TOPIC:

LIST PROCESSING USING ARRAYS
Apart from using simple data types in algorithms, in the average problem we can use three variables; \textit{num1}, \textit{num2}, \textit{num3}. These variables are independent of each other in terms of where they are stored in memory and also in terms of what type of data they store.

\textit{Num1}: stores an integer value

\hspace{1cm} stored at location 101 in memory

\textit{Num2}: stores a real value

\hspace{1cm} stored at location 1510
An array is a more complex data type. It is a structure which contains not just one data value, but multiple data values, all of the same type.

The values are all integers or all floating point numbers or are all characters.

The values are stored in contiguous (meaning adjacent) memory locations.

So for example; if the first value is stored at location 100, then the second value would be stored at location 101, the third 102... and so on.
An array is a data structure that is used to store a fixed number of data items all of the same type. The items (or elements) of the array are organized in sequence and can be accessed directly by specifying their positions in the sequence, using an index or subscript.
One-dimensional arrays are when only one index is used. If more than one index is used it is called a multi-dimensional array.

One-dimensional arrays: are list structures or vectors. Two-dimensional arrays: are tables structures or matrices.
A one-dimensional array can be represented as follows:

<table>
<thead>
<tr>
<th>TEMP [1]</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMP [2]</td>
<td>28</td>
</tr>
<tr>
<td>TEMP [3]</td>
<td>31</td>
</tr>
<tr>
<td>TEMP [4]</td>
<td>29</td>
</tr>
<tr>
<td>TEMP [5]</td>
<td>30</td>
</tr>
<tr>
<td>.........</td>
<td>:</td>
</tr>
<tr>
<td>TEMP [50]</td>
<td>:</td>
</tr>
</tbody>
</table>

Data values stored in adjacent memory locations.
To refer to the 2\textsuperscript{nd} temperature value, 28 we would specify in the array as a subscript of the array name. That is,

\begin{equation}
\text{Temp} [2]
\end{equation}

\textit{Subscript or index (that is, the position of the item in the array)}

\textbf{Temp} is the name of the array, 2 indicates the location of the value 28.
Arrays are sometimes called subscripted variables because the values stored in the locations of an array are accessed via the subscript or index of the array.

**Purpose of arrays:**
they are used to store and process a list of items. That is why sometimes manipulation of arrays can be termed as *list processing*.

**Example:**
A program is required to read a list of temperature values and find their mean.

Two Possible Solutions:

(i) If the list is short, say 3 temperature values, we may use 3 different variables, say `temp1`, `temp2`, `temp3`.

(ii) If the list is long, say 120 temperature values, it would be cumbersome and awkward to use 120 different variables.
The elements of an array can be accessed individually by:
- specifying the name of the array
- the index or subscript (this identifies the position of the element in the sequence.

When *manipulating arrays*, a special variable must be declared as the index of the array.

A single letter (such as i, j, or k) is commonly used as array index. Using the index, the array elements can be manipulated in the same way that we manipulate ordinary variables.

Traversing the array (that is, access each element in a sequential manner) requires a loop structure.
Initializing Arrays

Initial values can be assigned to array locations either by reading values directly into the locations or by use of an assignment statement.

For example: set all 10 locations of the array list to 0, initially.

Set i to 1
Repeat 10 times

\[
\begin{align*}
&\text{list } [i] \leftarrow 0 \\
&\text{increment } i \\
&\text{end-repeat}
\end{align*}
\]

The first time the loop is executed, i=1 and list [1] would be assigned 0. The second time through the loop i=2 and list [2] would be assigned 0. The third time through the loop, i=3 and list [3] would be assigned and so on, until the last location, that is list [10] is assigned 0.
NOTE:

When manipulating arrays, a special variable must be declared for use as the index of the array.

It is better to use short variable names, such as single letters of the alphabet for the index or subscript.
If the number of values to be read is known beforehand, we use a counted loop (that is, repeat ‘n’ times.)

If the number of values is unknown, we use a conditional loop (that is, while some condition is true do).
To read 10 values into the array \textit{temp}:

\begin{itemize}
  \item \textit{Set i to 1}
  \item \textit{Repeat 10 times}
    \begin{itemize}
      \item \textit{read temp \[i\]} \hspace{10pt} \{ \textit{get next value and store it in location temp[i]} \}
      \item \textit{increment i} \hspace{10pt} \{ \textit{add 1 to index to get to the next array location} \}
    \end{itemize}
  \item \textit{end-repeat}
\end{itemize}

If the input data was say, 21 23 25 26 22 27 29 26 24 26
First time through the loop, $i = 1$, the value 21 is stored in temp [1]

Second time through the loop, $i = 2$, the value 23 is stored in temp [2]

Third time through the loop, $i = 3$, the value 25 is stored in temp [3]

Fourth time through the loop, $i = 4$, the value 26 is stored in temp [4]

And so on... until the last location temp [10] is filled with the value 26

After the loop has been executed 10 times, the array would look like this:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>21</td>
<td>23</td>
<td>25</td>
<td>26</td>
<td>22</td>
<td>27</td>
<td>29</td>
<td>26</td>
<td>24</td>
<td>26</td>
</tr>
</tbody>
</table>

*The top numbers represented the index or subscript of the array, the bottom numbers are the data values stored in the various locations.*
The case of the Unknown number of items:

Consider the following problem:

Write an algorithm that reads a list of integers from the input stream and stores the values in an array called Positive_Num.

