Standard methods of solution, Trace Table

UNIT 7 : Algorithm, Design & Problem Solving

Topic Objectives:

- Explain the purpose of a given algorithm
- Understand standard methods of solution
- Complete a trace table to document a dry-run of an algorithm
- Identify errors in given algorithms and suggest ways of correcting these errors
- Write and amend algorithms for given problems or scenarios, using: pseudocode, program code and flowcharts

STANDARD METHODS OF SOLUTIONS

TOTALLING

Totalling is adding together a set of values.

To write a program to total you need to:

- Initialise the total to 0.
- Add the values together

(either individually or within a loop).

```
Total 0 //Intialiase with 0 outside the loop
//Inside the loop calculate total
Total Total + Value //Totalling statement
//Outside the loop, display the total
OUTPUT "Total =", Total
```

To find total of 50 numbers stored in a variable

```
Total   0 // Intialization
FOR Count   1 TO 50
    Output "Enter a number"
    INPUT Number
    Total ← Total + Number //Totalling
NEXT Count
OUTPUT "Total =", Total
```

To find total of 50 numbers stored in a 1D array

```
Total   0 // Intialization
FOR Count   1 TO 50
    Output "Enter a number"
    INPUT Number[Count]
    Total ← Total + Number[Count] //Totalling
NEXT Count
OUTPUT "Total =", Total
```

COUNTING

Counting is working out how many of something there are.

For example how many numbers were entered that were over 10.

To write a program to count you need to:

- Initialise a counter variable to 0.
- Increment (add one to) the counter each time an item is entered, or found .

```
Counter 0 //Intialiase with 0 outside the loop
//Inside the loop update Counter
Counter Counter + 1 //Counting statement
//Outside the loop, display the count
OUTPUT "Number of ..... =", Counter
```

To count the number of students passed when the mark are input into a variable

```
PassCount ← 0 //Initialization
FOR Counter ← 1 TO ClassSize
    INPUT Mark
    IF Mark > 50
      THEN
        PassCount ← PassCount + 1 //Counting
    ENDIF
NEXT Counter
OUTPUT "Number of students passed=",
PassCount
```

To count the number of students passed. Mark is stored in an array

```
PassCount ← 0 //Initialization
FOR Counter ← 1 TO ClassSize
    INPUT Mark[Counter]
    IF Mark [Counter]> 50
      THEN
        PassCount ← PassCount + 1 //Counting
    ENDIF
NEXT Counter
OUTPUT "Number of students passed=", PassCount
```

FINDING AVERAGE

- The **average** here is referring to the mean.
- This is the total of all the values added together, then divided by how many numbers there are.
- For example, if the data is 1, 3, 5, 8, 4, 2, 6, 9 , the average = (1 + 3 + 5 + 8 + 4 + 4 + 6 + 9) / 8 = 40 / 8 = 5
- The average is then calculated with the formulae **Total / Count.**

To find average mark of students in a class, where number of students is stored in the variable ClassSize

```
Total ← 0 // Intialization
FOR Count ← 1 TO ClassSize
    Output "Enter mark"
    INPUT Mark
    Total ← Total + Mark //Totalling
NEXT Count
AvgMark ← Total/ ClassSize
OUTPUT "Total Mark =", Total, "Average = ",
AvgMark
```

To find average mark of students in a class, where number of students is stored in the variable ClassSize

```
Total ← 0 // Intialization
FOR Count ← 1 TO ClassSize
    Output "Enter mark"
    INPUT Mark[Count]
    Total ← Total + Mark[Count] //Totalling
NEXT Count
AvgMark ← Total/ ClassSize
OUTPUT "Total Mark=", Total, "Average = ",
AvgMark
```

FINDING MINIMUM

The **minimum** value is the smallest value within a set. For example, the marks for a class are entered and the lowest mark is the minimum.

To do this in a program you need to:

- Initialise a minimum variable to be a large value, beyond those that will be entered.
- e.g. Minimum ← 9999.
- Compare each value to the minimum variable, e.g. IF Number < Minimum.
- If it is, it replaces the minimum value, e.g. Minimum ← Number.
- This means the new value is the minimum, so the next time a value is compared, it is checking it with the new minimum value.

