Task Force Zero00

assignment 001

1. An in-place sorting algorithm that finds max. element in each cycle and puts it in appropriate position in list by performing swapping adjacent elements.We continue swapping adjacent elements until they are in correct order.
    1. In first cycle,
        1. Start by comparing 1st and 2nd element and swap if 1st element is greater.
        2. After that do the same for 2nd and 3rd element.
        3. At the end of cycle you will get max element at the end of list.
    2. Now do the same in all subsequent cycles
    3. Perform this for (number of elements – 1) times.
    4. You will get sorted list.

Write a pseudocode from this above description.

2. Write a c++ code from this pseudocode and analyze the cost and times for each line

```
ZeroSort(Array, n)
{
    for i = 1 to n-1
    {
        value = Array[i]
        position = i
        while (position > 0 and Array[position-1] > value)
        {
            Array[position] = Array[position - 1]
            position = position - 1
        }
        Array[position] = value;
    }
}
```

3. Write a c++ code from this pseudocode and analyze the time complexity of the ZeroSearch

    hint: `ZeroSearch(array, start, end, key)`

    Input – An sorted array, start and end location, and the search key

    Output – location of the key (if found), otherwise wrong location.

```
Begin
   if start <= end then
      midFirst := start + (end - start) /3
      midSecond := midFirst + (end - start) / 3
      if array[midFirst] = key then
```

```
            return midFirst
         if array[midSecond] = key then
            return midSecond
         if key < array[midFirst] then
            call ZeroSearch(array, start, midFirst-1, key)
         if key > array[midSecond] then
            call ZeroSearch(array, midFirst+1, end, key)
         else
            call ZeroSearch(array, midFirst+1, midSecond-1, key)
      else
         return invalid location // that is return -1
End
```

4. Write down the c++ code from this pseudocode

```
Merge(LA, RA, A) // LeftArray (LA)  RightArray (RA)
{
    while(i<nL and j<nR)   // nL = Length of LeftArray (LA), nR = Length of RightArray (RA)
    {
        if(LA[i] <= RA[j])
        {
            A[k] = LA[i]
            i = i+1
        }
        else
        {
            A[k] = RA[j]
            j = j+1
        }
        k = k+1
    }
    while(i<nL)
    {
        A[k] = LA[i]
        i = i+1
        k = k+1
    }
    while(j<nR)
    {
        A[k] = RA[j]
        j = j+1
        k = k+1
    }
}
MergeSort(A)
{
    if(n<2)//n = length of Array (A)
        return
    for i = 0 to mid-1 // mid = n/2
        left[i] = A[i] // left = array of size(mid)
    for i = mid to n-1
        right[i-mid] = A[i] //right = array of size(n-mid)
    MergeSort(left)
    MergeSort(right)
    Merge(left, right, A)
}
```