Algorithm Design and Problem Solving

The program development life cycle, structure diagrams, flowcharts and pseudocode, the standard algorithms (counting, totalling, bubble sort, linear search), validation, verification, test data and trace tables.

Program development life cycle

Analysis

The first stage is used to understand and define the problem and requirements. There are two tasks: abstraction and decomposition.

Abstraction means simplifying the problem, removing unnecessary details from the problem and filtering out irrelevant characteristics from those elements.

Decomposition means breaking down a complex problem into smaller parts, which can then be subdivided into even smaller parts, that can be solved easily. It is divided into 4 parts:

- **Inputs**, which is the data used by the system that needs to be entered while the system is active.
- **Processes**, which are the tasks that need to be performed using the input data and any other previously stored data.
- **Outputs**, which is the information that needs to be displayed or printed for the users of the system.
- **Storage**, which is the data that needs to be stored in files on an appropriate medium for use in the future.

Design

The design stage comprises planning the solution and structure of the software. Flowcharts, pseudocode and structure diagrams are used to visualize what the final outcome should be.

Coding

This stage is where the actual code is written and the program is developed.

Testing

The last stage is when the program is tested to find errors and to check if it is working in the desired way. The program is tested using:

- **Normal** data, which is within range and should be accepted.
- **Extreme** data, which is at the extremes of the given range but is still acceptable.
- **Boundary** data, which is either on the limit of acceptability or test data that is just outside the limits of acceptability.

Note Academy

- **Erroneous / abnormal** data, which is out of range and should be rejected.

Computer systems and sub-systems

A **computer system** is a complete working system that can perform tasks based on instructions provided.

Sub-systems are distinct functional parts of a computer system that work together to perform specific tasks. Examples include:

- the CPU, which includes the ALU and CU.
- the memory sub-system, which includes the RAM and ROM.
- the input-output sub-system, which includes all the input and output devices.

These sub-systems comprise further sub-systems, such as the cache inside the CPU.

Ways of designing a solution

Structure Diagrams

Structure diagrams are visual representations of the hierarchical structure of a system, showing the relationships and dependencies between components. Their purpose is to break down the system into smaller, manageable modules and illustrate how they interact.

```
Islamic App
├── Prayer Times
│   ├── Prayer times throughout the year
│   └── Time zone from user
└── Quran Reader
    ├── PDF viewer
    └── Bookmarks
```

In the example above, we get an idea of how developers of an Islamic app can use structure diagrams to break down the app into smaller modules.

Flowcharts

Flowcharts are diagrammatic representations of a process or algorithm, using various symbols to denote different types of actions or steps. They visually map out the steps and decision points in a process, making it easier to understand and follow. There are 5 types of blocks used:

1. **Start and end:** These oval symbols represent the start and the end of an algorithm.
2. **Processes / actions:** These rectangle symbols represent the processes in an algorithm, such as `A ← 0`. A rectangle with extra vertical lines at its sides is used whenever a function or procedure is used, such

Note Academy

as `X ← GETPOINTS(CARD)`.

3. **Input and output:** A parallelogram is used to show the input and output of information, such as `INPUT X` and `OUTPUT "NOTE ACADEMY"`.
4. **Decision:** A diamond represents a decision (Yes/No or True/False) that results in two lines representing the different possible outcomes. It is also used when iteration takes place, for example `IS X < 0?`.

Example. Here is an algorithm that finds the larger of two numbers, written first as statements and then as a flowchart.

1. Start
2. Input the first number, A
3. Input the second number, B
4. Compare A and B
5. If A is greater than B, then A is the maximum
6. Else, B is the maximum
7. Output the maximum number
8. End

Pseudocode

Pseudocode is a method used to describe an algorithm using a combination of natural language and programming language constructs.

Assignment and variables

Variables can be assigned values so that they can be used later. This is how to do it:

```
Price ← 80
```

The arrow is used to indicate that the value 80 is being assigned to `Price`.

```
DiscountedPrice ← Price * 0.9
```

This assignment will assign the value 72 to `DiscountedPrice`. The `*` is the sign used for multiplication. The other signs are:

- `+` for addition.
- `-` for subtraction.
- `/` for division.
- `^` for raised to the power.

Note Academy

- Round brackets for prioritising and grouping.

