CSEN3001:
DESIGN AND ANALYSIS OF ALGORITHMS

UNIT-I: INTRODUCTION TO ALGORITHMS
Merge Sort

Divide and conquer – general idea

- **Divide a problem into subprograms of the same kind**
- **Solve subprograms using the same approach**
- **Combine partial solution (if necessary)**

Merge sort : 2 Way merging

Merging Two sorted lists:

Array A :
m =6 elements

Index	1	2	3	4	5	6	7	8	9
Element	7	9	11	21	23	44			

Array B :
n= 9 elements

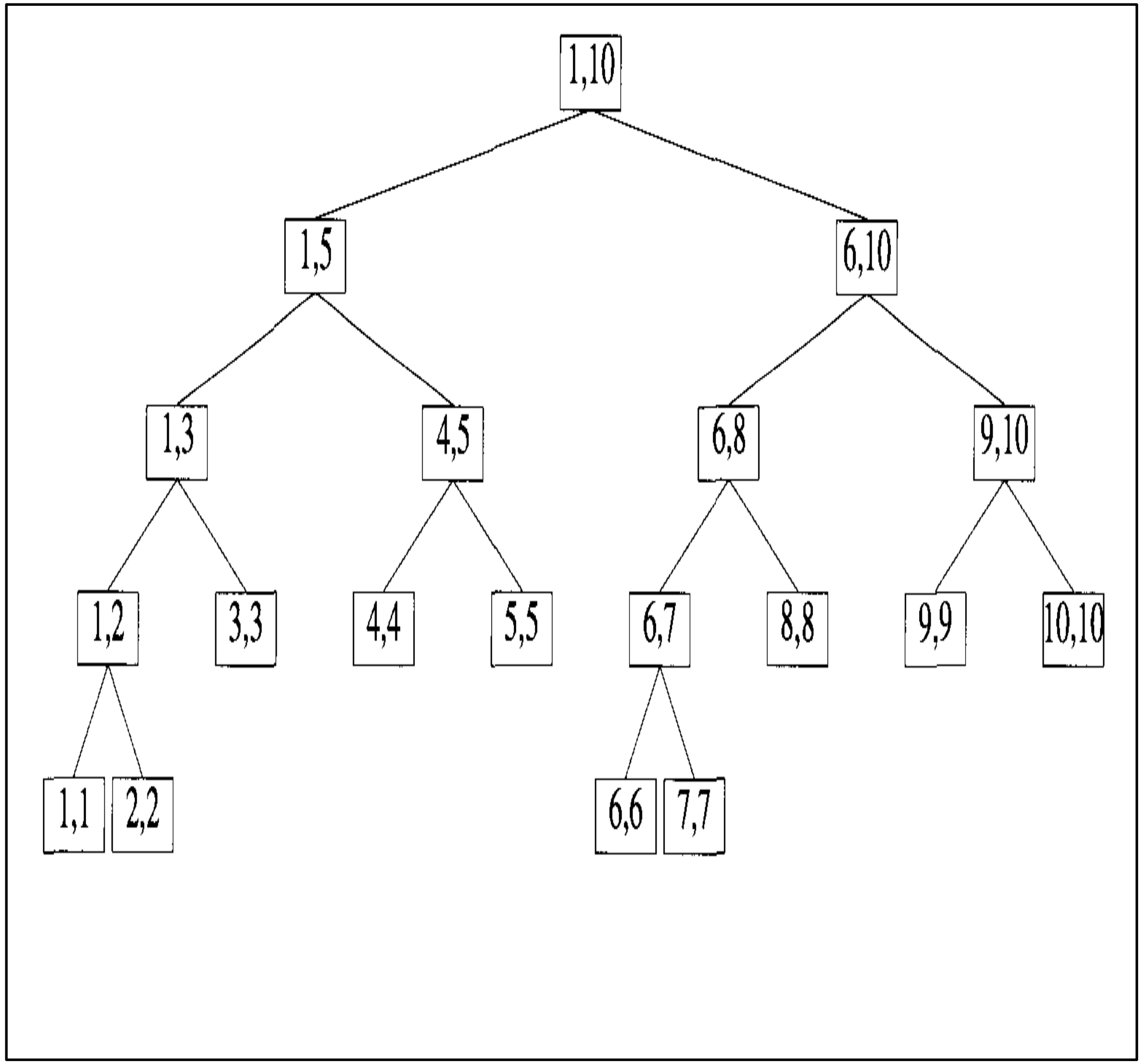
Index	1	2	3	4	5	6	7	8	9
Element	4	6	10	19	22	32	77	79	81

Array C:
p= m+n=6+9=15
elements

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Element	4	6	7	9	10	11	19	21	22	23	32	44	77	79	81	

