Design and Analysis of Algorithms

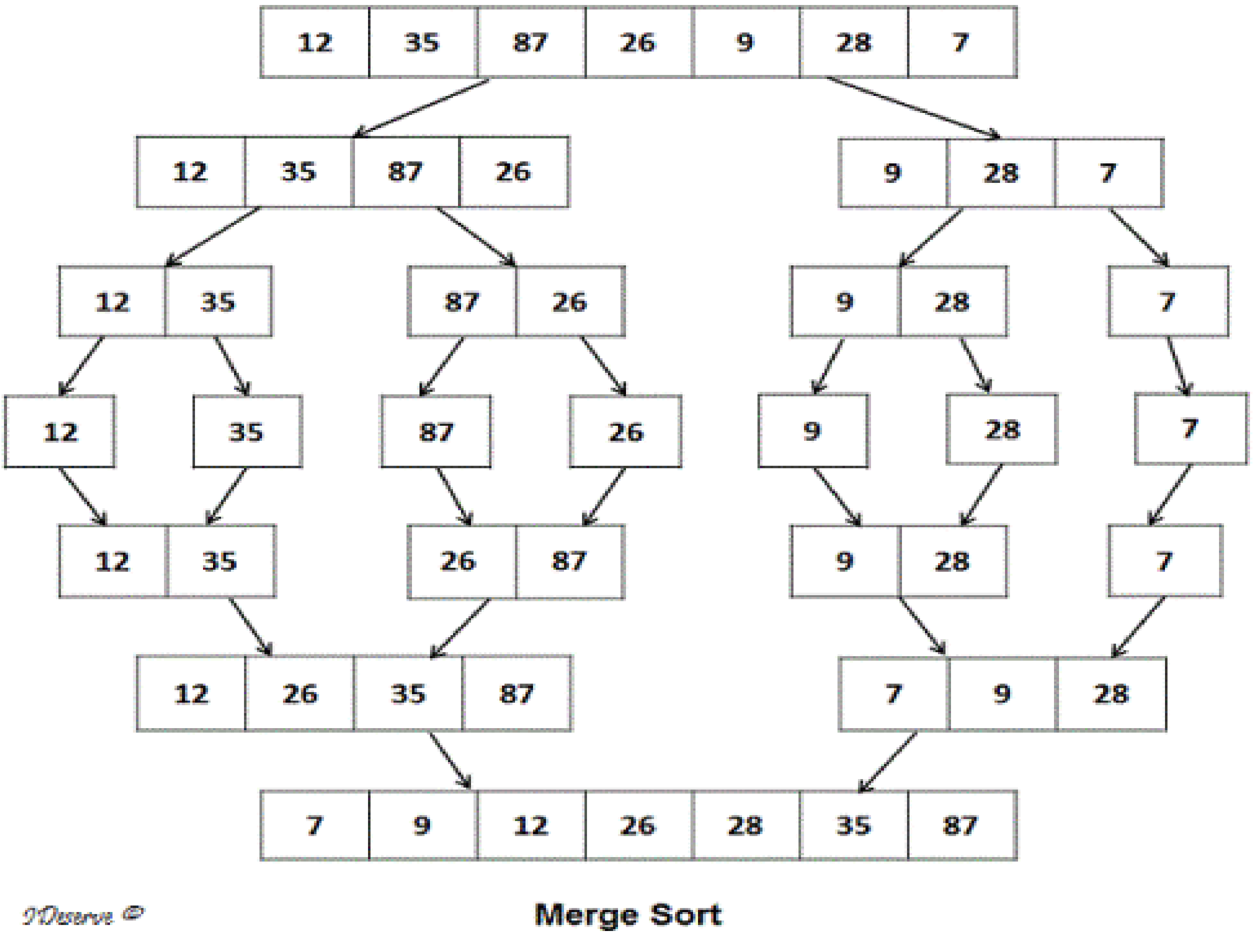
Module 4

Merge Sort

Merge Sort

- The merge sort algorithm based on the divide-and-conquer paradigm.
- It operates as follows.
  - Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
  - Conquer: Sort the two subsequences recursively using merge sort.
  - Combine: Merge the two sorted subsequences to produce the sorted answer.
- The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step.

Example



Merge Sort

Merge Sort Algorithm

- We merge by calling an auxiliary procedure MergeSort(low,high), where a is an array and low, high, and mid are indices into the array such that low $\leq$ mid $<$ high.
- The procedure assumes that the subarrays a[low .. mid] and a[mid + 1..high] are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray a[low .. high].

Merge Sort Algorithm

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Merge Sort Algorithm cont...

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

low mid mid+1 high
h j
i

Analyzing Merge Sort

- Assume that *n* is a power of 2 so that each divide step yields two subproblems, both of size exactly *n*/2.
- The base case occurs when *n* = 1.
- When $n \geq 2$, time for merge sort steps:
  - **Divide**: Just compute *mid* as the average of high and low, which takes constant time i.e. $\Theta(1)$.
  - **Conquer**: Recursively solve 2 subproblems, each of size *n*/2, which is $2T(n/2)$.
  - **Combine**: MERGE on an *n*-element subarray takes $\Theta(n)$ time.
- The recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

- **Solving the Merge Sort Recurrence**
  - By the master theorem,
  - a=2, b=2, f(n)=n, $n^{\log_b a} = n^{\log_2 2} = n^1 = f(n)$
  - Case 2 satisfied, therefore solution is

    $T(n) = \Theta(n \lg n)$.

Previous Year Questions

- Write an algorithm to merge two sorted arrays and analyse the complexity. (3)
- Write and explain merge sort algorithm using divide and conquer strategy. Also analyse the complexity. (4)
- Explain 2-way merge sort with an example. (6)