Divide and Conquer Design Problems

Merge sort:

```
//idea: split the array into equal halves repeatedly until we are left with single element array.
//since single element array is sorted so we will combine two single element arrays in sorted
order. Then it will keep on combining the two sorted arrays into a single sorted array until all
the elements are not sorted.
//The technique is inspired from post order traversal where the left half of array is divided into
sub problems, the right half of the array elements are divided into sub problems and finally
both the sorted left and right halves are merged in sorted order.
//All arrays will start from index#1 in pseudo codes.
mergeSort(A[], s, e)
{
  if(s < e) //base case
  {
    M = (s+e)/2 //mid calculation
    mergeSort(A, s, M) //recursive call for left half,   T(N/2)
    mergeSort(A, M+1, e) //recursive call for right half,  T(N/2)
    merge(A, s, M, e) //recursive call to merge sorted left and right halves. O(N)
  }
}
merge(A[], st, M, e)
{
  //calculate the size of temporary left and right sub-arrays
  s1 = M - s + 1 //first half elements will be stored in left sub-array.
  s2 = e - M //2nd hald elements will be stored in right sub-array.

  //memory allocation for left and right sub-arrays
  left[1...s1]
  right[1...s2]

  //copy the data in left and right sub-arrays
  copy(A,left,s,s1) //it will copy the first half elements in left
  copy(A,right,M+1,s2) //it will copy the 2nd half elements in right

  //iterators for left, right, and original arrays
  i = 1 //iterator for left sub array
  j = 1 //iterator for right sub array
  k=s //iterator for original array

  //merge the arrays in sorted order
  while(i <= s1 && j <= s2)
  {
    if(left[i] <= right[j])
    {
      A[k++] = left[i++]
    }
    else
    {
      A[k++] = right[j++]
    }
  }
  //copy pending elements of left sub-array
  while(i <= s1)
  {
    A[k++] = left[i++]
  }
  //copy pending elements of right sub-array
  while(j <= s2)
  {
    A[k++] = right[j++]
  }

  delete left and right
}
copy(A[], temp[], start_index, size)
{
  j = 1
  for(i=start_index to i < start_index+size)
  {
    temp[j++] = A[i++]
  }
}

Recurrence:
T(N) = T(N/2) + T(N/2) + O(N)
Draw the recursion tree.
...
The cost at each level is N so it will become: N+N+N+...+N
    N* #of terms(or #of levels)
since the problem size is divided by 2 in each recursive call so total terms or
levels are logN (base 2).
overall complexity: N*logN
```

Inversion count:

```
//Inversion: pair of indices that are not in correct order i.e., (not in the ascending sorted
order).
count_Inversions(A[], s, e){
  inv_Count = 0  //local variable
  if(s < e) //base case
  {
     M = (s+e)/2 //mid calculation
     inv_Count += count_Inversions(A, s, M) //recursive call for left half
     inv_Count += count_Inversions(A, M+1, e) //recursive call for right half
     inv_Count += count(A, s, M, e) //recursive call to count the inversions in entire array.
  }
  return inv_Count
}
count(A[], st, M, e)
{
  inv_Count = 0

  //calculate the size of temporary left and right sub-arrays
  s1 = M - s + 1 //first half elements will be stored in left sub-array.
  s2 = e - M //2nd hald elements will be stored in right sub-array.

  //memory allocation for left and right sub-arrays
  left[1...s1]
  right[1...s2]

  //copy the data in left and right sub-arrays
  copy(A,left,s,s1) //it will copy the first half elements in left
  copy(A,right,M+1,s2) //it will copy the 2nd half elements in right

  //iterators for left, right, and original arrays
  i = 1 //iterator for left sub array
  j = 1 //iterator for right sub array
  k=s //iterator for original array

  //merge the arrays in sorted order
  while(i <= s1 && j <= s2)
  {
     if(left[i] <= right[j])
     {
        A[k++] = left[i++]
     }
     else
     {
        A[k++] = right[j++]
        inv_Count += s1-i+1
//elements in left sub-array from index "i" till the last element have inversions with index j
//since "j" will be updated so we need to count all the inversions.
     }
  }
  //copy pending elements of left sub-array
  while(i <= s1)
  {
     A[k++] = left[i++]
  }
  //copy pending elements of right sub-array
  while(j <= s2)
  {
     A[k++] = right[j++]
  }

  delete left and right
  return inv_Count
}
copy(A[], temp[], start_index, size)
{
  j = 1
  for(i=start_index to i < start_index+size)
  {
     temp[j++] = A[i++]
  }
}

//Recurrence: T(N) = T(N/2) + T(N/2) + O(N)
//overall time complexity: O(N*logN)
```