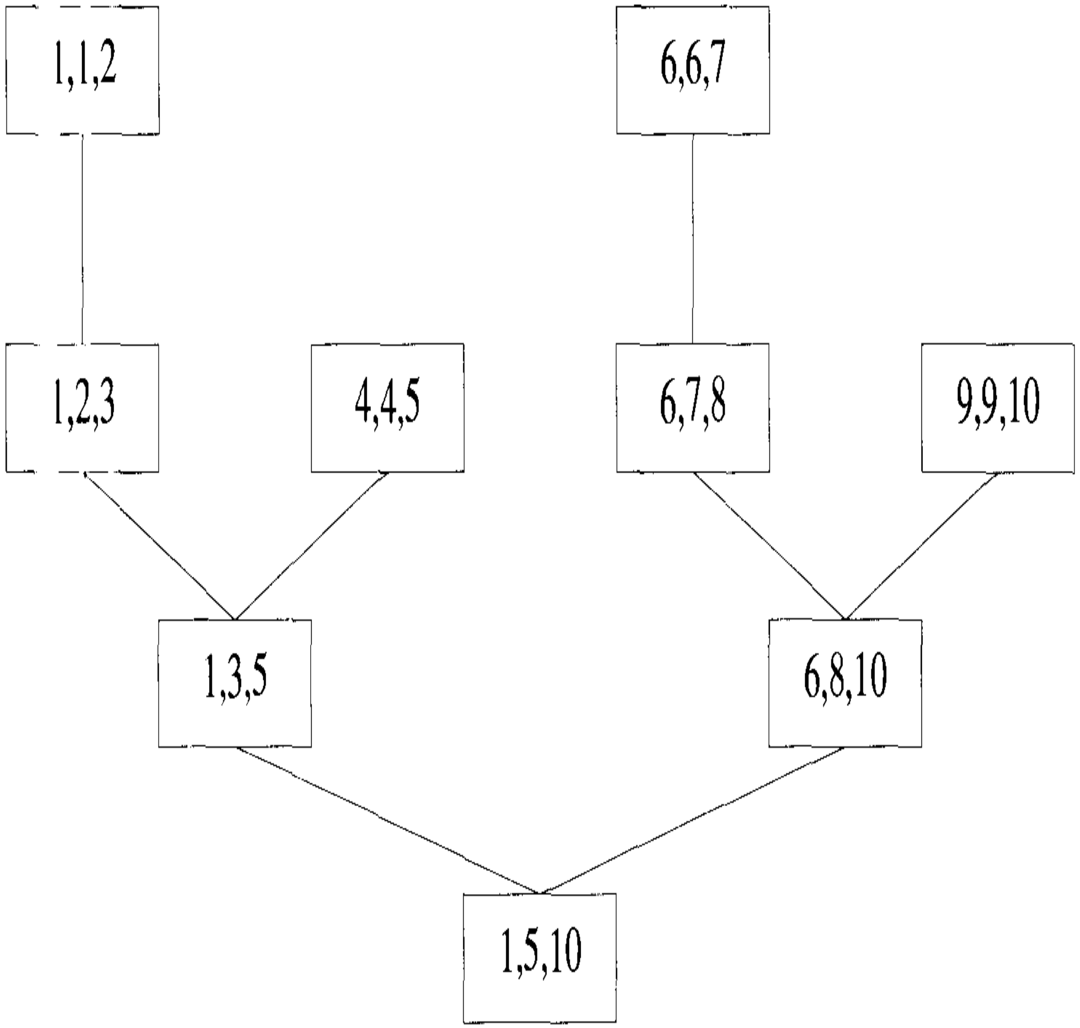
Merge sort : 2 Way merging

```
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.
{           //there are more than one element
        if (low< high) then
        {    // Divide P into sub problems.Find where to
                split the set.
                mid:=(low + high)/2;
                // Solve the sub-problems.
                MergeSort(low, mid);
                MergeSort(mid+ 1, high);
                // Combine the solutions.
                Merge(low, mid, high);
        } // end of if
}// end of MergeSort
```



Merge sort : 2 Way merging

```
Algorithm Merge(low, mid, high)

// a[low, high] is a global array containing two sorted subsets in a[low :
//mid] and 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++;}

                        else {b[i]:=a[j]; j++;}

                        i++;

            } // end of while

            if  (h>mid) then for k:= j to high do { b[i]:=a[k]; i++}

            else for k:= h to mid do { b[i] := a[k]; i++}

            for k:= low to high do a[k]:= b[k]

}
```



Space requirement of Merge Sort

- Additional n locations are required to merge the two sorted portions of max size n/2 and n/2 i.e., in the last phase of exiting from the recursion.
- Every time merged elements from the auxiliary array should be copied back to the original array.

- Stable Sorting: A sorting method is said to be stable if at the end of the method, identical elements occur in the same order as in the original unsorted set.

- Is Merge Sort a Stable Sorting Method?
- Is Bubble Sort a Stable Sorting Method?

Time Complexity

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

Algorithm: Mergesort with Internal Insertion Sort

```
Algorithm MergeSort1(low, high)
// The global array a[low : high]is sorted in non-decreasing order
//using the auxiliary array link[low : high]. The values in the link
//represent a list of the indices low through high, giving a[ ] in sorted
//order. A pointer to the beginning of the list is returned.
{  if {(high-low) < 15)then
        return InsertionSort(a, link, low, high);
   else
     {mid := (low+high)/2;
      q := MergeSort1(low, mid);
      r := MergeSort1(mid+1, high);
      return Mergel(q, r);
      }
   }
```

Insertion Sort:
// 1st element is already in sorted order
for j :=2 to n do {
Place a[j] in its correct position in the sorted set a[l:j-1];
}

Algorithm: Insertion Sort

```
Algorithm InsertionSort(a, n)
// Sort the array a[1 : n] into non-decreasing order, n >= 1.
{   for j := 2 to n do
      { // a[1: j-1] is already sorted.
          item := a[j]; i := j -1;
          while ((i >= 1) and (item<a[i]))do
           { a[i+ 1]:=a[i]; i :=i - 1;
            }
          a[i+ 1] := item;
        } //end of for
  }  // end of algorithm
```

The statements within the while loop can be executed zero up to a maximum of j times. Since j goes from 2 to n, the worst-case time of this procedure is bounded by

$$\sum_{2\leq j\leq n} j = n(n+1)/2 - 1 = \Theta(n^2)$$

Its best-case computing time is 0(n) under the assumption that the body of the while loop is never entered. This will be true when the data is already in sorted order

THANK YOU