Name: Debnath Kundu (MT22026) & Manvendra Kumar Nema (MT22038)

Answer for Q1:

// this function will be called as BubbleSort(A,n)

```
1.  def BubbleSort(A,i):
            #A : is array of numbers, i :is size of array
2.      if i<=1:
3.          return
4.      MoveMaxtoLast(A,i,i)
5.      BubbleSort(A,i-1)

6.  def MoveMaxtoLast(A,i,m):
    # A : is array of numbers, i : is index of array, m : is the size of array
7.      if i<=1:
8.          return
9.      if a[m-i] > a[m-i+1]:
10.             swap( a[m-i],a[m-i+1] )
11.     MoveMaxtoLast(A,i-1,m)
```

Answer for Q2:

[Note: Index range from i=0 to (n-1)]

In the **MoveMaxtoLast** function the value of i varies from **m** to **1**, thus **(m-i)** varies from **0** to **(m-1)**. The **MoveMaxtoLast** function swap value of **(m-i)th** index with **(m-i+1)th** index if **A[m-i]** > **A[m-i+1]**, else the function recursively call itself with **(i-1),** thereby checking further down the array for any other swaps, if any.
Hence, the minimum value of **i** can be **1**, which is also the ***base case*** because when **i=1**, that means that the last element of the given array now contains the largest value. After the end of each recursive call, **i** is reduced so that **(m-i)** increases, thus the next index can be accessed until the end of the array is reached.
The **MoveMaxtoLast** function swaps **A[current index]** with **A[next index]** if they are not in ascending order. As a result, the then-maximum element is relocated into the proper position at the conclusion of each function call.

BubbleSort recursively calls the **MoveMaxtoLast** function with array **A[0...m-1]** which puts the largest element, say **A[k]** into its place i.e. last index **A[m-1]**. Then recursively reduce the size of the array to **A [0...m-1]** and call the last index again until the size of the array is 1, which is trivially sorted. So in each recursive iteration, the largest element gets into place until a single element is left. In **kth** recursive call last **(k-1)** elements are in their correct place i.e. in sorted order.
BubbleSort function is called **BubbleSort(A,n)** where **A** is the array of size **n**.

Proof by Induction (MoveMaxtoLast) :

- *Base case* - while comparing and swapping consecutive elements the end of array is reached, that is, i=1.

- *Induction hypothesis* - Assume that **MoveMaxtoLast**(A,i,m) puts the last element at the end of the sub-array for all i<=k.
- *Induction claim* - to show that at the end of function call, the largest element reaches the last index of subarray i.e **(m-1)th** index A[0…m-1].
  Here m is the size of the sub-array.

Two cases can happen to A:

a. A[m - k] > A [m – k +1]

b. A[m - k] < A [m – k +1]

i=k+1

Case(a): Line 8 is a no-op. Condition in line 9 is true. Hence, the if conditional block will get executed. Before executing line 10, **A[0…k-1]** is unsorted and **A[k…n]** is sorted. After line 10, since A[m-i] > A[m-i+1] so it will get swapped & then the recursion is invoked until the largest element in subarray goes to last index, i.e. it will be placed at the **(m-1)th** index of the sub-array.
After the end of the recursive call, the array **A[k-1…n]** is sorted.

Case(b): Line 8 is a no-op. Since the if condition in line 10 is not satisfied, this call will not modify A, and the value of **(m-i)** will keep increasing until the end of the array. If no change occurs, then at the end of that recursive call, the last element present at **(m-1)th** index must be the largest element, which is already in its correct place.

***Proof by Induction:* (BubbleSort):**

- *Base case* - i=1, A[0] array is a single element array, so the array is sorted trivially & the largest element is in its correct place.
- *Induction hypothesis* - Assume that **BubbleSort(A,i)** places the **kth** largest element in place for all **i<=k**.
- *Induction claim* - To show that at the end of function call (**k+1)th** largest element is put in place, i.e., the sub-array A[n-k,...n] is sorted when **BubbleSort** returns.

i=k+1

Line 3 is a no-op, until & unless the array size reduces to 1. Otherwise, ***MoveMaxtoLast*** is called with the 1st argument **i** which represents the last index of the array & the 2nd argument **i** represents the size of the array. After line 4 is executed, the **ith** largest element will be placed correctly in the last index of that **i-sized** sub-array.
Then the function recursively calls itself BubbleSort**(A,i-1).** So, with each recursion, the largest element is correctly placed in its correct position in the smaller sub-array. To generalize, the initial part of the array A[0…i-1] is not sorted while the later part of the array A[i...n] is always sorted.

As a single element array is considered to be easily sorted, this process continues until the array size drops to 1, which is also the base case, at which point the method returns.

Answer for Q3:

Proof of complexity-

Let the time complexity of **MoveMaxtoLast** & **BubbleSort** functions be denoted by **MML(n)** and **BS(n)**. Since each call of **MoveMaxtoLast** function will recursively call itself **(n-1)** times, thus:

$MML(n) = MML(n-1) + d$ **…(1)** *[where d is a constant]*

Base case: $MML(1) = 1$ when only one element left to put in last place that means it is in place.

$MML(n-1) = MML(n-2) + (d)$

From (1) *[by substitution]* :

$MML(n) = MML(n-2) + (d + d)$

…..

$MML(n) = MML(n-k) + k(d)$

$Let\ n - k = 1 \Rightarrow k = n - 1$

$MML(n) = MML(1) + (n-1)d$

$MML(n) = 1 + (n-1)d$

$MML\,(n) = O(n)$

The **BubbleSort** method recursively calls itself **n** times during **i=n down to 1**, thereby also invoking **MoveMaxtoLast** method **n** times. Now, the **MoveMaxtoLast** method calls itself **n** times during its recursion.

Base case: $BS(1) = 1$, $i.e.$ when only one element of the array is present, it's already sorted.

$BS(n) = BS(n-1) + d'n$ $[\,d'\ is\ a\ constant\,]$

$BS(n-1) = BS(n-2) + d'(n-1)$

$BS(n) = BS(n-2) + (\,d'(n-1) + d'n\,)$

$BS(n) = BS(n-k) + (\,d'(n-k+1) + \ldots\ d'(n-1) + d'n\,)$

$Let\,(n-k) = 1\,, \Rightarrow k = n - 1$

$BS(n) = BS(1) + d'\,(2 + 3 + \ldots n)$

$BS(n) = 1 + d'(n(n+1))/2 - d'$

$BS(n) = O(N\text{^}2)$

Therefore, the time complexity of the above recurrence relation is **$O(N^2)$.**