Computer Science
Teacher: Maruf Ahmed

Bubble Sort in Ascending order (inefficient one):

- The outer loop will be used to count the number of passes which will be maximum array size – 1 (if array index starts from 1) or - 2 (if array index starts from 0) depending on the starting index
- We need to compare two adjacent elements till the end of the array. For example we compare first value with second value and we find out that first value is bigger and second value is smaller. Then we have to swap the two values using a temporary variable. This is how it will continue
- This is to be remembered that the last comparison would be the size of the array – 1 (if array index starts from 1) or - 2 (if array index starts from 0) depending on the starting index
- In each scan/pass we would get the largest value at the end of the array. The other values may or may not be in the correct order.
- We would need two loops. The outer loop would continue till the array size – 1 position or - 2 positions depending on the staring index to control the passes and the inner loop would be used to compare between two adjacent elements till the array size – 1 position or - 2 positions depending on the staring index
- For Ascending order the comparison will be greater than (>) and for descending order the comparison will be smaller than (<)

Bubble Sort in Ascending order (efficient one):

- The outer loop will be used to count the number of passes which will be maximum array size – 1 (if array index starts from 1) or - 2 (if array index starts from 0) depending on the starting index
- We need to compare two adjacent elements till the end of the array. For example we compare first value with second value and we find out that first value is bigger and second value is smaller. Then we have to swap the two values using a temporary variable. This is how it will continue
- This is to be remembered that the last comparison would be the size of the array – 1 (if array index starts from 1) or - 2 (if array index starts from 0) depending on the starting index
- In each scan/pass we would get the largest value at the end of the array. The other values may or may not be in the correct order.
- We would need two loops. The outer loop would continue till the array size – 1 position or – 2 positions. Since for each iteration of outer loop one value will be fixed at its position that is why the inner loop would be reduced by 1 every time so that unnecessary checking is not done
- If we have gone through the whole of the inner loop (one entire pass) without swapping any values, we know that the array elements must be in the correct order. We can introduce a FLAG variable to accomplish this. In that case we have to come out of the outer loop as the array is already sorted
- For Ascending order the comparison will be greater than (>) and for descending order the comparison will be smaller than (<)

What is a flag variable?
A flag variable, in its simplest form, is a variable you define to have one value until some condition is true, in which case you change the variable's value.

Flag variable is used as a signal in programming to let the program know that a certain condition has met. It usually acts as a boolean variable indicating a condition to be either true or false.

In Bubble sort we have to use swapping technique
A ← 5
B ← 10
Swap the value of A and B
We need to introduce a third variable so that no value gets lost. The process is as follows:
Temp ← A
A ← B
B ← Temp

Nested loop: When one loop is inside another loop this is known as nested loop. The first loop is known as the outer loop and the second loop is known as the inner loop. For every iteration of the outer loop there is a complete cycle of the inner loop.

Bubble sort in ascending order:
The following values are stored in an array. The name of the array is Sort

15	16	6	8	5

****
Solution of Bubble sort (inefficient one): The first index starts at 1.

```
FOR i ← 1 to LENGTH(Sort) - 1
    FOR j ← 1 TO LENGTH(Sort) - 1
        IF Sort[j] > Sort[j+1] THEN
            Temp ← Sort[j]
            Sort[j] ← Sort[j+1]
            Sort[j+1] ← Temp
        ENDIF
    NEXT j
NEXT i
```

****
Solution of Bubble sort (inefficient one): The first index starts at 0.

```
FOR i ← 0 to LENGTH(Sort) - 2
    FOR j ← 0 TO LENGTH(Sort) - 2
        IF Sort[j] > Sort[j+1] THEN
            Temp ← Sort[j]
            Sort[j] ← Sort[j+1]
            Sort[j+1] ← Temp
        ENDIF
    NEXT j
NEXT i
```

****
Efficient one with post-conditional loop. The first index starts at 1.

```
n ←  LENGTH(Sort) - 1
REPEAT
    WasSwapped ← FALSE
    FOR j ← 1 TO n
        IF Sort[j] > Sort[j+1] THEN
            Temp ← Sort[j]
            Sort[j] ← Sort[j+1]
```

```
            Sort[j+1] ← Temp
            WasSwapped ← TRUE
        ENDIF
    NEXT j
    n ←  n - 1
UNTIL WasSwapped = FALSE
```

****

Efficient one with pre-conditional loop. The first index starts at 1.

```
n ←  LENGTH(Sort) - 1
WasSwapped ← TRUE
WHILE WasSwapped = TRUE DO
    WasSwapped ← FALSE
    FOR i ← 1 TO n
        IF Sort[i] > Sort[i+1] THEN
            Temp ← Sort[i]
            Sort[i] ← Sort[i+1]
            Sort[i+1] ← Temp
            WasSwapped ← TRUE
        ENDIF
    NEXT i
    n ←  n - 1
ENDWHILE
```

****

Suppose a 2D array has been created with 5 rows and 2 columns. First column stores Student ID and second column stores marks of students. Sort the data of the second column in Ascending order

```
ArrayList=[[1,70],
           [2,15],
           [3,55],
           [4,21],
           [5,71]]
```

