Session 3

Merge Sort and Quick Sort

Merge Sort

The **Merge Sort** algorithm closely follows the **Divide and Conquer** paradigm (pattern). The divide and conquer approach involves three main steps:

Divide: Here we divide a problem into a no. of sub-problems having smaller instances of the same problem.

Conquer: Then we conquer the sub-problems by solving them recursively.

Combine: And in last, we Combines the solutions of the sub-problems into the solution for the original problem.

Algorithm

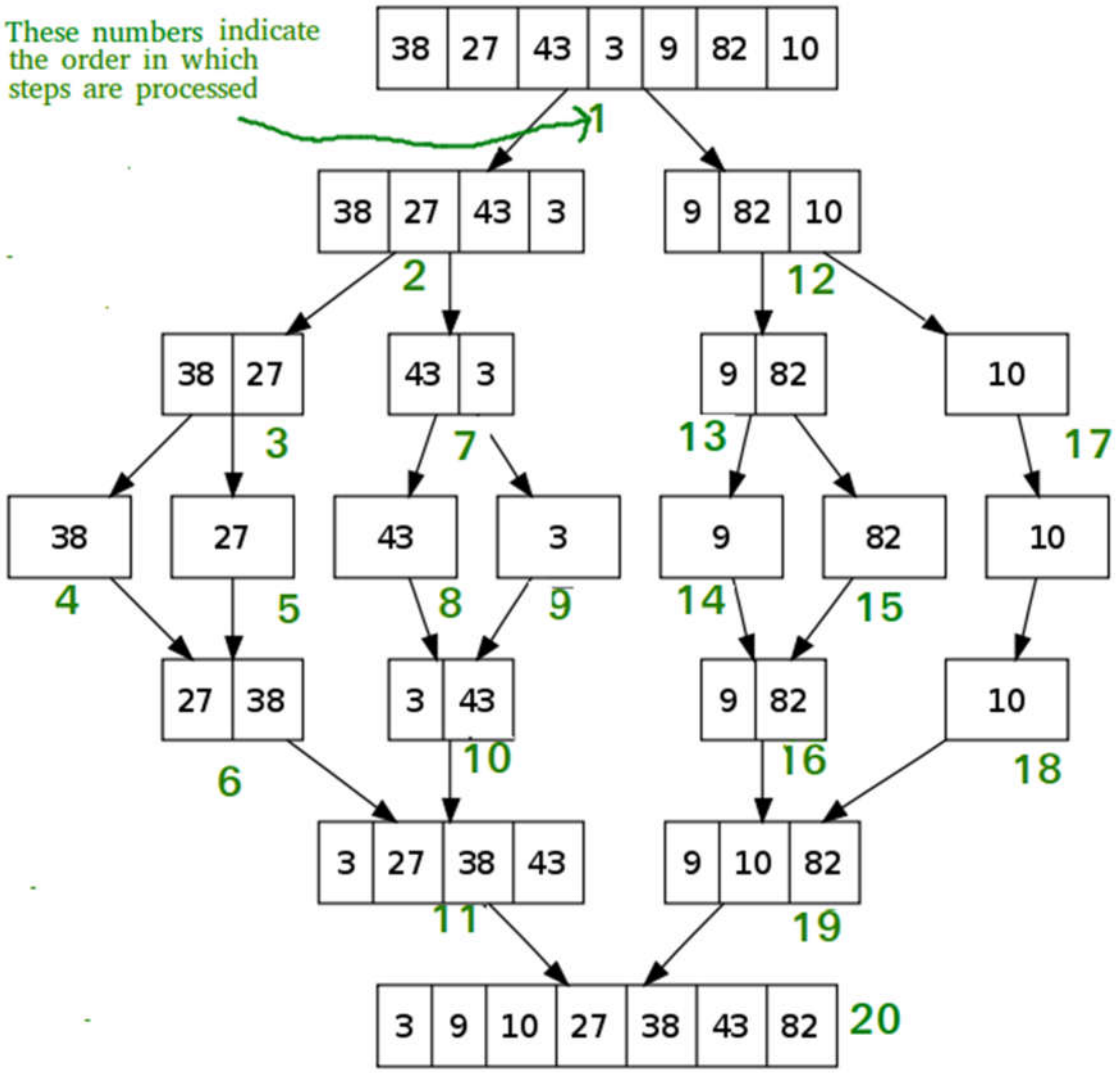
```
MRGE-SORT(A, p, r)

1. If p < r
2. q = ( p + r ) /2
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q+1, r)
5. MERGE(A, p, q, r)
```

```
MERGE (A, p, q, r)

1.  n1 = q – p +1
2.  n2 = r – q
3.  let L [1.. n1 + 1 ] and R [1.. n2 + 1 ]  be new arrays
4.  for i=1 to  n1
5.        L[i ]  =  A [ p + i -1]
6.  for j=1 to n2
7.        R[ j ]= A[ q + j ]
8.  L [n1 + 1 ] =  ∞
9.  R [n2 + 1 ] =  ∞
10. i = 1
11. j = 1
12. for k = p to r
13.       if L[i ] ≤R [ j ]
14.            A[ k ] = L[ i ]
15.            i = i + 1
16.       else
17.            A[ k ] = R [ j ]
18.            j = j + 1
```

Example



QuickSort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.

1. Always pick first element as pivot (implemented below).
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quick sort 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.

Algorithm

```
Quicksort(f, l)
1.  if f < l then
2.      i = f + 1
3.      j = l
4.      while x[i] < x[f] do
5.          i = i + 1
6.      while x[j] > x[f] do
7.          j = j – 1
8.      while i < j do
9.          swap x[i] with x[j]
10.         while x[i] < x[f] do
11.             i = i + 1
12.         while x[j] > x[f] do
13.             j = j – 1
14.     swap x[j] with x[f]
15.     Quicksort(f, j-1)
16.     Quicksort(j+1, l)
```

Example

arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6 low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, **i = -1**
Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same
j = 1 : Since arr[j] > pivot, do nothing // No change in i and arr[]
j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30
j = 3 : Since arr[j] > pivot, do nothing// No change in i and arr[]
j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped
We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped
Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Practice Problems

1. Implementation of the "Merge & Quick Sort" Algorithm to create a program in C/CPP or JAVA. Also demonstrate a comparison table (Data movement and Data Comparison) of 5 different Sorting Algorithms.
2. Take an array and sort it using merge sort. Use modulo value of each item as comparison value.
3. Find median of an array using quicksort. You cannot sort the entire array. Additionally, if it is evident that median is in a particular half of the array, you must stop sorting the other half of the array.