INPUT and OUTPUT

```
INPUT Name
OUTPUT 'Note Academy'
OUTPUT 'My name is ', Name
```

The pseudocode above shows how to input and output things.

- The first line will take input and store it in a variable called `Name`.
- The second line will output the words 'Note Academy'.
- The last line will combine the phrase 'My name is ' and what is stored in `Name`, because a comma combines two things.

IF ELSE statement

```
IF Marks > 49
   THEN
      OUTPUT 'Pass'
   ELSE
      OUTPUT 'Fail'
ENDIF
```

This pseudocode shows a simple **conditional statement**. This means that if `Marks` is greater than 49, it will output 'Pass'. Otherwise, it will output 'Fail'.

We can also have multiple conditions in the same statement. This is how to do it:

```
IF Marks > 49 OR Grade = 'A'
```

We can also have a conditional statement within a conditional statement. This is called a **nested IF** statement.

When iteration or selection statements are nested inside another statement, it is called a **nested statement**. This allows for more complex logic and execution flows. It helps in organizing the code logically and hierarchically, making it easier to understand and maintain.

Note Academy

```
IF Marks > 49
   THEN
      IF Marks > 74
         THEN
            OUTPUT 'Grade = A'
         ELSE
            OUTPUT 'Grade = B'
      ENDIF
   ELSE
      OUTPUT 'Grade = F'
ENDIF
```

CASE OF statement

Instead of combining multiple IF ELSE statements, the CASE OF statement is used.

```
CASE OF Marks
   < 50: OUTPUT 'Grade = F'
   < 60: OUTPUT 'Grade = D'
   < 70: OUTPUT 'Grade = C'
   < 80: OUTPUT 'Grade = B'
   OTHERWISE OUTPUT 'Grade = A'
ENDCASE
```

FOR loop

This is a **count-controlled** loop, which means that it will run a set number of times. The loop given below will output '-' 10 times.

```
FOR x ← 1 TO 10
   OUTPUT '-'
NEXT x
```

REPEAT UNTIL loop

This is a **post-condition** loop which means that the actions in this loop take place at least once. It keeps running until the condition is met.

Note Academy

```
Count ← 0
Total ← 0
Num ← 0
REPEAT
   Count ← Count + 1
   OUTPUT "Input number to add to total"
   INPUT Num
   Total ← Total + Num
UNTIL Count = 10
OUTPUT Total
```

The loop above will run 10 times. Each time, it will take a number and add it to the total. When the loop is complete, it will output the total once.

WHILE loop

This is a **pre-condition** loop, which means that it will check if the condition is true before the actions in the loop take place.

```
INPUT Age
WHILE Age < 17 OR Age > 100
   OUTPUT 'Enter a valid age'
   INPUT Age
ENDWHILE
```

This loop will ensure that the user enters an age between 17 and 100. If the user enters the correct age the first time, the loop will not run. This loop will keep running until the user enters an age between 17 and 100.

Standard methods of solution

Counting

To count how many times something occurs, a value (in this case, `count`) is initialised to 0 and then is incremented every time it needs to be.

```
count ← 0
FOR i ← 1 TO LENGTH(list)
   IF list[i] = 'Note Academy'
      THEN
         count ← count + 1
   ENDIF
NEXT i
```

Note Academy

Finding maximum, minimum and average values

To find the largest value, the smallest value and the average of the elements in a list, use this pseudocode.

```
max ← list[1]
min ← list[1]
total ← 0
FOR i ← 1 TO LENGTH(list)
   IF list[i] > max
      THEN max ← list[i]
   ENDIF
   IF list[i] < min
      THEN min ← list[i]
   ENDIF
   total ← total + list[i]
NEXT i
average ← total / LENGTH(list)
OUTPUT max, min, average
```

Every item in the list is compared with `max`. If the value is greater, the `max` variable is updated. Otherwise, it is rejected. The same is done for `min` with the smaller value.

