Welcome To My Presentation

My Presentation Topic is

Merge Sort

Presented By

Tushar Sarkar

Student ID : 18CSE35

Second Year Second Semester

Department of CSE, BSMRSTU.

Content

- Introduction
- Divide and Conquer Strategy
- Divide, Conquer & Combine
- Merge Sort Algorithm
- Example Explain
- Pseudocode
- Time Complexity
- Conclusion

Introduction

- 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.

Divide and Conquer Strategy

- Using the **Divide and Conquer** technique, we divide a problem into sub-problems.
- When the solution to each sub-problem is ready, we 'combine' the results from the sub-problems to solve the main problem.
- Suppose we had to sort an array $\boxed{A}$.
- A sub-problem would be to sort a sub-section of this array starting at index $\boxed{p}$ and ending at index $\boxed{r}$, denoted as $\boxed{A[p...r]}$.

Divide, Conquer & Combine

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.

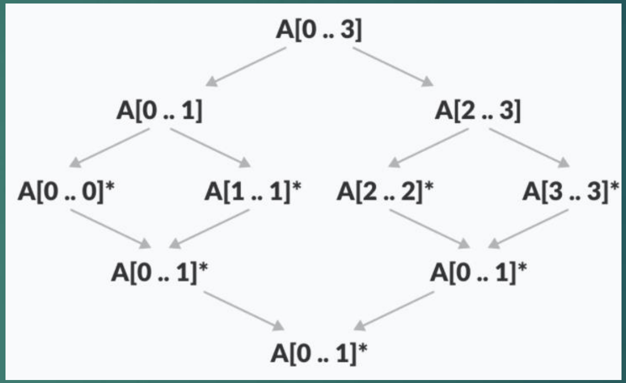
Combine

- 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].

Merge Sort 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.

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```



- To sort an entire array, we need to call MergeSort(A, 0, length(A)-1)
- As shown in the image, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.

Example

- Suppose an array is A. Starting index p & ending index r. Then denoted array is A[p..r].

  Let p = 0, r = 6

A

0	1	2	3	4	5	6
38	**27**	**43**	**3**	**9**	**82**	**10**

This is an unsorted array.

So sorted this array—

A

0	1	2	3	4	5	6
3	**9**	**10**	**27**	**38**	**43**	**82**

Example Explain(Divide)

