers
- … where the first and last digit are the same

Question 6

4 The pseudocode represents an algorithm.

The pre-defined function `DIV` gives the value of the result of integer division.
For example, `Y = 11 DIV 4` gives the value `Y = 2`

```
Count ← 0
INPUT Limit
FOR In ← 1 TO Limit
  Logic ← TRUE
  Test ← 2
  INPUT Number
  REPEAT
    IF Number / Test = Number DIV Test
      THEN
        Logic ← FALSE
      ELSE
        Test ← Test + 1
    ENDIF
  UNTIL NOT Logic OR Test >= Number DIV 2
  IF Logic
    THEN
      Store[Count] ← Number
      Count ← Count + 1
  ENDIF
NEXT In
FOR Out ← 0 TO Count - 1
  OUTPUT Store[Out]
NEXT Out
```

(a) Complete the trace table for the algorithm using this input data:
5, 9, 5, 8, 10, 7

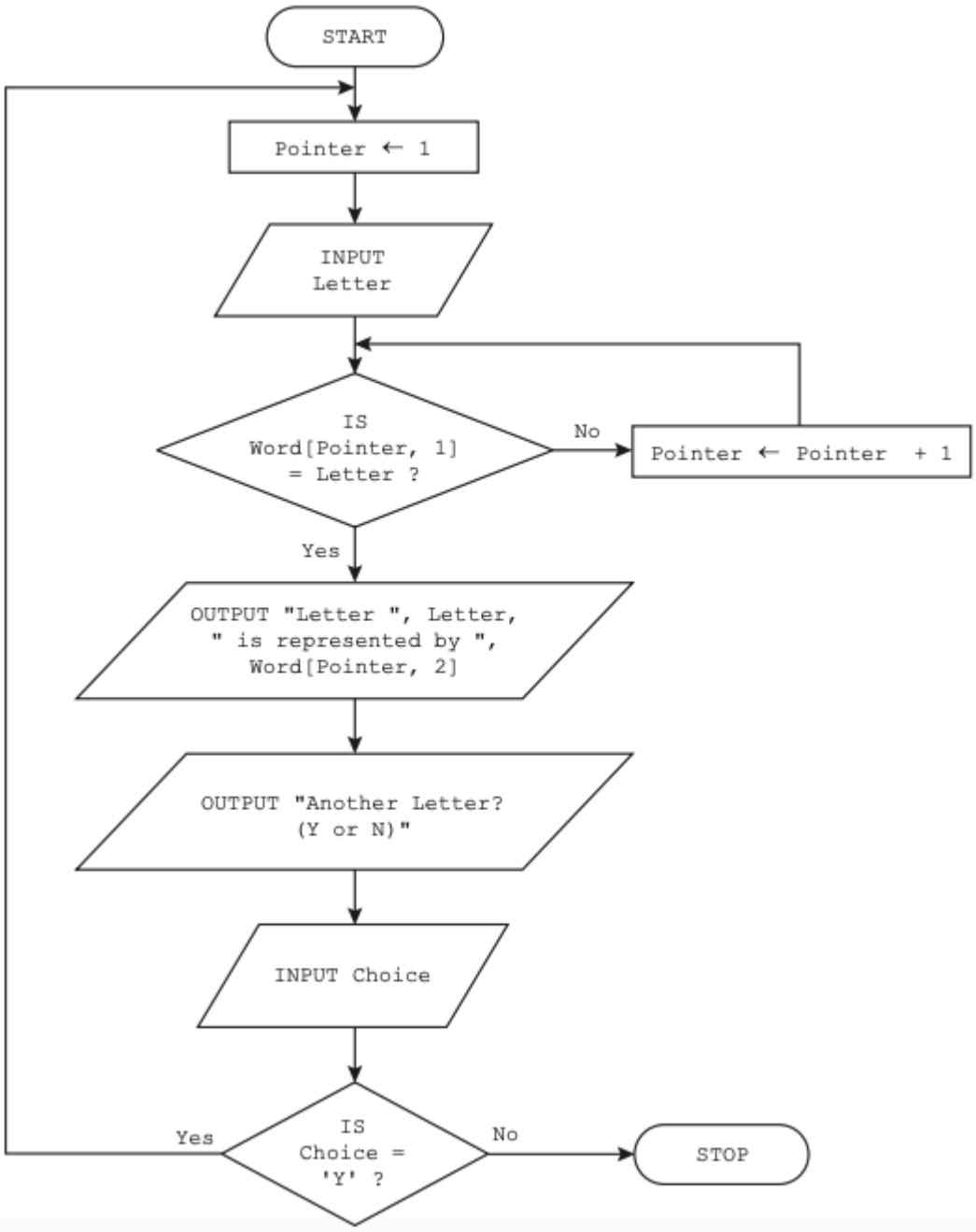
In	Logic	Test	Number	Store [Count]	Count	Limit	Out	OUTPUT

