Divide-and-conquer method: Divide-and-conquer are probably the best known general algorithm design technique. The principle behind the Divide-and-conquer algorithm design technique is that it is easier to solve several smaller instance of a problem than the larger one.

The "divide-and-conquer" technique involves solving a particular problem by dividing it into one or more cub-problems of smaller size, recursively solving each sub-problem and then "merging" the solution of sub-problems to produce a solution to the original problem.

Divide-and-conquer algorithms work according to the following general plan.

1. **Divide:** Divide the problem into a number of smaller sub-problems ideally of about the same size.
2. **Conquer:** The smaller sub-problems are solved, typically recursively. If the sub-problem sizes are small enough, just solve the sub-problems in a straight forward manner.
3. **Combine:** If necessary, the solution obtained the smaller problems are connected to get the solution to the original problem.

The following figure shows-

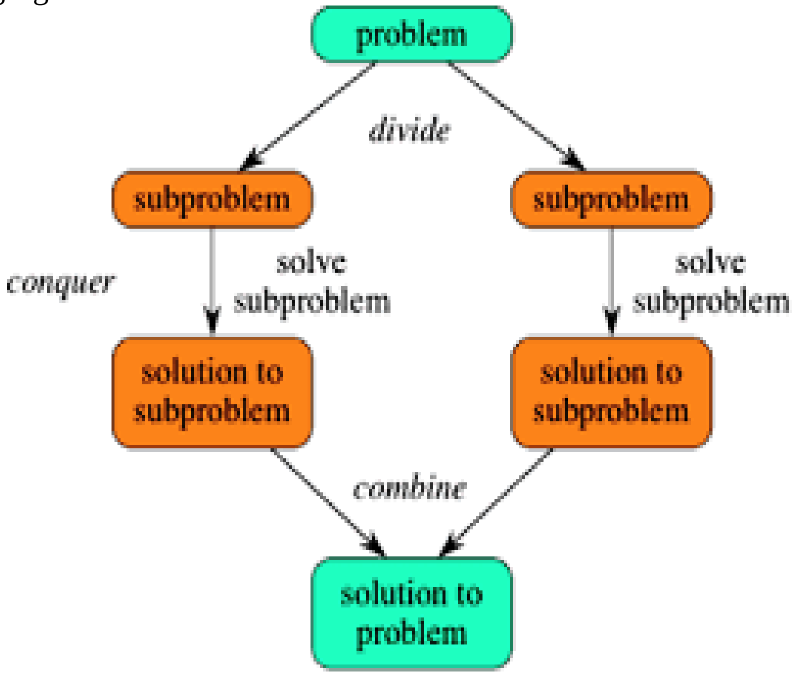


Fig: Divide and Conquer typical case

Control abstraction for divide-and-conquer technique:

Control abstraction means a procedure whose flow of control is clear but whose primary operations are satisfied by other procedure whose precise meanings are left undefined.

```
Algorithm DandC(p)
 {
    if small (p) then
    return S(p)
    else
       {
          Divide P into small instances $P_1$, $P_2$, $P_3$........$P_k$, $k\geq1$;
          Apply DandC to each of these sub-problems;\
          return combine (DandC($P_1$), DandC($P_1$),.... (DandC($P_k$);
       }
  }
```

Algorithm: Control abstraction for divide-and-conquer

DandC(p) is the divide-and-conquer algorithm, where P is the problem to be solved. Small(p) is a Boolean valued function(i.e., either true or false) that determines whether the input size is small enough that the answer can be computed without splitting. If this, is so the function S is invoked. Otherwise the problem P is divided into smaller sub-problems. These sub-problems P_1, P_2, P_3……..P_k, are solved by receive applications of DandC.

Combine is a function that combines the solution of the K sub-problems to get the solution for original problem 'P'.

Example: Specify an application that divide-and-conquer cannot be applied.