Totalling

To calculate the sum of all elements in a list, use the pseudocode below.

```
total ← 0
FOR i ← 1 TO LENGTH(list)
   total ← total + list[i]
NEXT i
OUTPUT total
```

Bubble sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Note Academy

```
Boundary ← 9
REPEAT
   Swap ← FALSE
   FOR i ← 1 TO Boundary
      IF Age[i] > Age[i + 1]
         THEN
            Temp ← Age[i]
            Age[i] ← Age[i + 1]
            Age[i + 1] ← Temp
            Swap ← TRUE
      ENDIF
   NEXT i
   Boundary ← Boundary - 1
UNTIL NOT Swap OR Boundary = 1
```

- During each pass, the algorithm compares each pair of adjacent elements. If the first element of the pair is greater than the second, they are swapped.
- This sorts the array into **ascending** order. To sort it in descending order, replace the `>` sign with `<`.
- The loop keeps running until no swaps are needed, indicating that the list is sorted.
- This example sorts an array of 10 items, so the first boundary is 9 (the last pair compared is items 9 and 10).

Linear search

A linear search is used to check if a value can be found in a list or an array (discussed in Chapter 8). It checks each index one by one. The pseudocode is as follows.

Note Academy

```
OUTPUT "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
   ELSE
      OUTPUT Name, " was not in the list."
ENDIF
```

- `Found` is a Boolean value that is updated when the item is found.
- The loop will keep running until the item is found or all the items in the array are checked.

Validation and verification

Validation checks are crucial in ensuring that the data entered into a system is correct, sensible, and useful. They help to minimize errors and improve data quality.

Range check

It ensures that the input data falls within a specified range. For example, if the user enters a month, it must be between 1 and 12.

```
IF score < 0 OR score > 100
   THEN
      OUTPUT "Error: Score must be between 0 and 100"
ENDIF
```

Length check

It ensures that the input data has the correct number of characters. For example, if the user enters a bank account number, it must be 19 characters.

Note Academy

```
IF LENGTH(username) < 5 OR LENGTH(username) > 15
   THEN
      OUTPUT "Username must be between 5 and 15 characters"
ENDIF
```

Type check

It ensures that the input data is of the correct data type. For example, if the user enters their age, it must be in integers and not letters.

```
REPEAT
   INPUT NumParticipants
   IF NumParticipants <> DIV(NumParticipants, 1)
      THEN
         OUTPUT "This must be a whole number, re-enter"
   ENDIF
UNTIL NumParticipants = DIV(NumParticipants, 1)
```

Presence check

This check makes sure that a required field is not left blank, for example the first name field should not be left empty.

```
REPEAT
   INPUT EmailAddress
   IF EmailAddress = ""
      THEN
         OUTPUT "*=Required"
   ENDIF
UNTIL EmailAddress <> ""
```

Format check

This check ensures that the input data matches a specific pattern or format. For example, an email must have an `@` in it.

Check digit has already been discussed in Chapter 2.

Verification checks are crucial in ensuring that the data entered into a system is accurate and correctly represents the intended information. They help to detect and correct errors that may have been introduced during data entry.

Visual check

Note Academy

A visual check manually compares the entered data against the original source to ensure accuracy. For example, comparing a printed invoice to the data entered in the system.

Double entry check

This check ensures data accuracy by entering the data twice and comparing the two entries, for example entering a new password twice to ensure they match.

Trace tables and finding errors

Trace Tables

Trace tables are a method used to manually track the execution of an algorithm step by step, recording the state of variables at each step. This process is called a **dry run**. They are particularly useful for debugging and understanding how an algorithm processes data.

The flowchart below is a bubble sort of the five values in `X`. Its trace table is shown underneath, with the first pass filled in and the second pass left blank to complete.

The trace table shows a dry run of the flowchart. It shows the values of the variables every time they change. It also shows every time that the program outputs a value. Trace tables can also be made when an algorithm is presented as pseudocode.

Tips on Solving and Amending Algorithms

1. Read the question carefully and **understand** what you are required to do.
2. If you have to make an algorithm from scratch, **break down the problem** into sub-problems.
3. Try to **make a list of all the things** that you need to solve the problem. For example, you should check if you will need to add a counter in the algorithm to count the value of something. Similarly, if you need to perform some actions a set number of times, you should know that you have to use a FOR loop.
4. The next step will be to **combine them** all and make sure they work seamlessly and solve the problem at hand.
5. Once you have written the pseudocode, program code or made the flowchart, **test the data** to see if you get the result that is required. If time permits, try to do a dry run.
6. If you need to amend an algorithm, the first step is to understand what the algorithm **should do** versus what the algorithm that you have to amend **actually does**. Then you have to identify the places where the algorithm is taking the wrong action. Finally, amend the algorithm and test it to see if it works as it is required to.

Note Academy