In	Logic	Test	Number	Store [Count]	Count	Limit	Out	OUTPUT
					0	5		
1	TRUE	2	9					
		3						
	FALSE							
2	TRUE	2	5					
		3		5	1			
3	TRUE	2	8					
	FALSE							
4	TRUE	2	10					
	FALSE							
5	TRUE	2	7					
		3		7	2		0	5
							1	7

State the purpose of this algorithm

- Finds and outputs the prime numbers
- Stores the prime numbers in an array

Question 7


START
Pointer ← 1
INPUT
Letter
IS
Word[Pointer, 1]
= Letter ?
No
Pointer ← Pointer + 1
Yes
OUTPUT "Letter ", Letter,
" is represented by ",
Word[Pointer, 2]
OUTPUT "Another Letter?
(Y or N)"
INPUT Choice
IS
Choice =
'Y' ?
Yes
No
STOP

The table represents the two-dimensional (2D) array `Word[]` which stores the first half of the phonetic alphabet used for radio transmission. For example, `Word[10,1]` is 'J'.

Index	1	2
1	A	Alpha
2	B	Bravo
3	C	Charlie
4	D	Delta
5	E	Echo
6	F	Foxtrot
7	G	Golf
8	H	Hotel
9	I	India
10	J	Juliet
11	K	Kilo
12	L	Lima
13	M	Mike

(a) Complete the trace table for the algorithm by using the input data: F, Y, D, N

Pointer	Letter	Choice	OUTPUT

Pointer	Letter	Choice	OUTPUT
1	F		
2			
3			
4			
5			
6			Letter F is represented by Foxtrot
			Another Letter? (Y or N)
		Y	
1	D		
2			
3			
4			Letter D is represented by Delta
			Another Letter? (Y or N)
		N	

NOTE: remember to add the statement “Another Letter? (Y or N)!

Identify the type of algorithm used.

Linear search

Describe one problem that could occur with this algorithm if an invalid character was input.

- The algorithm would not stop
- … because it would not have found the item it was seeking

Or

- The array would run out of values after the pointer reached 13
- the algorithm will crash

Practice on identifying & correcting errors

Question 1

The pseudocode algorithm should work as a calculator and output the result.

```
1   Continue ← 1
2   WHILE Continue = 0
3     OUTPUT "Enter 1 for +, 2 for -, 3 for * or 4 for /"
4     INPUT Operator
5     OUTPUT "Enter the first value"
6     INPUT Value1
7     OUTPUT "Enter the second value"
8     OUTPUT Value2
9     IF Operator
10      1: Answer ← Value1 + Value2
11      2: Answer ← Value1 - Value2
12      3: Answer ← Value1 * Value2
13      4: Answer ← Value1 / Value2
14    ENDCASE
15    OUTPUT "The answer is ", Value1
16    OUTPUT "Do you wish to enter more values (Yes or No)?"
17    INPUT MoreValues
18    IF MoreValues = "No"
19      THEN
20        Continue ← 1
21    ENDIF
22  UNTIL Continue = 0
```

a. Find the five errors in the pseudocode and suggest a correction for each error.

```
1 Continue ← 1
2 REPEAT
3    OUTPUT "Enter 1 for +, 2 for -, 3 for * or 4 for /"
4    INPUT Operator
5    OUTPUT "Enter the first value"
6    INPUT Value1
7    OUTPUT "Enter the second value"
8    INPUT Value2
9    CASE OF Operator
10     1: Answer ← Value1 + Value2
11     2: Answer ← Value1 - Value2
12     3: Answer ← Value1 * Value2
13     4: Answer ← Value1 / Value2
14   ENDCASE
15   OUTPUT "The answer is ", Answer
16   OUTPUT "Do you wish to enter more values (Yes or No)?"
17   INPUT MoreValues
18   IF MoreValues = "No"
19     THEN
20       Continue ← 0
21   ENDIF
22 UNTIL Continue = 0
```

b. The algorithm needs changing to allow only the numbers 1, 2, 3, or 4 to be entered for the input variable `Operator`. Write the pseudocode to perform this task and state where in the algorithm it would be located.

```
WHILE Operator <1 OR Operator >4 DO
    OUTPUT "Enter a valid choice: 1,2,3 or 4
    INPUT Operator
ENDWHILE
```