Solution: Let us consider the problem of computing the sum of n numbers a_0, a_1,…..a_{n-1}. If n>1, we divide the problem into two instances of the same problem. That is to compute the sum of the first [n/2] numbers and to compute the sum of the remaining [n/2] numbers. Once each of these two sum is compute (by applying the same method recursively), we can add their values to get the sum in question-

$$a_0+ a_1+\dots+a_{n-1}= (a_0+ a_1+\dots+a_{[n/2]-1})+ a_{[n/2]-1}+\dots\dots+ a_{n-1}).$$

For example, the sum of 1 to 10 numbers is as follows-

$$\begin{aligned}(1+2+3+4+\dots\dots+10) &= (1+2+3+4+5)+(6+7+8+9+10)\\ &= [(1+2) + (3+4+5)] + [(6+7) + (8+9+10)]\\ &= \dots\\ &= \dots\\ &= (1) + (2) +\dots\dots+ (10).\end{aligned}$$

This is not an efficient way to compute the sum of n numbers using divide-and-conquer technique. In this type of problem, it is better to use brute-force method.

Applications of Divide-and Conquer: The applications of divide-and-conquer methods are-

1. Binary search.
2. Quick sort
3. Merge sort.

Binary Search:

Binary search is an efficient searching technique that works with only sorted lists. So the list must be sorted before using the binary search method. Binary search is based on divide-and-conquer technique.

The process of binary search is as follows:

The method starts with looking at the middle element of the list. If it matches with the key element, then search is complete. Otherwise, the key element may be in the first half or second half of the list. If the key element is less than the middle element, then the search continues with the first half of the list. If the key element is greater than the middle element, then the search continues with the second half of the list. This process continues until the key element is found or the search fails indicating that the key is not there in the list.

Consider the list of elements: -4, -1, 0, 5, 10, 18, 32, 33, 98, 147, 154, 198, 250, 500.
Trace the binary search algorithm searching for the element -1.

Sol: The given list of elements are:

Low														High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Searching key '-1': Here the key to search is '-1'

First calculate mid;

Mid = (low + high)/2

= (0 +14) /2 =7

Low							Mid							High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half → ← Second Half →

Here, the search key -1 is less than the middle element (32) in the list. So the search process continues with the first half of the list.

Low						High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid = (0+6)/2

=3.

Low			Mid			High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half → ←Second Half→

The search key '-1' is less than the middle element (5) in the list. So the search process continues with the first half of the list.

Low		High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid= (0+2)/2

=1

Low	Mid	High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Here, the search key -1 is found at position 1.

The following algorithm gives the *iterative binary Search Algorithm*

```
Algorithm BinarySearch(a, n, key)
  {
    // a is an array of size n elements
    // key is the element to be searched
    // if key is found in array a, then return j, such that
    //key = a[i]
    //otherwise return -1.
      Low: = 0;
      High: = n-1;
     While (low ≤ high) do
       {
          Mid: = (low + high)/2;
          If ( key = a[mid]) then
               Return mid;
          Else if (key < a[mid])
              {
                High: = mid +1;
              }
         Else if( key > a[mid])
             {
               Low: = mid +1;
              }
  }
```

The following algorithm gives *Recursive Binary Search*

```
   Algorithms Binsearch ( a, n, key, low, high)
         {
          // a is array of size n
         // Key is the element to be searched
        // if key is found then return j, such that key = a[i].
       //otherwise return -1
If ( low ≤ high) then
  {
     Mid: = (low + high)/2;
          If ( key = a[mid]) then
               Return mid;
          Else if (key < a[mid])
               Binsearch ( a, n, key, low, mid-1);
          Else if ( key > a[mid])
               Binsearch ( a, n, key, mid+1, high);
     }
  Return -1;
}
```

Advantages of Binary Search: The main advantage of binary search is that it is faster than sequential (linear) search. Because it takes fewer comparisons, to determine whether the given key is in the list, then the linear search method.

Disadvantages of Binary Search: The disadvantage of binary search is that can be applied to only a sorted list of elements. The binary search is unsuccessful if the list is unsorted.
Efficiency of Binary Search: To evaluate binary search, count the number of comparisons in the best case, average case, and worst case.
Best Case: The best case occurs if the middle element happens to be the key element. Then only one comparison is needed to find it. Thus the efficiency of binary search is O(1).
Ex: Let the given list is: 1, 5, 10, 11, 12.