#solution

```
boundary ← 5
Noswaps ← FALSE
WHILE Noswaps = FALSE DO
    noswaps ← TRUE
    FOR row ← 1 TO boundary - 1
        IF ArrayList[row,2] > ArrayList[row+1,2] THEN
            FOR column ← 1 TO 2
                Temp ← ArrayList[row, column]
                ArrayList[row, column] ← ArrayList[row+1, column]
                ArrayList[row+1,column] ← temp
            NEXT column
            Noswaps ← FALSE
        ENDIF
    NEXT row
    boundary ← boundary-1
ENDWHILE
```

****

Bubble sort using 2D array (Python code):

```
result=[[1,70],
        [2,15],
        [3,55],
        [4,21],
        [5,71]]

boundary=5
noswaps=False
while noswaps==False:
    noswaps=True
    for row in range(boundary-1):
        if result[row][1]>result[row+1][1]:
            for column in range(2):
                temp=result[row][column]
                result[row][column]=result[row+1][column]
                result[row + 1][column]=temp
            noswaps=False
    boundary=boundary-1

print(result)
```

16	14	5	6	8

Insertion sort

Insertion sort is a sorting mechanism which takes one value at a time. It takes one value and restarts after adjusting that value. The following algorithm is based on lower bound of the array as 0 and the sorting has been performed in ascending order.

Algorithm of Insertion Sort:

- The array will be divided into two parts. The sorted part and the unsorted part. The sorted part will be having only one value and the unsorted part will be having the other values.
- Only one value will be extracted at a time from the unsorted part and can be placed in its correct position in the sorted part. That means the sorted part will keep on increasing and the unsorted par will keep on decreasing
- Start from the second element of the array
- Extract the second element which is the current element to be placed in its appropriate position. In the next turn extract the third element and do the same for all the other elements one by one
- Check if the previous element is greater than the current extracted element. If it is true then place the previous element in current element's position. Continue this checking with all the previous elements one by one. Once the previous element is not greater than the current element then place the current element in the next index position

How to achieve the algorithm for an array whose index starts at 0

- Start an outer loop which will start from second element of the array and continue till the end of the array
    - Extract the element to be checked from second position in the array and put it in a variable
    - Reduce the index of the array by 1 and place the new index in another variable
    - Run an inner loop based on a condition. As long as the value of the reduced index is greater than the extracted value and the index is greater than or equal to 0 then
        - Place the array element of reduced index value to the next index position of the array
        - Reduce the index by 1
        - This loop will continue as long as both the conditions in the loop are true
    - Place the extracted element in the array at position of the index + 1

We can write this algorithm using pseudocode. Assume the values to be sorted are stored in a 1D array, List which has the lower bound as 0.

```
List ← [5,4,10,1,6,2]
NumberOfitems←LENGTH(List)
FOR Pointer ← 1 TO NumberOfitems -1
   ItemToBeInserted ← List[Pointer]
   CurrentItem ← Pointer – 1
   WHILE (List[CurrentItem] > ItemToBeInserted) AND (CurrentItem >-1) DO
      List[CurrentItem + 1] ← List[CurrentItem]
      CurrentItem ← CurrentItem – 1
   ENDWHILE
   List[CurrentItem + 1] ← ItemToBeInserted // insert item
NEXT Pointer
```

We can write this algorithm using pseudocode. Assume the values to be sorted are stored in a 1D array, List which has the lower bound as 1.

```
List ← [5,4,10,1,6,2]
NumberOfitems←LENGTH(List)
FOR Pointer ← 2 TO NumberOfitems
    ItemToBeInserted ← List[Pointer]
    CurrentItem ← Pointer – 1
    WHILE (List[CurrentItem] > ItemToBeInserted) AND (CurrentItem > 0) DO
        List[CurrentItem + 1] ← List[CurrentItem]
        CurrentItem ← CurrentItem – 1
    ENDWHILE
    List[CurrentItem + 1] ← ItemToBeInserted // insert item
NEXT Pointer
```

Another way to do this:

```
#Starting index is 0 here
FOR Pointer ← 1 TO (Max − 1)
    ItemToInsert ← Numbers[Pointer]
    CurrentItem ← Pointer
    WHILE (Numbers[CurrentItem − 1] > ItemToInsert) AND (CurrentItem > 0) DO
        Numbers[CurrentItem] ← Numbers[CurrentItem − 1]
        CurrentItem ← CurrentItem – 1
    ENDWHILE
    Numbers[CurrentItem] ← ItemToInsert
NEXT Pointer
```

State two situations when an insertion sort is more efficient than a bubble sort

- When the list is almost sorted because it will stop as soon as it is sorted
- When there are a large number of data items because it will perform fewer comparisons/loops

Linear search:
In Linear Search every element will be searched sequentially i.e. one after another to find the value. Suppose an array already has some values stored in it. You have to take a value as input from a user and check if the value is in the array. If the value is found then display the **index number** where it was found. If the value was not found then display a message. Assume that the name of the array is ArrayList of any size. The lower bound of the array is 0.