OR

```
REPEAT
    IF Operator < 1 OR Operator > 4
          THEN
                OUTPUT "Enter 1, 2, 3 or 4"
                INPUT Operator
          ENDIF
UNTIL Operator >= 1 AND Operator <= 4
```

Location: after line 4 / between lines 2-5

Question 2

An algorithm has been written in pseudocode to generate 50 positive random integers with values less than or equal to 100. These numbers are stored in the array `NumRand[]`

The function `RandUp(X,Y)` generates a random integer greater than `X` and less than or equal to `Y`
For example, `RandUp(1,4)` generates `2` or `3` or `4`

```
1 Count ← 0
2 WHILE Counter > 50 DO
3     NumRand[Counter] ← RandUp(1,100)
4     Counter ← Counter - 2
5 ENDWHILE
```

Find the four errors in the pseudocode and write a correction for each error.

```
Counter ← 0
WHILE Counter < 50 DO
     NumRand[Counter} ← RandUp(0, 100)
     Counter ← Counter + 1
ENDWHILE
```

Question 3

An algorithm has been written in pseudocode to generate 50 positive random integers with values less than or equal to 100. These random integers are stored in the array `RandNum[ ]`

The function `Rand(X, Y)` generates a random integer greater than or equal to `X` and less than `Y`. For example, `Rand(1, 4)` generates `1` or `2` or `3`.

```
1 Count ← 0
2 REPEAT
3     RandNum[Counter] ← Rand(1, 100)
4     Count ← Count + 2
5 UNTIL Count <= 50
```

Find the four errors in the pseudocode and write a correction for each error.

```
Counter ← 0
REPEAT
     RandNum[Counter] ← Rand(1, 100)
     Counter ← Counter + 1
UNTIL Counter >= 50
```

Question 4

This algorithm checks passwords.

- Each password must be 8 or more characters in length; the predefined function `Length` returns the number of characters.
- Each password is entered twice, and the two entries must match.
- Either `Accept` or `Reject` is output.
- An input of 999 stops the process.

```
REPEAT
  OUTPUT "Please enter password"
  INPUT Password
  IF Length(Password) >= 8
    THEN
      INPUT PasswordRepeat
      IF Password <> PasswordRepeat
        THEN
          OUTPUT "Reject"
        ELSE
          OUTPUT "Accept"
      ENDIF
    ELSE
      OUTPUT "Reject"
  ENDIF
UNTIL Password = 999
```

Explain how algorithm could be extended to allow 3 attempts at inputting the matching password. Any pseudocode statements used must be fully explained. [4 marks]

- Position: before INPUT PasswordRepeat // at start
  - use a variable counter for number of tries or flag
  - initialise variable counter or flag
- Position after IF Length(Password) >= 8 THEN or after INPUT PasswordRepeat

    - insert REPEAT/WHILE/(conditional) loop
- Position after OUTPUT "Reject"
    - add one to counter (for number of tries)
    - output a message "Try again"
    - add INPUT PasswordRepeat
- Position after OUTPUT "Accept"
    - reset flag to show password matched
- Position after ENDIF
    - (insert UNTIL/ENDWHILE)to exit the loop after three tries or if the repeated password matches the original

Question 5

The pseudocode algorithm should allow a user to input the number of scores to be entered and then enter the scores. The scores are totalled, the total is output and the option to enter another set of scores is offered.

```
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
```

Identify the four errors in the pseudocode and suggest a correction for each error.