Low		Mid		High
1	5	$\boxed{10}$	11	12

Let key = 10.
Since the key is the middle element and is found at our first attempt.
Worst Case: Assume that in worst case, the key element is not there in the list. So the process of divides the list in half continues until there is only one item left to check.

Items left to search	*Comparisons so far*
16	0
8	1
4	2
2	3
1	4

For a list of size 16, there are 4 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half.
In general, if n is the size of the list and c is the number of comparisons, then

$$C = \log_2 n$$
$$\therefore \text{Eficiency in worst case} = O(\log n)$$

Average Case: In binary search, the average case efficiency is near to the worst case efficiency. So the average case efficiency will be taken as O(log n).
∴ Efficiency in average case = O (log n).

Binary Search	
Best Case	O(1)
Average Case	O(log n)
Worst Case	O(log n)

Space Complexity is O(n)

Quick Sort:

The quick sort is considered to be a fast method to sort the elements. It was developed by CAR Hoare. This method is based on divide-and-conquer technique i.e. the entire list is divided into various partitions and sorting is applied again and again on these partitions. This method is also called as partition exchange sorts.
The quick sort can be illustrated by the following example
12 6 18 4 9 8 2 15

The reduction step of the quick sort algorithm finds the final position of one of the numbers. In this example, we use the first number, 12, which is called the pivot (rotate) element. This is accomplished as follows-

Let 'i' be the position of the second element and 'j' be the position of the last element. i.e. i =2 and j =8, in this example.

Assume that a [n+1] =∞, where '**a**' is an array of size n.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	i	j
12	6	18	4	9	8	2	15	α	2	8

First scan the list from left to right (from i to j) can compare each and every element with the pivot. This process continues until an element found which is greater than or equal to pivot element. If such an element found, then that element position becomes the value of 'i'.

Now scan the list from right to left (from j to i) and compare each and every element with the pivot. This process continues until an element found which is less than or equal to pivot element. If such an element finds then that element's position become 'j' value.

Now compare 'i' and 'j'. If i <j, then swap a[i] and a[j]. Otherwise swap pivot element and a[j].

Continue the above process the entire list is sorted.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	i	j
$\textcircled{12}$	6	18	4	9	8	2	15	α	2	8
$\textcircled{12}$	6	18	4	9	8	2	15	α	3	7
12	6	2	4	9	8	18	15	α	7	6

Since i = 7 $\nless$ j=6, then swap pivot element and 6th element (j^{th} element), we get

8 6 2 4 9 $\textcircled{12}$ 18 15

Thus pivot reaches its original position. The elements on left to the right pivot are smaller than pivot (12) and right to pivot are greater pivot (12).

$$\underbrace{8\quad 6\quad 2\quad 4\quad 9}_{\text{Sublist 1}}\quad \textcircled{12}\quad \underbrace{18\quad 15}_{\text{Sublist 2}}$$

Now take sub-list**1** and sub-list**2** and apply the above process recursively, at last we get sorted list.

Ex 2: Let the given list is-

8 18 56 34 9 92 6 2 64

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	i	j
8	18	56	34	9	92	6	2	64	α	2	98
8	18	56	34	9	92	6	2	64	α	2	8
8	2	56	34	9	92	6	18	64	α	3	7
8	2	6	34	9	92	56	18	64	α	4	3

Since i $\nless$ j, then swap j^{th} element, and pivot element, we get

$$\overset{\longleftrightarrow}{6\quad 2}_{\text{Sublist 1}}\quad \textcircled{8}\quad \overset{}{\underset{\longleftrightarrow}{34\quad 9\quad 92\quad 56\quad 18\quad 64}}_{\text{Sublist 2}}$$

Now take a sub-list that has more than one element and follow the same process as above. At last, we get the sorted list that is, we get

