Merge Sort : Divide and Conquer

input : an array of unsorted elements

output : an array of sorted elements

small problem : If an array contains only a single element in an array then it is itself a sorted array and that we consider as the small problem.

50 12 23 14 89 90 27 #values

1 2 3 4 5 6 7 #index

a[1:7]c1 - (12,14,23,27,50,89,90)

a[1:4] c2 **a[5:7] c3**

c2-(12,14,23,50) **c3-(27,89,90)**

a[1:2]c4 **a[3:4] c5** **a[5:6]c6** **a[7:7]c7**

c4-(12,50) **c5-(14,23)** **c6-(89,90)** **c7-(27)**

a[1:1]c8 a[2:2]c9 a[3:3]c10 a[4:4]c11 a[5:5]c12 a[6:6]c13

c8-(50) c9-(12) c10-(23) c11-(14) c12-(89) c13-(90)

Function call -> Preorder

Function execute -> Postorder

Merge Procedure :

Worst case number of comparisons in Merge Procedure :

(10 20 30 40) (11 21 31 41)

1 2 3 4 5 6 7 8

10 11 20 21 30 31 40 41
1 2 3 4 5 6 7 8

Number of comparisons : 7

(10,11) = 10 1st time

(20,11) = 11 2nd time

(20,21) = 20 3rd time

(30,21) = 21 4th time

(30,31) = 30 5th time

(40,31) = 31 6th time

(40,41) = 40 7th time

41 = 41 No comparison

m + n - 1

m = number of elements in sorted subarray

1 n = number of elements in sorted subarray

2 m = 4 , n = 4

4 + 4 - 1 = 7

Best case number of comparison in Merge Procedure

(10 20 30 40 50 60) (5 6)

1 2 3 4 5 6 7 8

5 6 10 20 30 40 50 60

1 2 3 4 5 6 7 8

(10,5) = 5 1st comparison
(10,6) = 6 2nd comparison

General formula for best case scenario of merge procedure

: min(m,n)

m = number of elements in sorted subarray 1

n = number of elements in sorted subarray 2

Overall Time Complexity of Merge Procedure :

Number of moves in best and in worst case scenario = m + n

Time complexity = Number of moves + Number of comparisons = O(m + n)

Implementation :

```
def merge(arr, l, m, r):
        // Find sizes of two subarrays to be merged
        n1 = m - l + 1
        n2 = r - m
        // Create temp arrays
        array1 = []
        array2=[]
        // Copy data to temp arrays
```

```
for i in range(n1):
        array.append(arr[l+i])
forin range(n2):
            arrary2.append(arr[m+1+j])
 // Initial indexes of first and second subarrays
i = 0, j = 0
 // Initial index of merged subarray array
k = l
while(i<n1 && j<n2):
        if array1[i]<=array[j]:
            arr[k]=array1[i]:
            i++
        else:
            arr[k]=array2[j]
            j++
        k++
 // Copy remaining elements of array1[] if any
 while (i < n1):
        arr[k] = array1[i]
        i++
        k++
 // Copy remaining elements of array2[] if any
while (j < n2):
        arr[k] = array2[j]
        j++
        k++
```

Note : MergeSort is an outplace sorting algorithm because here we are using a new array to store the elements after doing comparisons.

Mergesort Algorithm :

```
MergeSort(arr,i,j):     #arr array , i -starting index, j - ending index

        // small problem

        if(i == j):

                return arr[i]

        // big problem

        else:

                mid = (i + j)/2                          // Divide              O(1)

                                                         // Conquer

                MergeSort(arr,i,mid);                    // Left side tree      T(n/2)

                MergeSort(arr,mid+1,j);                  // Right side tree     T(n/2)
                MergeProcedure(arr,i,mid,mid+1,j);       // combine             O(n)
```

Overall Time Complexity :

Best, average and worst case scenario

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

= O(n logn)