Line 3 – should be `FullScore ← 0`
Line 7 – should be `FullScore ← FullScore + Score`
Line 8 – should be `NEXT Allow ENDFOR //` alternatively Line 5 could be `REPEAT` with `StoreLoop ← 0` just above it and `StoreLoop ← StoreLoop + 1` between lines 7 and 8.
Line 11 – should be `INPUT Another`

Correct Algorithm 1

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

Correct Algorithm 2

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

Question 6

An algorithm has been written in pseudocode to:

- input 25 positive whole numbers less than 100
- find and output the largest number
- find and output the average of all the numbers

```
01  A ← 0
02  B ← 0
03  C ← 0
04  REPEAT
05    REPEAT
06      INPUT D
07    UNTIL D > 0 AND D < 100 AND D = INT(D)
08    IF D > B
09      THEN
10        B ← D
11    ENDIF
12    C ← C + D
13    A ← A + 1
14  UNTIL A >= 25
15  E ← C / A
16  OUTPUT "Largest number is ", B
17  OUTPUT "Average is ", E
```

The algorithm needs to be changed to include finding and outputting the smallest number input. Describe how you would change the algorithm.

- new variable for minimum...
- ... set to first value/high value
- ... at start of program / before line 4
- test input / D for less than minimum
- ... replace value minimum if input less than
- ... after line 7 and before line 14
- new output for minimum (with appropriate message)
- ... at end of program // after line 14

Question 7

An algorithm allows a user to input their password and checks that there are at least eight characters in the password. Then, the user is asked to re-input the password to check that both inputs are the same. The user is allowed three attempts at inputting a password of the correct length and a matching pair of passwords. The pre-defined function `LEN(X)` returns the number of characters in the string, `X`

```
01  Attempt ← 0
02  REPEAT
03    PassCheck ← TRUE
04    OUTPUT "Please enter your password "
05    INPUT Password
06    IF LEN(Password) < 8
07      THEN
08        PassCheck ← TRUE
09      ELSE
10        OUTPUT "Please re-enter your password "
11        INPUT Password2
12        IF Password <> Password
13           THEN
14           PassCheck ← FALSE
15        ENDIF
16    ENDIF
17    Attempt ← Attempt + 1
18  UNTIL PassCheck OR Attempt <> 3
19  IF PassCheck
20    THEN
21     OUTPUT "Password success"
22    ELSE
23      OUTPUT "Password fail"
24  ENDIF
```

Identify the three errors in the pseudocode and suggest a correction to remove each error.

One mark per mark point, max **three**

- line 8 / `PassCheck ← TRUE`

correction `PassCheck ← FALSE`

- line 12 / `IF Password <> Password`

correction `IF Password2 <> Password` // `IF Password <> Password2`

- line 18 / `UNTIL PassCheck OR Attempt <> 3`

correction `UNTIL PassCheck OR Attempt = 3 / UNTIL PassCheck OR Attempt >= 3`

Question 8

An algorithm has been written in pseudocode to allow 100 positive numbers to be input. The total and the average of the numbers are output.

```
01 Counter ← 100
02 Total ← 0
03 WHILE Counter > 100 DO
04    INPUT Number
05    IF Number > 0
06       THEN
07         Total ← Total + Counter
08         Counter ← Counter + 1
09    ENDCASE
10 ENDWHILE
11 OUTPUT "The total value of your numbers is ", Total
12 OUTPUT "The average value of your numbers is ", Total / 100
```

Identify the four errors in the pseudocode and suggest corrections

One mark per mark point, max **four**

- Line 01 / `Counter ← 100`
  should be `Counter ← 0`

- Line 03 / `While Counter > 100 DO`
  should be `While Counter < 100 DO`

- Line 07 / `Total ← Total + Counter`
  should be `Total ← Total + Number`

- Line 09 / `ENDCASE`
  should be `ENDIF`

Describe the changes you should make to the corrected algorithm so that a count-controlled loop is used to allow 100 positive numbers to be input. You do not need to rewrite the algorithm.

- replace line 03
- with FOR
- … with limits 0 to 99 / 1 to 100
- replace line 05 to check if Number is not positive
- … (if Number is not positive) insert a validation and re-input routine between lines 06 and 07 …
- … that will repeat until a positive value is entered
- remove the counter update / line 08
- replace line 10 / ENDWHILE with NEXT

Question 9

An algorithm has been written in pseudocode to calculate a check digit for a four-digit number.
The algorithm then outputs the five-digit number including the check digit.
The algorithm stops when –1 is input as the fourth digit.

```
01 Flag ← FALSE
02 REPEAT
03     Total ← 0
04     FOR Counter ← 1 TO 4
05       OUTPUT "Enter a digit ", Counter
06       INPUT Number[Counter]
07       Total ← Total + Number * Counter
08       IF Number[Counter] = 0
09         THEN
10           Flag ← TRUE
11       ENDIF
12     NEXT Counter
13     IF NOT Flag
14       THEN
15         Number[5] ← MOD(Total, 10)
16         FOR Counter ← 0 TO 5
17             OUTPUT Number[Counter]
18         NEXT
19     ENDIF
20 UNTIL Flag
```

Identify the three errors in the pseudocode and suggest a correction for each error.