Compatibility count:

```
//All arrays will start from index#1 (pseudo code)
compCount(A[], s, e)
{
   cm_Count = 0
   if(s < e) //base case
   {
      M = (s+e)/2 //mid calculation
      cm_Count += compCount(A, s, M) //recursive call for left half
      cm_Count += compCount(A, M+1, e) //recursive call for right half
      cm_Count += count(A, s, M, e)
//recursive call to count the compatible elements in the entire array sorted left and right.
   }
   return cm_Count
}
count(A[], st, M, e){
   cm_Count = 0

   //calculate the size of temporary left and right sub-arrays
   s1 = M - s + 1 //first half elements will be stored in left sub-array.
   s2 = e - M //2nd hald elements will be stored in right sub-array.

   //memory allocation for left and right sub-arrays
   left[1...s1]
   right[1...s2]
//copy the data in left and right sub-arrays
   copy(A,left,s,s1) //it will copy the first half elements in left
   copy(A,right,M+1,s2) //it will copy the 2nd half elements in right

   //iterators for left, right, and original arrays
   i = 1 //iterator for left sub array
   j = 1 //iterator for right sub array
   k=s //iterator for original array

   //merge the arrays in sorted order
   while(i <= s1 && j <= s2)
   {
      if(left[i] <= right[j])
      {
        A[k++] = left[i++]
        cm_Count += s2-j+1
//if "i" is compatible with "j" then it is understood that it is compatible with all the elements
//in the right sub-array after j. so count all the compatibilities.
      }
      else
      {
        A[k++] = right[j++]
      }
   }
   //copy pending elements of left sub-array
   while(i <= s1)
   {
      A[k++] = left[i++]
   }
   //copy pending elements of right sub-array
   while(j <= s2)
   {
      A[k++] = right[j++]
   }

   delete left and right
   return cm_Count
}
copy(A[], temp[], start_index, size){
   j = 1
   for(i=start_index to i < start_index+size)
   {
      temp[j++] = A[i++]
   }
}
//Recurrence: T(N) = T(N/2) + T(N/2) + O(N)
//overall time complexity: O(N*logN)
```

Quick sort:

```
//The technique is inspired from pre-order traversal.
QuickSort(A[], s, e)
{
  if(s < e)
  {
    p_Index = partition(A, s, e) //place the pivot at its actual position
    QuickSort(A, s, p_Index-1) //recursive call for elements on the left side of pivot
    QuickSort(A, p_Index+1, e) //recursive call for elements on the right side of pivot
  }
}

partition(A[], s, e)
{
//pivot could be any element of array. you can randomly select any index in the given range
//swap the element at that index with the last index.
  pivot = e, i = s, j = e-1
  while(i < j)
  {
    while(i < e && A[i] < A[pivot])
    {
      i++   //skip all the elements smaller than pivot
    }
    while(j > s && A[j] > A[pivot])
    {
      j--  //skip all the elements greater than pivot
    }
    if(i < j)
    {
      swap(A[i], A[j])  //swap index i and j and update both indices
      i++
      j--
  }
  if(A[i] > A[pivot])
  {
    swap(A[i], A[pivot])
  }
  return i   //return pivot index
}
```

Recurrence:
Derive the recurrence and solve?

Maximum subarray sum:

Maximum (Contiguous) SubArray Sum Problem
Time complexity of Brute force technique = O(N^2)
Kadane's Algorithm time complexity for this task: O(N*logN)
Home Task: Is it possible to solve the problem in linear time?
=>If yes then design an algorithm that should solve this problem in linear time.
=>If it is not then make it possible.

Main Idea of Kadane's Algorithm:
We are intrested to determine maximum subArray (starting point, ending point and the sum of elements in the max subArray)
Use divide and conquer technique just like merge sort and divide the array into 2 halves.
Possibility of following three cases:
1: maximum subArray will be entirely in the left half
2: maximum subArray will be entirely in the right half
3: maximum subArray will be in the cross-sectional region of left and right half.