0	1	2	3	4	5	6
38	27	43	3	9	82	10

0	1	2	3
38	27	43	3

4	5	6
9	82	10

0	1
38	27

2	3
43	3

4	5
9	82

6
10

0
38

1
27

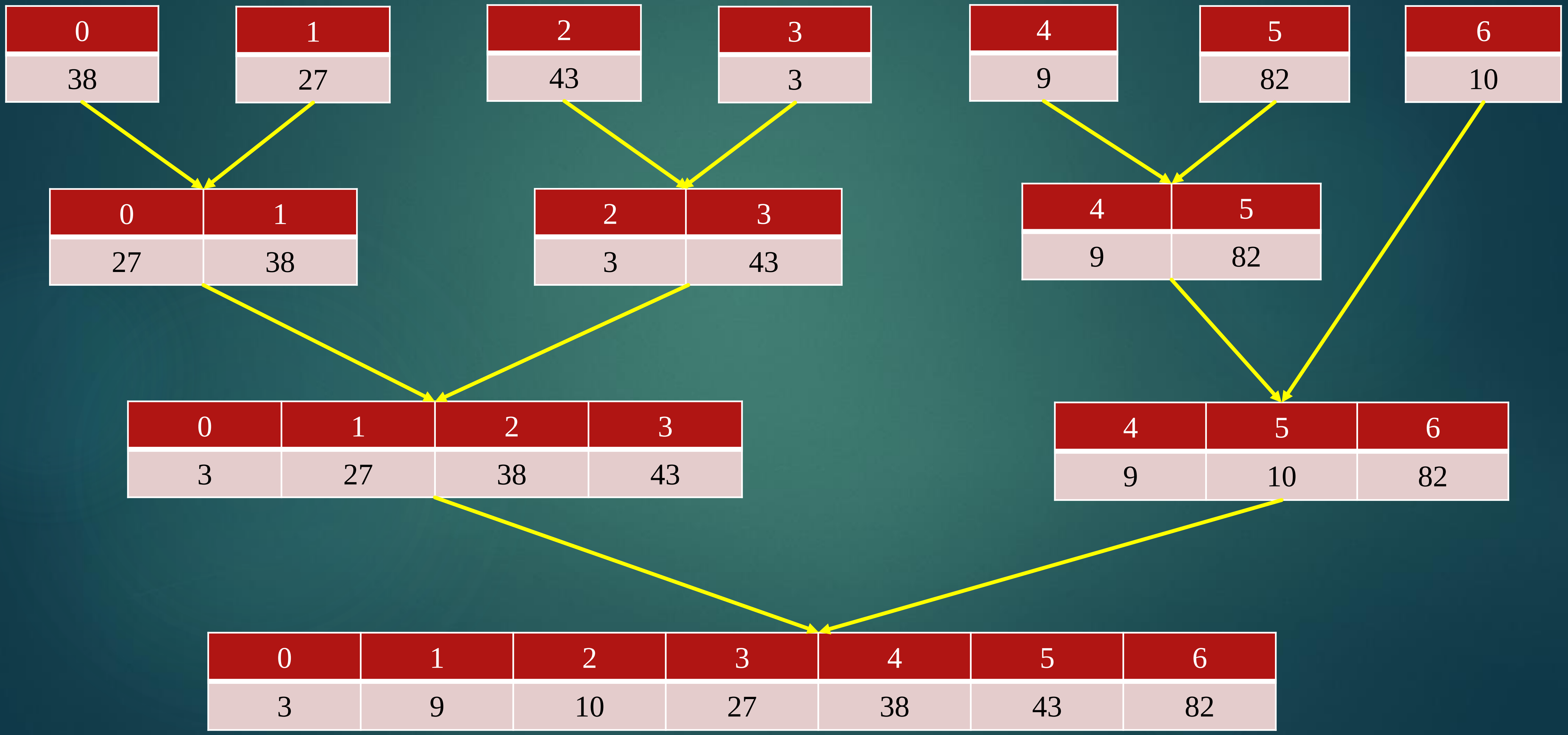
2
43

3
3

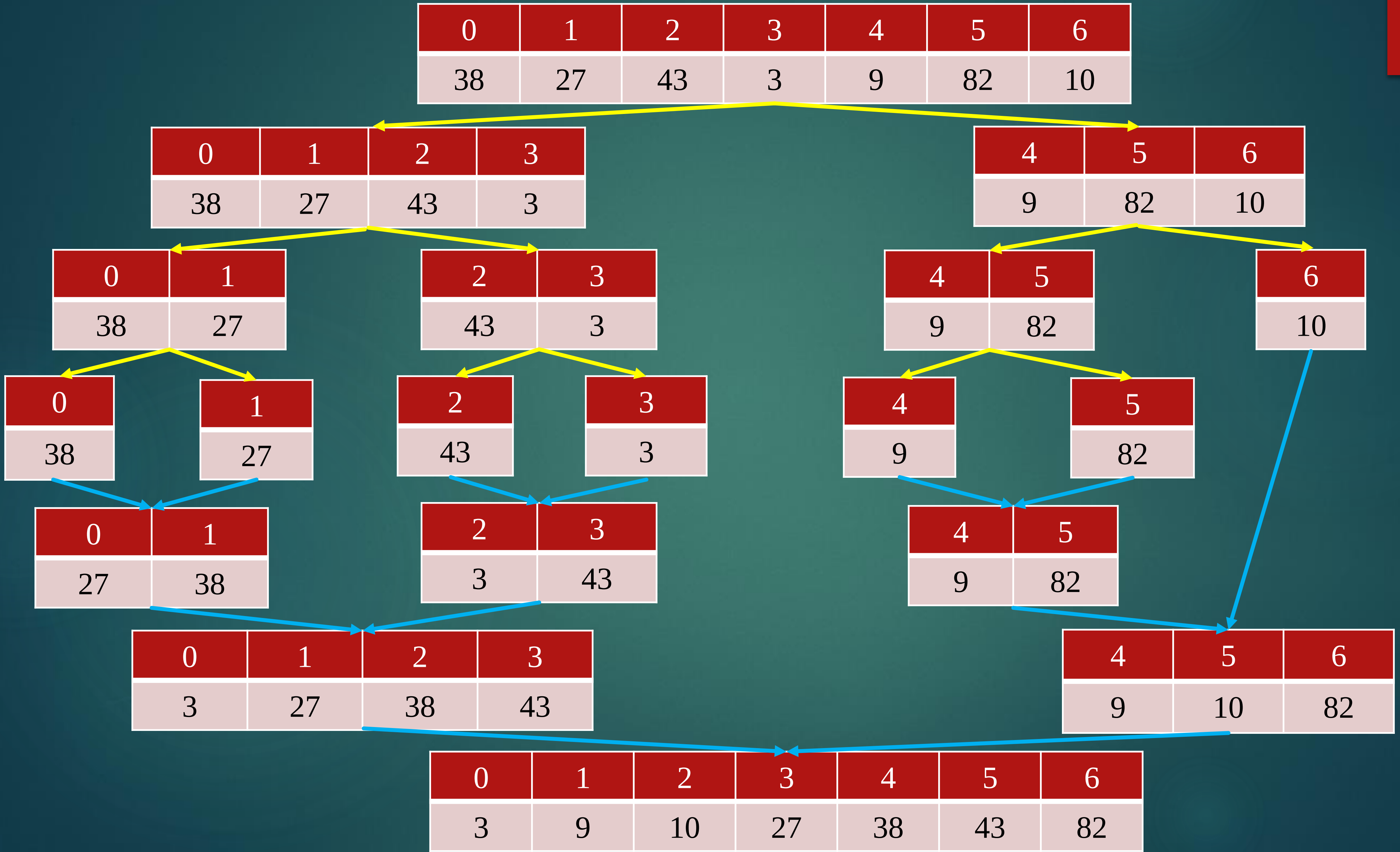
4
9

5
82

Example Explain(Conquer & Combine)



Example(Divide, Conquer & Combine)



Pseudocode

```
//Create a NJew Function________________
void Marge(int A[],int left,int mid,int right){
    int i,l_size,r_size;
    l_size = mid - left + 1;
    r_size = right - mid;
    int L[l_size],R[r_size];
    for(i = 0; i < l_size; i++){
        L[i] = A[left + i];
    }
    for(i = 0; i < r_size; i++){
        R[i] = A[mid + 1 + i];
    }
    int a_index, l_index=0, r_index=0;
    for(a_index = left; l_index < l_size && r_index < r_size; a_index++){
        if(L[l_index] <= R[r_index]){
            A[a_index] = L[l_index];
            l_index++;
        }
        else{
            A[a_index] = R[r_index];
            r_index++;
        }
    }
    while(l_index < l_size){
        A[a_index] = L[l_index];
        a_index++;
        l_index++;
    }
    while(r_index < r_size){
        A[a_index] = R[r_index];
        a_index++;
        r_index++;
    }
}
void Marge_Sort(int A[],int left,int right){
    if(left < right){
        int mid;
        mid = left + (right-left)/2;
        Marge_Sort(A,left,mid);
        Marge_Sort(A,mid+1,right);
        Marge(A,left,mid,right);
    }
```

C:\WINDOWS\system32\cmd.exe - pause

```
Plz enter the test case of T : 1
Plz enter the value of N: 7
The elements of A : 38 27 43 3 9 82 10