One mark for each error identified and correction

- Line `07 Total ← Total + Number * Counter`
  should be `Total ← Total + Number[Counter] * Counter`
- Line `08 IF Number[Counter] = 0`
  should be `IF Number[Counter] = -1` // should be `IF Number[Counter] < 0`
- Line `16 FOR Counter ← 0 TO 5`
  should be `FOR Counter ← 1 TO 5`

The algorithm does not check that each input is a single digit. Identify the place in the algorithm where this check should occur. Write pseudocode for this check. Your pseudocode must make sure that the input is a single digit and checks for –1

One mark for place in algorithm (**max one**)

- around lines `05` and `06`
- line `07`
- (immediately) after the input of the number

Three marks pseudocode
One mark for each point (**max three**)

- Use of `REPEAT … UNTIL` // any working loop structure
- check for `>0` // `>=0`
- check for `<10` // `>9`
- check for whole number
- check for -1
- check for length of digit <> 1

Example

```
REPEAT
    OUTPUT "Enter a digit "
    INPUT Number[Counter]
UNTIL Number[Counter] = Round(Number[Counter],0) AND ((Number[Counter] = -1) OR
     (Number[Counter] > 0 AND Number[Counter] < 10))
```

Question 10

An algorithm has been written in pseudocode to input some numbers. It only outputs any numbers that are greater than or equal to `100`. The number `999` is not output and stops the algorithm.

```
INPUT Number
WHILE Numbers <> 999 DO
    IF Number > 100
        THEN
            OUTPUT Number
    ENDIF
ENDWHILE
OUTPUT Number
```

Identify the four errors in the pseudocode and suggest corrections.

One mark for each error identified and correction:

- `Numbers` **should be** `Number`
- `IF Number > 100` **should be** `IF Number >= 100`
- `INPUT Number` **is missing from inside the loop, insert** `INPUT Number` **after the** `ENDIF` **statement.**
- **The final** `OUTPUT Number` **is not needed, remove it.**

Question 11

This section of program code asks for 80 numbers between 100 and 1000 inclusive to be entered. The pseudocode checks that the numbers are in the correct range and then stores the valid numbers in an array. It counts how many of the numbers are larger than 500 and then outputs the result when finished.

```
01 Count ← 0
02 FOR Index ← 1 TO 80
03     INPUT "Enter a number between 100 and 1000 ", Number
04         WHILE Number <= 99 AND Number >= 1001
05         INPUT "This is incorrect, please try again", Number
06     ENDWHILE
07     Num[80] ← Number
08     IF Number > 500 THEN Count ← Count + 1 ENDIF
09 UNTIL Index = 80
10 PRINT Index
11 PRINT "numbers were larger than 500"
```

There are four lines of code that contain errors. State the line number for each error and write the correct code for that line.

One mark for each error identified plus suggested correction (the corrected lines must be written in full).

Correct lines:

Line 4 `WHILE Number <= 99 OR Number >= 1001`
Line 7 `Num[Index] ← Number`
Line 9 `NEXT Index`
Line 10 `PRINT Count`

Question 12

```
01 DECLARE A[1:10] : STRING
02 DECLARE T : STRING
03 DECLARE C, L : INTEGER
04 L ← 10
05 FOR C ← 1 TO L
06     OUTPUT "Please enter name "
07     INPUT A[C]
08 NEXT C
09 FOR C ← 1 TO L
10     FOR L ← 1 TO 9
11         IF A[L] > A[L + 1]
12           THEN
13             T ← A[L]
14             A[L] ← A[L + 1]
15             A[L + 1] ← T
16         ENDIF
17     NEXT L
18 NEXT C
19 FOR C ← 1 TO L
20     OUTPUT "Name ", C, " is ", A[C]
21 NEXT C
```

State the purpose of this algorithm

Displaying/sort 10 names in alphabetical order.

State four processes in this algorithm.