The list is terminated by a sentinel value of -1.
Here the number of items to be read is unknown, so we use a conditional loop as follows:

Set $i$ to 1
Read number
While (number is NOT equal to -1) do the following:
   Positive_Num[$i$] $\leftarrow$ number \{store current number in array location\}
   Increment $i$ \{add 1 to index to get to next location\}
   Read number \{read next number\}
End-while
Num_of_Items $\leftarrow$ $i = 1$

The loop will execute until number receives a value of -1. The -1 will not be stored in the array. The last statement stores the actual number of data items read in the variable Num_of_Items.
This value will be needed in future if we need to print the data values or traverse the array for any other purpose.
Writing or printing values stored in arrays is similar to reading values into arrays, except that, in the case of writing, we would already know the actual number of items stored in the array.

Therefore, we can simply use a counted loop as in the example below:

```
Set j to 1
Repeat 10 times
    display temp [j]
    increment j
end-repeat
```
Alternatively, if we stored the actual number of items read in a variable called Num_of_Items, we could display the items in array list as follows:

- Set j to 1
- While \((j \leq \text{Num\_of\_Items})\) do the following:
  - Display list \([j]\)
  - Increment \(j\)
- End-while
Traversing an array simply means:
Moving through the array in a sequential manner, visiting each element, in order to manipulate the elements in one way or another.

Example:

An array is traversed when we print the elements, or when we search the list for a particular item OR when we sort the list of items in a particular order.

In traversing an array, we must establish a loop and increment the index within the loop in order to get to the next element.
A classic example of traversing an array is to perform a linear search.

How a linear search is done?
Linear search involves examining each element in the array, one by one, starting with the first element and comparing each element with the item/value being searched.

The search ends when a match is found or when the end of the array is reached. The following is an algorithm for performing a linear search of an array.
This algorithm searches a list for the presence of an item called, target. In this case, a target is an integer value. A Boolean (meaning a system of algebraic notation) variable found is set to true if the target value is found, otherwise, Found is set to false.

Set Size to 50
Set target to 27
Set index to 1
Set Found to false

While ((Found is false) AND (index <= Size))
    if (list[index] = target) then
        set Found to true \textit{if value is found set the flag to true}
    else
        increment index \textit{otherwise, move on to the next element in the array}
end-while
If (found = true) then
    display “Target found at location”, index
else
    display “Target not found”
Stop.
Design an algorithm that reads a list of students test scores and determine the number of students who failed the test. A student is deemed to have failed if his/her score is less than the class average. The list is terminated by a value of 999. Print the class average as well as the number of students who failed.

Defining Diagram:

<table>
<thead>
<tr>
<th>Input</th>
<th>Processing</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read List of test-scores</td>
<td>1. Calculate average score</td>
<td>Average</td>
</tr>
<tr>
<td></td>
<td>2. Count number of scores that are less than the average</td>
<td>Number -Failed</td>
</tr>
<tr>
<td></td>
<td>3. Print average, number-failed</td>
<td></td>
</tr>
</tbody>
</table>
This algorithm reads a list of test scores and determines the number of scores that fall below the class average.

**Variables Used:** scores is the array which stores the test scores, i is the index to the array, sum scores the total scores, Number-Failed stores the number of students who failed the test.

Set i to 1
Set sum to 0
Set Number-Failed to 0
Read Num
While (num not equal to 999) do the following:
   - Scores[i] ← num  \{store each data value in the array\}
   - Sum ← sum + scores[i]  \{compute the sum of the scores\}
   - Increment i  \{advance to the next location in the array\}
   - Read Num
End-while
Set No-of-Items to i
average ← sum ÷ No-of-Items

\{Traverse array to count how many values are less than average\}
Set Index to 1

Repeat No-of-Items times \{counted loop\}

If (scores [index] < average) then

    Number – Failed \leftarrow Number - Failed + 1

End-if

Increment index

End-repeat

Display “The average test score is:”, average

Display “The number of students who failed is:”, Number-Failed

Stop.
Points to Note When Manipulating Arrays

- It is illegal to refer to an element outside the array bounds. For example, if the size of an array temp is declared to be 50, then it would be illegal to have a statement such as:

Print temp [51]

- Therefore, when looping through an array, the subscript should never go below 1 and should always be less than or equal to the total number of elements in the array. So temp [-1] should be an illegal reference since the index is negative.

- Some programming languages such as C requires that the first element in an array start at subscript 0, and the last element at subscript SIZE-1. Other languages such as Pascal require that the initial subscript be 1 and the last element is at subscript SIZE. For the purposes of an algorithm, which should be language-independent, it is recommended that array subscripts start at 1.
Points to Note When Manipulating Arrays

- All the elements in an array must be of the same type. That is, all integers, or all real numbers, or all characters.

- The size of the array is fixed when the array is declared in a programming language. For example, an array may be declared to have 100 locations. This does not mean that there are 100 elements in the array. This means that 100 locations are reserved in memory for a particular array or list. Depending on the number of data items in the input list, all 100 locations may not be used. If there are 75 data items, then only 75 locations of the array will have data; if, on the other hand, there are 100 data items, then all 100 locations will be used. The algorithm should keep track of the number of data items stored in the array and manipulate the list accordingly. There must also be a means of determining which locations have values and which are empty.