To find lowest mark from a class of students with strength ClassSize

```
Minimum  10000 // Intialization
FOR Count  1 TO ClassSize
    Output "Enter mark"
    INPUT Mark
    IF Mark < Minimum //Comparing
        THEN
            Minimum  Mark
    ENDIF
NEXT Count
OUTPUT "Lowest Mark = ", Minimum
```

To find lowest mark from a class of students with strength ClassSize. Mark is stored in an array

```
Minimum  10000 // Intialization
FOR Count  1 TO ClassSize
    Output "Enter mark"
    INPUT Mark[Count]
    IF Mark[Count] > Minimum //Comparing
        THEN
            Minimum  Mark[Count]
    ENDIF
NEXT Count
OUTPUT "Lowest Mark = ", Minimum
```

FINDING MAXIMUM

The **maximum** value is the largest value within a set.

For example, the marks for a class are entered and the highest mark is the maximum.

To do this in a program you need to:

- Initialise a maximum variable to be a small value, beyond those that will be entered
- e.g. Maximum ← - 9999.
- Compare each value to the maximum variable, e.g. IF Number > Maximum.
- If it is, it replaces the maximum value, e.g. Maximum ← Number.
- This means the new value is the maximum, so the next time a value is compared, it is checking it with the new maximum value.

To find highest mark from a class of students with strength ClassSize

```
Maximum  -10000 // Intialization
FOR Count  1 TO ClassSize
     Output "Enter mark"
     INPUT Mark
     IF Mark < Maximum //Comparing
          THEN
            Maximum  Mark
     ENDIF
NEXT Count
OUTPUT "Highest Mark = ", Maximum
```

To find highest mark from a class of students with strength ClassSize. Mark is stored in an array

```
Maximum  -10000 // Intialization
FOR Count  1 TO ClassSize
     Output "Enter mark"
     INPUT Mark[Count]
     IF Mark[Count] > Maximum //Comparing
          THEN
            Maximum  Mark[Count]
     ENDIF
NEXT Count
OUTPUT "Highest Mark = ", Maximum
```

BUBBLE SORT

A **sorting algorithm** takes a set of data and rearranges it to be in a specific order, e.g. in ascending or alphabetical order. One example of this is a bubble sort.

Bubble Sort - A sorting algorithm that moves through the list repeatedly swapping values in pairs.

```
Last  50
REPEAT
  Swap  FALSE //Swap is a Boolean variable
  FOR Index  1 TO Last - 1
     IF Values[Index] > Values[Index + 1]
          THEN
            Temp  Values[Index]
            Values[Index]  Values[Index + 1]
            Values[Index + 1]  Temp
            Swap  TRUE
     ENDIF
  NEXT
  Last  Last - 1
UNTIL NOT Swap OR Last = 1
```

```
Last  50
REPEAT
  Swap  0 // Swap is an integer variable
  FOR Index  1 TO Last - 1
    IF Values[Index] > Values[Index + 1]
      THEN
        Temp  Values[Index]
        Values[Index]  Values[Index +
1]
        Values[Index + 1]  Temp
        Swap  1
      ENDIF
  NEXT
  Last  Last - 1
UNTIL Swap=0 OR Last = 1
```

To do bubble sort in a 2D array with 2 columns, the data needs to be sorted based on one of the columns.

```
Last ← 50
REPEAT
 Swap ← FALSE
 FOR Index ← 1 TO Last - 1
   IF Values[Index,1] > Values[Index + 1,1] //Using first column details
     THEN
          //Swapping the first column values
      Temp1 ← Values[Index,1]
      Values[Index,1]← Values[Index + 1,1]
      Values[Index + 1,1]] ← Temp1
          //Swapping the second column values
      Temp2 ← Values[Index,2]
      Values[Index,2]← Values[Index + 1,2]
      Values[Index + 1,2]] ← Temp2
      Swap ← TRUE
   ENDIF
 NEXT
 Last ← Last - 1
UNTIL NOT Swap OR Last = 1
```

LINEAR SEARCH

A **linear search** will check each item one at a time, starting with the first item and continuing until it either finds the item, or it checks the last value.

```
OUTPUT "Enter name to find "
INPUT Name
Found ← FALSE
Counter ← 1
REPEAT
   IF Name = Name[Counter]
      THEN
        Found ← TRUE
   ELSE
      Counter ← Counter + 1
   ENDIF
UNTIL Found OR Counter > ClassSize
IF Found
   THEN
      OUTPUT Name, " found"
ELSE
     OUTPUT Name, " not found."
ENDIF
```

```
OUTPUT "Enter name to find "
INPUT Name
Found ← 0
Counter ← 1
REPEAT
   IF Name = Name[Counter]
      THEN
        Found ← 1
   ELSE
      Counter ← Counter + 1
   ENDIF
UNTIL Found=1 OR Counter > ClassSize
IF Found=1
   THEN
      OUTPUT Name, " found"
ELSE
     OUTPUT Name, " not found."
ENDIF
```