- Initialisation
- Inputting 10 names
- Storing the names in an array
- Sorting the names in alphabetical order using a bubble sort
- Displaying the 10 names
- Iteration

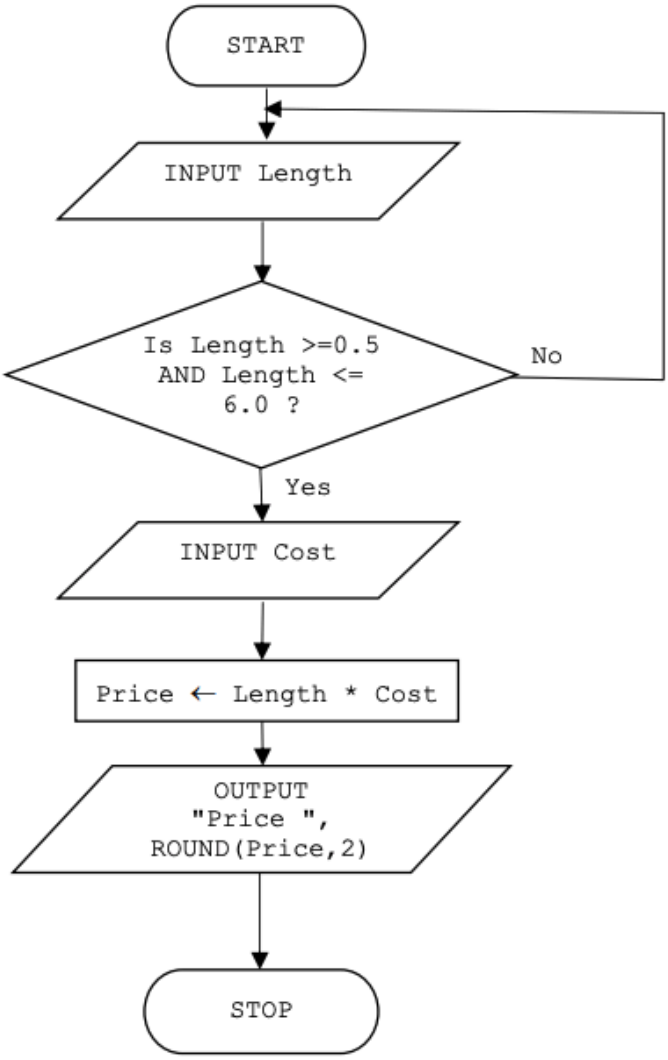
Question 13

A programmer is designing an algorithm to calculate the cost of a length of rope.
The program requirements are:

- input two values: the length of rope in metres `Length` and the cost of one metre `Cost`
- perform a validation check on the length to ensure that the value is between `0.5` and `6.0` inclusive
- calculate the price `Price`
- output the price rounded to two decimal places.

Use the variable names given.

Complete a flowchart for this algorithm.



- correct use of flowchart symbols
- working flow lines and complete
- both inputs correct
- working range check
- working calculation
- correct output rounded to two decimal places

Give two different sets of test data for this algorithm and state the purpose of each set.

- 1 and 1
- normal data to ensure the algorithm accepts this test data

- –1 and 1
- abnormal data for length to ensure that it is rejected

Complete the headings for the trace table to show a dry-run for this algorithm.

Length	Cost	Price	OUTPUT

Describe an improvement that should be made to the requirements for this algorithm.

- Validate `Cost` ...
- ... with a range/presence check
- add another validation check for Length

Question 14

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. The contents of the array are then output.

```
01 DECLARE City ARRAY[1:50, 1:2] OF BOOLEAN
02 DECLARE Count : INTEGER
03 DECLARE Out : INTEGER
04 Count ← 1
05 IF
06     OUTPUT "Enter the name of the city"
07     INPUT City[Count, 2]
08     OUTPUT "Enter the name of the country"
09     INPUT City[Count, 2]
10     Count ← Count + 1
11 UNTIL Count = 50
12 FOR Out ← 1 TO 1
13     OUTPUT "The city ", City[Out, 1], " is in ", City[Out, 2]
14 NEXT Out
```

(a) Identify the **four** errors in the pseudocode and suggest corrections.