So we need to calculate the sum of (left, right and the cross-sectional region) and the Maximum of (left, right and the cross-sectional region) will be the maximum subArray.

There are two recursive calls with input size (n/2) so one part of recurrence will be 2T(n/2). The function maxCross sectional will take linear time to calculate the sum of cross-sectional region. So the complete recurrence will be

T(N) = 1 n = 1
2T(n/2) + O(N) n > 1

// We need following information for left, right and cross-sectional region:
1: left boundary (starting point of subArray)
2: right boundary (ending point of subArray)
3: sum of the elements in subArray

// A[] = Array // L = starting index of array, // R = last index of array, // M = mid index, // L_st = starting point left subArray, // L_end = ending point of left subArray
// L_sum = sum of the elements in the left subArray // R_st = starting point left subArray // R_end = ending point of left subArray // R_sum = sum of the elements in the left subArray
// Cr_st = starting point of cross-sectional subArray, Cr_end = ending point of cross-sectional subArray, Cr_sum = sum of elements in the cross-sectional subArray

```
maxSubArray(A[], L, R)
{
  if (L >= R)
          return (L,R,A[L]);  //(L: left boundary, R: right boundary and A[L]: the sum of
subArray. Since it is a single unit array so sum of the sub array will be equal to A[L])
  mid = (L+R)/2
  (L_st, L_end, L_sum) = maxSubArray(A,L,M); //recursive call for left half. Eventually it will
return the information for left half when base case will be encountered
```

```
  (R_st, R_end, R_sum) = maxSubArray(A,M+1,R); //recursive call for right half. Eventually it will return the information for right half when base case will be encountered
  (Cr_st, Cr_end, Cr_sum) = maxCross(A,L,M,R); //maxCross will return the information for left cross-sectional region

//All we need to compare is just three values i.e.,(L_sum, R_sum and Cr_Sum)
  if (L_sum > R_sum && R_sum > Cr_sum)
      return (L_st, L_end, L_sum)   //returning the information of maximum subArray
  else if(R_sum > Cr_Sum)
      return (R_st, R_end, R_sum)
  else
      return (Cr_st, Cr_end, Cr_sum)
}
```

The starting point of maximum cross-sectional part will be in the left subArray and ending point will be in the right subArray and what will be the sum of elements in cross-sectional part?

We need 4 main variables:

1: max_Left: starting point of cross-sectional part in the left subArray
2: max_Right: ending point of cross-sectional part in the right subArray
3: L_sum: maximum sum of left subArray
4: R_sum: maximum sum of right subArray

```
maxCross(A[], L, M, R)
{
  L_sum = INT_MIN (-ve infinity)
  max_Left    //(to keep the starting index)
  sum = 0

//basically we need the sum of only those (contiguous) elements of left-subArray providing us the maximum result.
//idea: start from mid and keep including the elements of left-part in L_sum if the result is increasing.
  for(i= mid downto L)
  {
      sum = sum + A[i]
      if(L_sum < sum)     //we need to update the value of L_sum and max_left when this condition is true
            {
                    L_sum = sum    // it will update the value L_sum
        max_Left = i   //it will shift the max_Left to the starting point
            }
  }

//just like left-part we need the sum of only those (contiguous) elements of right-subArray providing us the maximum result.
//right part: start from mid+1 and keep including the elements of right-part in R_sum if the result is increasing.
  R_sum = INT_MIN (-ve infinity)
  max_Right  //(to keep the last index)
  sum = 0    //reset sum variable

  for(i= mid+1 to R)
  {
      sum = sum + A[i]
      if(R_sum < sum)
      {
//we need to update the value of R_sum and max_Right when this condition is true
           R_sum = sum      //it will update the value of R_sum
           Max_Right = i   //it will shift the max_Right to the ending point
       }
  }

//Max_left is pointing to the starting point of left subArray (we can say "starting position of
cross-sectional part")
//Max_right is pointing to the ending point of right subArray (we can say "ending position of
cross-sectional part")
//if we add (L_sum and R_sum) then it will provide you the sum of cross-sectional region

   return (max_Left, max_Right, L_Sum+R_Sum)

//Time complexity analysis of maxCross: two loop from (mid downto L) and (mid+1 to R).
Since the sum of L+R will be equal to total size i.e., 'N' so linear time
}
Recurrence: T(N) = T(N/2) + T(N/2) + O(N)
Overall time complexity: O(N*logN)
```