Before Marge Sorting Array : 38 27 43 3 9 82 10
After Marge Sorting Array : 3 9 10 27 38 43 82

Press any key to continue . . .
```

Time Complexity

- We denote with *n* the number of elements.
- Since we repeatedly divide the (sub)arrays into two equally sized parts, if we double the number of elements *n*, we only need one additional step of divisions *d*.

  if $n=4$ then $d=2$

  if $n=8$ then $d=3$

- On each merge stage, we have to merge a total of *n* elements (on the first stage $n \times 1$, on the second stage $n/2 \times 2$, on the third stage $n/4 \times 4$, etc.
- The merge process does not contain any nested loops, so it is executed with linear complexity: If the array size is doubled, the merge time doubles, too. The total effort is, therefore, the same at all merge levels.
- So we have n elements times $\log_2 n$ division and merge stages.
- Therefore: The time complexity of Merge Sort is: $O(n \log n)$

Conclusion

- Merge sort is an interesting algorithm and forms a great case-study to understand data structures and algorithms.
- In order to develop strong foundations in computer science, you are advised to thoroughly understand various sorting algorithms that will help you pick up the basics.
- Merge sort uses a *divide-and-conquer* method recursively sorts the elements of a list while *Bubble*, *Insertion* and *Selection* have a quadratic time complexity that limit its use to small number of elements.
- Merge sort uses *divide-and-conquer* to speed up the sorting.

Thank You