#Linear search using 1D Array

Solution using WHILE loop

```
ArrayLength← LENGTH(ArrayList)
Index ← 0
IsFound ← FALSE
OUTPUT “Please enter the value: ”
INPUT Item
WHILE IsFound = FALSE AND Index <ArrayLength DO
    IF ArrayList[Index] = Item THEN
        OUTPUT “The value found at location ”, Index
        IsFound ← TRUE
    ELSE
        Index ← Index + 1
    ENDIF
ENDWHILE
IF IsFound = FALSE THEN
    OUTPUT “The value was not found”
ENDIF
```

#Solution using REPEAT loop for 1D array

```
ArrayLength← LENGTH(ArrayList)
Index ← 0
IsFound ← FALSE
OUTPUT “Please enter the value: ”
INPUT Item
REPEAT
    IF ArrayList[Index] = Item THEN
        OUTPUT “The value found at location ”, Index
        IsFound ← TRUE
```

```
    ELSE
        Index ← Index + 1
    ENDIF
UNTIL IsFound = TRUE OR Index >=ArrayLength
IF IsFound = FALSE THEN
    OUTPUT “The value was not found”
ENDIF
```

#Linear search using 2D Array.

Solution using WHILE loop using 2D array

```
ArrayLength← LENGTH(ArrayList)
IsFound ← FALSE
OUTPUT “Please enter the value: ”
INPUT Item
WHILE IsFound = FALSE AND Index <ArrayLength DO
    FOR Row = 0 TO ArrayLength -1
        FOR Column ← 0 TO 2 #Assuming that it has 3 columns in it
            IF ArrayList[Row, Column]= Item THEN
                OUTPUT “The value found at location ”, Row, Column
                IsFound ← TRUE
            ELSE
                Index ← Index + 1
            ENDIF
        NEXT Column
    NEXT Row
ENDWHILE
IF IsFound = FALSE THEN
    OUTPUT “The value was not found”
ENDIF
```

#Linear search using 2D Array

Solution using REPEAT loop using 2D array

```
ArrayLength← LENGTH(ArrayList)
IsFound ← FALSE
INPUT “Please enter the value: ”, Item
```

```
REPEAT
    FOR Row = 0 TO ArrayLength -1
        FOR Column ← 0 TO 2 #Assuming that it has 3 columns in it
            IF ArrayList[Row, Column] = Item THEN
                OUTPUT “The value found at location ”, Row, Column
                IsFound ← TRUE
            ELSE
                Index ← Index + 1
            ENDIF
        NEXT Column
    NEXT Row
UNTIL IsFound = TRUE OR Index >=ArrayLength
IF IsFound = FALSE THEN
    OUTPUT “The value was not found”
ENDIF
```

BINARY SEARCH

1. Array must be sorted in a particular order first. In this case it is sorted in Ascending order.
2. The array will be divided into 2 equal parts recursively and every time the middle index will be calculated using the following formula
    a. (LB + UB) DIV 2 or INT((LB+UB)/2)
3. We will get middle index in whole number.
4. There could be three cases.
    a. Case 1: The value will be checked with the middle index value and is found at the middle index
    b. Case 2: The value we are looking for is greater than the middle index value. In this case we have to search the right part of the array and will get a new LB by LB ← Middle+1.
    c. Case 3: The value we are looking for is smaller than the middle index value. In this case we have to search the left part of the array and will get a new UB by UB ← Middle-1.
5. If the value is found then it will be found at the middle position only
6. This searching will continue as long as UB is greater than or equal to LB. When LB becomes greater than UB then the loop will terminate and that means the value is not present in the array

PSEUDOCODE for Binary Search:

Suppose an array has values as: 'a','b','c','d','e','f' and the name of the array is Alpha

```
DECLARE Alpha : ARRAY[1:6] OF CHAR
OUTPUT 'Enter a value you are looking for: "
INPUT V2S
IsFound ← FALSE
UB ← LENGTH(Alpha)
LB ← 1
WHILE UB>=LB AND IsFound = FALSE DO
   Middle ← INT((UB+LB)/2)
   IF Alpha[Middle] = V2S THEN
      IsFound ← TRUE
      OUTPUT("Alphabet Found at: ", Middle)
   ELSE
      IF Alpha[Middle]<V2S THEN
         LB ← Middle+1
      ELSE
         UB ← Middle-1
      ENDIF
   ENDIF
ENDWHILE
IF IsFound = FALSE THEN
   OUTPUT "Alphabet Not Found"
ENDIF
```

PYTHON CODE FOR BINARY SEARCH:

```
a=[5,9,17,23,25,45,59,63,71,89]
ub=len(a)-1 #Since array index starts from 0 that is why length - 1 is taken
lb=0
isFound=False
v2s=int(input('Enter a value to search: '))
while ub>=lb and isFound==False:
    #mid=int((ub+lb)/2)
    mid = (ub + lb) // 2
    if a[mid]==v2s:
        print('Value found at location ',mid)
        isFound=True
    elif a[mid]<v2s:
        lb=mid+1
    else:
        ub=mid-1
if ub<lb:
    print('Value could not be found...')
```

5 9 17 23 25 45 59 63 71 89

Suppose we are looking for value 60