MP1 Line 01 / `DECLARE City ARRAY[1:50, 1:2] OF BOOLEAN` should be `DECLARE City : ARRAY[1:50, 1:2] OF STRING` Line 05 / `IF` should be `REPEAT`

MP2 Line 07 / `INPUT City[Count, 2]` should be `INPUT City[Count, 1]`

MP3 Line 11 / `UNTIL Count = 50` // Line 04 / `Count ← 1` **AND** Line 10 / `Count ← Count + 1` should be `UNTIL Count = 51` / `UNTIL Count > 50` // Line 04 / `Count ← 0` **AND** move Line 10 to beginning of loop / Line 06

MP4 Line 12 / `FOR Out ← 1 TO 1` should be `FOR Out ← 1 TO 50`

Correct algorithm:

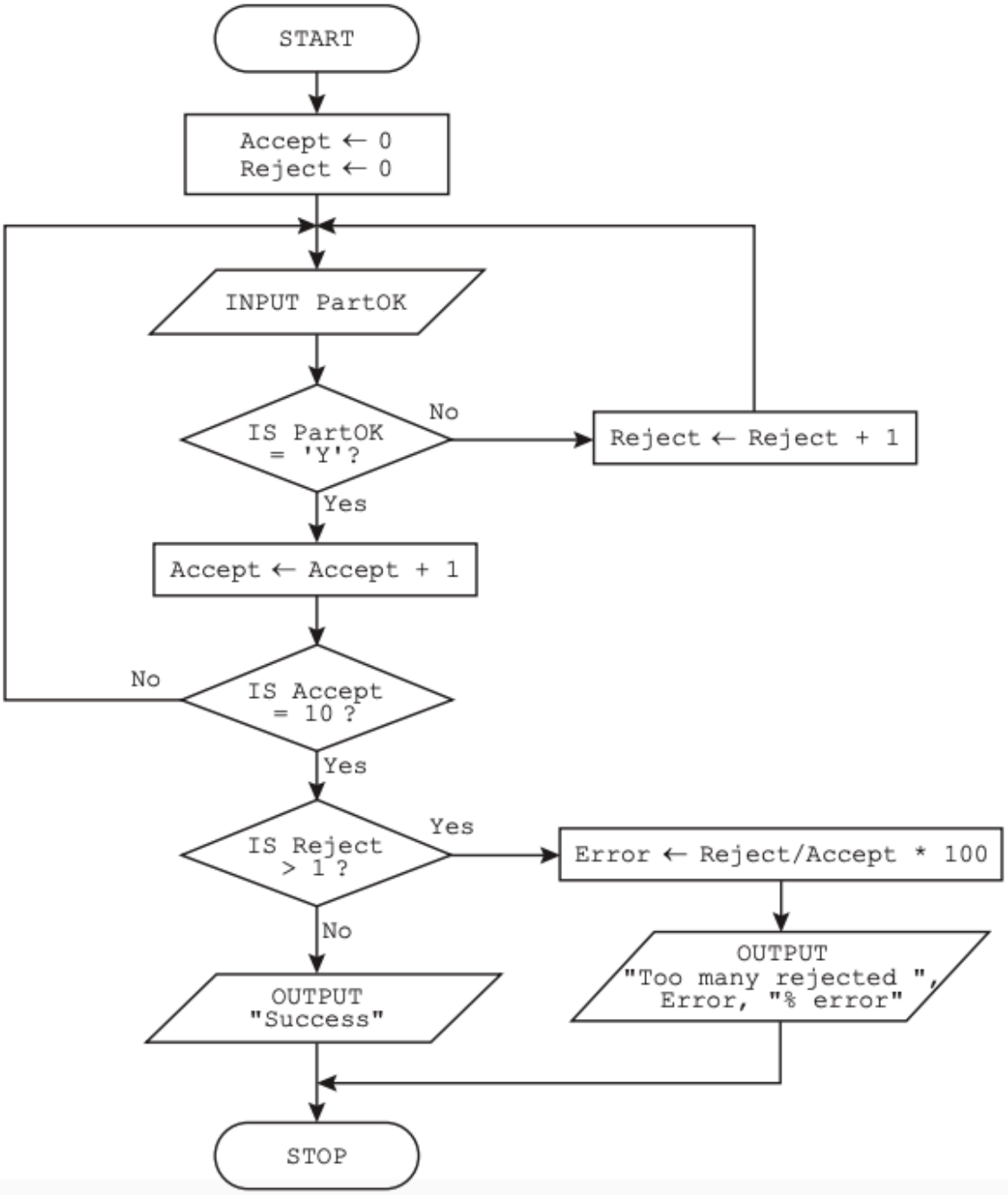
```
01 DECLARE City : ARRAY[1:50, 1:2] OF STRING
02 DECLARE Count : INTEGER
03 DECLARE Out : INTEGER
04 Count ← 1
05 REPEAT
06     OUTPUT "Enter the name of the city"
07     INPUT City[Count, 1]
08     OUTPUT "Enter the name of the country"
09     INPUT City[Count, 2]
10     Count ← Count + 1
11 UNTIL Count > 50
12 FOR Out ← 1 TO 50
13     OUTPUT "The city ", City[Out, 1], " is in ",
               City[Out, 2]
14 NEXT Out
```

