Assignment 3

Name: Amit Kumar Roll No: 13096

Q1:

```
#1  Given List A
#2  count=0
#3  MergeSort(A,0,length(A)-1)
#4  print count
#5  Function MergeSort(A,p,q):
#6      If p>=q Then:
#7          return new List(A[p])
#8      Else:
#9          X=MergeSort(A,p,floor((p+q)/2))
#10         Y=MergeSort(A,floor((p+q)/2)+1,q)
#11         Z=Merge(X,Y)
#12     return Z
#13 Function Merge(X,Y):
#14     global count
#15     a=length(X)
#16     b=length(Y)
#17     i=0,j=0,k=0
#18     Z=Empty List
#19     While i!=a and j!=b Do:
#20         If X[i]<Y[i] Then:
#21             Z[k]=X[i]
#22             k=k+1,i=i+1
#23         Else:
#24             count+=a-i
#25             Z[k]=Y[j]
#26             k=k+1,j=j+1
#27     End While
#28     While i!=a Do:
#29         Z[k]=X[i]
#30         i=i+1,k=k+1
#31     End While
#32     While j!=b Do:
#33         Z[k]=Y[j]
#34         j=j+1,k=k+1
#35     End While
#36     Return Z
```

Explanation:

#5 **MergeSort** sorts the Array elements from p to q in nondecreasing order

#6 if start index is same as end it means there is only one element so we have to return that array

#7 create new list containing only one element A[p]

#9,10,11 divide array in two parts and sort them individually and then merge them

#13 **Merge** merges two given non-decreasing arrays in nondecreasing order, counts the number of inversions, finally returns merged array

#14 **count** is a global variable used to count total number of inversions

Complexity:

MergeSort function is only source for time complexity, so lets calculate complexity of MergeSort.
In MergeSort we call Merge which takes steps of order of n, so we need to add n to time complexity
So for MergeSort $T(n)=2*T(n/2) +$ n
Now using **Master's Theorem** Complexity of algorithm is **O(n****log***(n))**

Proof of Correctness:

In Function Merge, where we calculate number of inversions, we have two nondecreasing arrays so number of inversions in each individual array are 0, and there internal inversions are calculated previously when we formed these arrays using merge. So we just have to calculate number of inversions after merging.

Elements in Y(Second Array) will have higher index than that of X(First Array).
So for inversion Element(Y)<Element(X) and when this happens all elements left in First Array will be greater than that element from Y, here we get **(a-i)** inversions, where a is total elements in Array X initially and i is index of current smallest element in X.
Our assumption is correct that initially arrays are nondecreasing and there inversion are calculated previously and so Algorithm is correct.

Q2:

```
#1   Given list A of length n and k is also given
#2   d=ceil(n/k)
#3   lengthList
#4   For i from 0 to n-1 Do:
#5       B[floor(i/k)][i%k]=A[i]
#6   End For
#7   For j from 0 to d-1 Do:
#8       MergeSort(B[j],0,length(B[j])
#9       lengthList[j]=length(B[j])-1
#10  End For
#11  For m from 0 to k-1 Do:
#12      maxValue=B[m][0]
#13      indexForMax=0
```

```
#14         For p from 1 to d-1 Do:
#15             If length(B[p]) > 0 and maxValue < B[p][lengthList[p]] Then:
#16                 maxValue=B[p][lengthList[p]]
#17                 indexForMax=p
#18         maxList[m]=maxValue
#19         lengthList ( indexForMax )=lengthList ( indexForMax ) - 1
#20     End For
#21     printList maxList
```

Explanation:

#1 Given array A and k, which is number of largest integers to be found

#2 ceil(n) is smallest integer greater than or equal to n, i.e. ceil(2.5)=3

#3 lengthList is a list containing lengths of each sub array

#4 This loop makes **d** sub arrays

#7 This loop sorts all the sub-arrays in nondecreasing order and stores length of each in lengthList, MergeSort is same as discussed in class and as used in Q1
All the d sub-arrays are sorted

#11-20 now we have to compare largest element of each sub-array and find the first largest number, then we will delete that number from list and search for next highest number. This will repeat for k time. As each sub-array is sorted in nondecreasing order, last element of each sub-array will largest of that sub-array

Complexity:

Here complexity is because of for loops.

Line#4-6 n steps

Line#7-10 (n/k)*(c0*k*log(k)) = c0*n*log(k)

Line#11-20 (k+c1)*(d+c2) <= c3*k*n/k = c3*n

So total steps

n + c0*n*log(k) + c3*n

This means complexity is **O(n*log(k))**

Proof of correctness:

We divided array into d sub arrays, each sub array is sorted in nondecreasing order.
Now first of all we calculated max of all sub-arrays, that will largest of initial array.
Now similarly we can found 2nd highest, then third highest and so on.
Hence the output will be k largest elements.