Experiment No 2

Aim: To study and implement Merge and Quick Sort

Theory

A) Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm. Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution. Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem. Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide: If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer: In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine: When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

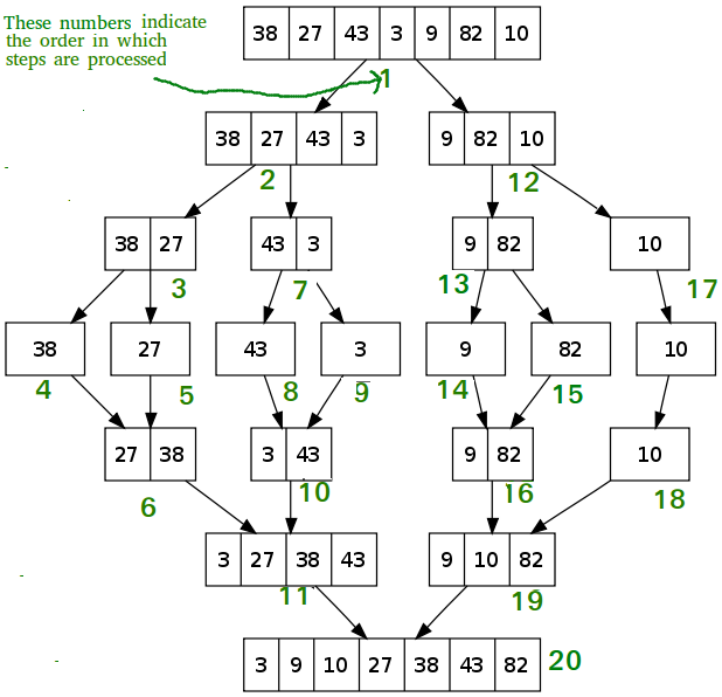
MergeSort Algorithm

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r. After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

Algorithm for Merge Sort:

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

For Example:



B) Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

In Quicksort ,

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.

2. Always pick last element as pivot (implemented below)

3. Pick a random element as pivot.

4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down

the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Algorithm of Quick Sort

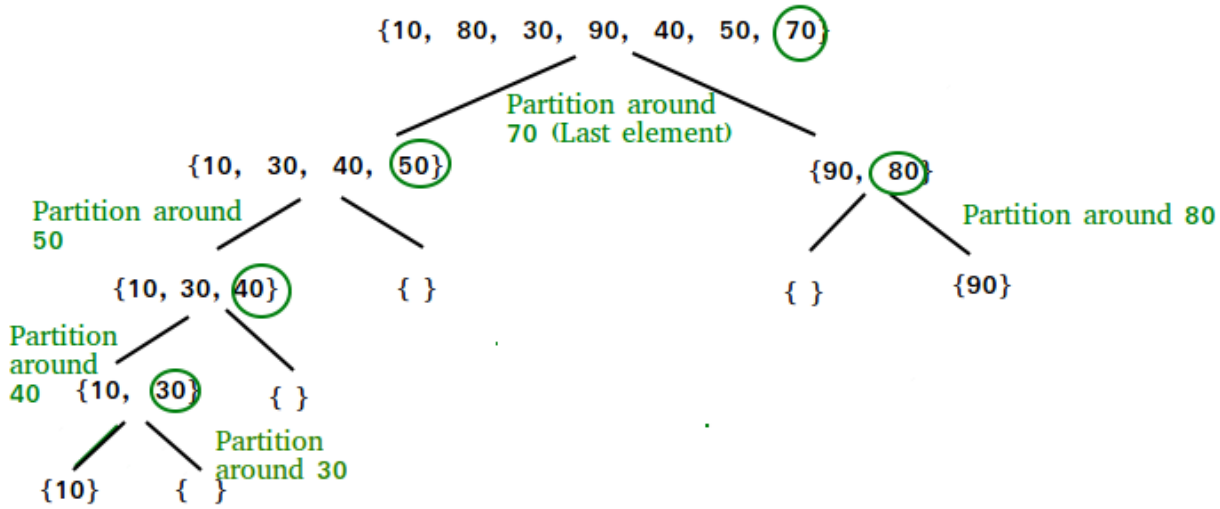
Quick Sort Algorithm:

```
1.  QUICKSORT (array A, start, end)
2.  {
3.   1 if (start < end)
4.   2 {
5.  3 p = partition(A, start, end)
6.  4 QUICKSORT (A, start, p - 1)
7.  5 QUICKSORT (A, p + 1, end)
8.  6 }
9.  }
```

Partition Algorithm

```
1.  PARTITION (array A, start, end)
2.  {
3.   1 pivot ? A[end]
4.   2 i ? start-1
5.   3 for j ? start to end -1 {
6.   4 do if (A[j] < pivot) {
7.   5 then i ? i + 1
8.   6 swap A[i] with A[j]
9.   7  }}
10.  8 swap A[i+1] with A[end]
11.  9 return i+1
12.  }
```

For Example:



Analysis of Merge and Quick Sort complexities

a) Following is the Time and Space Complexity of Merge Sort:

Average Time Complexity: O(N logN)

Worst Case Time Complexity: O(N logN)

Best Case Time Complexity: O(N)

Space Complexity: O(N); O(1) with Linked Lists

b) Following is the Time and Space Complexity of Quick Sort:

Average Time Complexity: O(N logN)

Worst Case Time Complexity: O(N^2); O(N logN) with median of medians

Best Case Time Complexity: O(N logN)

Space Complexity: O(N); O(logN) with Hoare's original version

Conclusion: Thus, we have successfully studied and implemented Merge and Quick Sort Algorithms.