ASSIGNMENT-3

Quick Sorting and Merge Sorting Techniques

1. Quick Sort

ALGORITHM:

Step 1: Start

Step 2: User input a array to be sorted say A

Step 3: Initialize integer variables p and q at the start and end of the array

Step 4: Define a function partition with arguments containing A,l,h

. if p<q continue to the next

. pivot=A[l] , p=l , q=h

. p=p+1 repeat until A[p]<=pivot

. q=q-1 , if p<q swap A[p] and A[q] → Repeat these two steps until A[q]>pivot

. else swap A[l] and A[q]

. return q

Step 5: Define a function Quicksort with arguments containing A,l,h

. if l<h continue

. Q=partition(A,l,h)

. Quicksort(A,l,Q-1) → function call

. Quicksort(A,Q+1,h)

Step 6: Display the sorted array

Step 7: Stop

THEORETICAL TIME COMPLEXITY:

WORST CASE: O(n^2)

AVERAGE CASE: O(nlogn)

BEST CASE: O(nlogn)

ANALYSIS:

General time taken by quick sort: T(k) + T(n-k-1) + O(n)

In worst case: T(0) + T(n-1) + O(n)

= O(n^2 - n)

= O(n^2)

In worst case: T(n/9) +T(10n/9) +O(n) [when one set containing n/9 and other 9n/10 elements]

= O(nlogn)

In best case: 2T(n/2) + O(n) [when pivot element is taken to be at the middle]

= O(nlogn)

2. Merge Sort

ALGORITHM:

Step 1: start

Step 2: Ask the user to i/p the array to be sorted.

Step 3: Initialize two int variables i,j at the starting and ending indices of array, int mid=floor((i+j)/2).

Step 4: Define a function that partitions array into 2 halves, with arguments lower bound, upper bound, array.

- Call this function repeatedly until we get trivial elements.
- Partition (i, mid, array)
- Partition (mid+1, j, array)
- Partition (i, (i+mid)/2, array)
- Partition((i+mid)/2)+1,j,array) and so on until we get single element.

Step 5: After partitioning is done, define another function merge that takes the output (sub-array) of partition function as its input argument.

- Merge(sub-array)

  {

  - Compare the sub-array's first elements.
  - the smallest element and put it into the result array.
  - Continue the process until all elements have been put into the result. Array

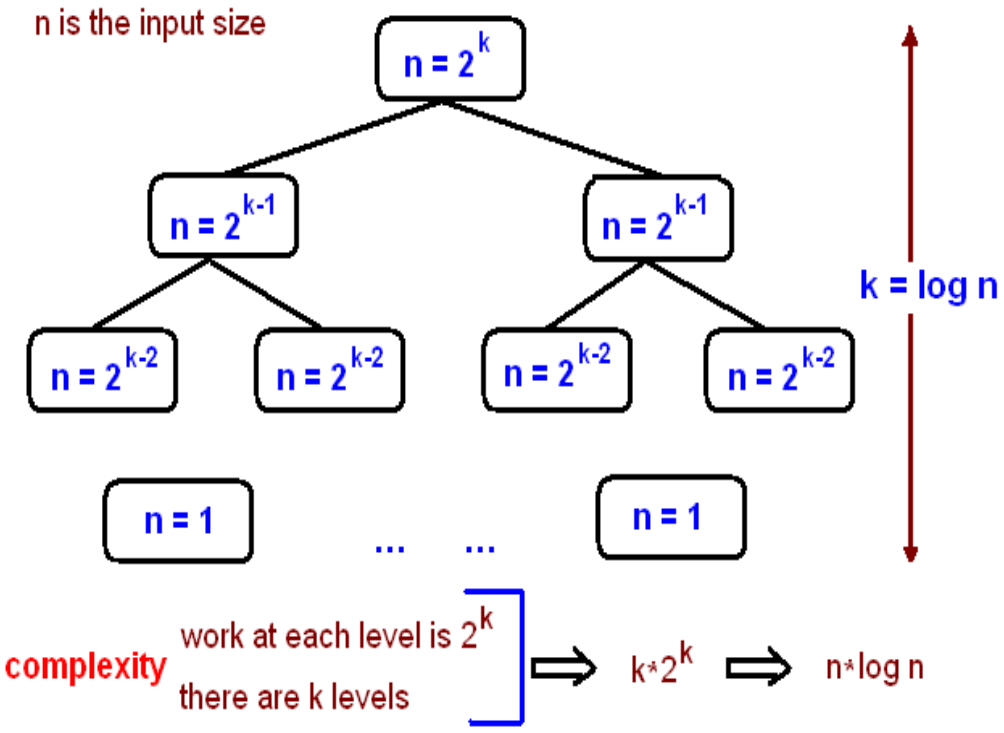
  }

Step 6: Display result array.

Step 7: END

THEORITICAL TIME COMPLEXITIES:

- BEST CASE: O(nlogn)
- AVERAGE CASE: O(nlogn)
- WORST CASE: O(nlogn)



OBSERVATIONS

TECHNIQUE	INPUTS	TIME TAKEN
Quick Sort(Ascending)	1.[2.5,4.5,3.0,1.2,6.5,8.9,7.4,6.3] 2.[5,3,6,3,4,5,4,6,4]	1. 0.032s 2. 0.032s
Quick Sort(Descending)	1. [2.5,4.5,3.0,1.2,6.5,8.9,7.4,6.3] 2. [5,3,6,3,4,5,4,6,4]	1. 0.022s 2. 0.027s
Merge Sort(Ascending)	1. [2.5,4.5,3.0,1.2,6.5,8.9,7.4,6.3] 2. [5,3,6,3,4,5,4,6,4]	1. 0.027s 2. 0.024s
Merge Sort(Descending)	1. [2.5,4.5,3.0,1.2,6.5,8.9,7.4,6.3] 2. [5,3,6,3,4,5,4,6,4]	1. 0.029s 2. 0.033s

<>FOR QUICK SORTING, THE FIRST ELEMENT IS TAKEN AS PIVOT ELEMENT.

CONCLUSION

In quick sorting, when the pivot element is the smallest or largest in the array, it is observed to be the worst case scenario.

In merge sorting, the time complexity is same in all cases because merge sort divides the array in two halves and takes linear times to merge two halves.