Describe the changes you should make to the corrected algorithm to allow the name of a country to be input and to display only the stored cities from that country.

- MP1 add an input (and prompt to ask) for the country to be searched
- MP2 …between lines 11 and 12
- MP3 …using a new variable for the input
- MP4 Add an IF statement to check if the current `Country` array element matches the country being searched
- MP5 …between lines 12 and 13
- MP6 …if it does, allow the output in line 13 // the output in line 13 should be after a THEN
- MP7 If it does not, check the next element.

Question 15

This is an algorithm to find if a batch of parts has been manufactured successfully.



Describe how the algorithm should be changed to accept 'Y' or 'y' for a successfully manufactured part.

- after the Input box // before the first decision box
- insert a process box
- to convert the input to upper case

OR

- change the first decision / add another decision box
- to accept 'y' as well
- by adding OR PartOK = 'y'

Validation & verification questions

Question 1

Give one piece of normal test data and one piece of erroneous test data that could be used to validate the input of an email address. State reason for your choice in each case.

- Normal test data: computerscience@cambridge.org.uk
- Reason: this is a valid email address (containing the @ symbol) and should be accepted

- Erroneous test data: computerscienceisgreat
- Reason: this is just a string, and should be rejected (an email address needs a '@')

Question 2

Tick one or more boxes in each row to match the type(s) of test data to each description.

Description	Types of test data			
	Boundary	Erroneous / Abnormal	Extreme	Normal
test data that is always on the limit of acceptability				
test data that is either on the limit of acceptability or test data that is just outside the limit of acceptability				
test data that will always be rejected				
test data that is within the limits of acceptability				

Description	Types of test data			
	Boundary	Erroneous / Abnormal	Extreme	Normal
test data that is always on the limit of acceptability			✔	
test data that is either on the limit of acceptability or test data that is just outside the limit of acceptability	✔			
test data that will always be rejected		✔		
test data that is within the limits of acceptability			✔	✔

Question 3

A PIN (personal identification number) is input into a banking app by the user. Before the PIN is accepted, the following validation checks are performed:

- **check 1 – each character must be a digit**
- **check 2 – there must be exactly four digits**
- **check 3 – the value of the PIN must be between 1000 and 9999 inclusive.**

Describe each validation check.

Check 1

- uses a type check
- to to ensure that the value is a number / integer

Check 2

- uses a length check
- to ensure that there are only 4 characters / digits

Check 3

- use a range check
- to ensure that the value is >= 1000 and <=9999

The PIN can be changed by the user. Describe how the new PIN could be verified before use.

- input the new PIN
- input the new PIN again
- check that both PINs are the same
- check that the new PIN is not the same as the old PIN

Question 4

Program needs to make sure value input for a measurement meets the following rules:

- **the value is a positive number**
- **a value is always input**
- **the value is less than 1000.**

Describe the validation checks that the programmer would need to use.

- range check with acceptable values is (greater than) zero and less than 1000
- presence check to ensure the program will not continue until a value has been entered
- type/character check to ensure that a number is entered
- length check to ensure there are no more than 3 digits entered

Question 5

Format check is used to make sure that any date entered is in the dd/mm/yyyy style.)
Give one example of normal test data and one example of abnormal test data you

should use to make sure the format check is working properly. State a reason for each of your choices of test data.

- Normal – 30/12/1960 ...
- Reason – the date is written in the correct format and) should be accepted.

- Abnormal – 30/Dec/1960 ...
- Reason – the month is not written in the correct format and) should be rejected.

Describe how a length check could be used with the date entered.

- check that there are 10 characters in total
- if the date is too long/short it will be rejected