2 6 8 9 18 34 56 64 92

The following algorithm shows the quick sort algorithm-

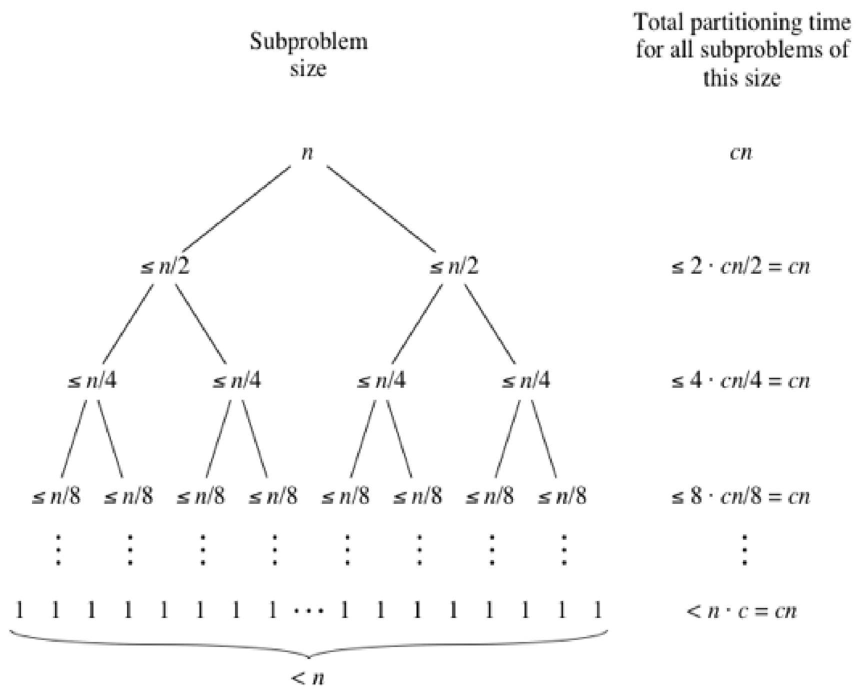
```
Algorithm Quicksort(i, j)
  {
    // sorts the array from a[i] through a[j]
      If ( i <j) then     //if there are more than one element
        {
            //divide P into two sub-programs
                K: = partition (a, i, j+1);
                //Here K denotes the position of the partitioning element
                 //solve the sub problems
                 Quicksort(i, K-1);
                 Quicksort(K=1, j);
              // There is no need for combining solution
        }
  }
Algorithm Partition (a, left, right)
  {
     // The element from a[left] through a[right] are rearranged in such a manner that if initially
     // pivot =a[left] then after completion a[j]= pivot, then return. Here j is the position where
     // pivot partition the list into two partitions. Note that a[right]= ∞.
            pivot: a[left];
             i:= left;  j:=right;
          repeat
           {
             repeat
              i: =i+1;
            until (a[i] ≥ pivot);
            repeat
               j: =j-1;
              until (a[j] < pivot);
                 if( i<j) then
                   Swap (a, i, j);
            }until (i ≥ j);
             a[left]: = a[j];
             a[j]: = pivot;
            return j;
  }
Algorithm Swap (a, i, j)
  {
     //Example a[i] with a[j]
          temp:= a[i];
         a[i]: = a[j];
         a[j]:= temp;
  }
```

Advantages of Quick-sort: Quick-sort is the fastest sorting method among all the sorting methods. But it is somewhat complex and little difficult to implement than other sorting methods.

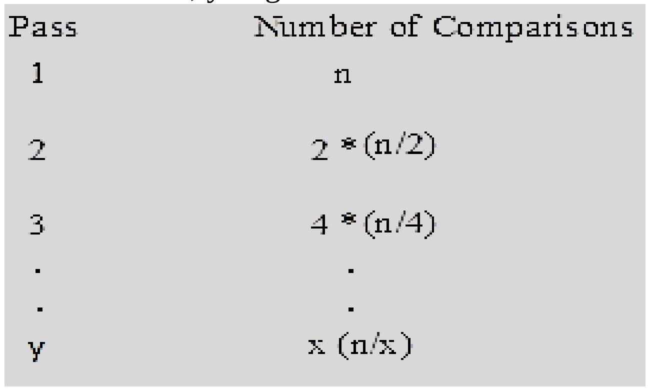
Efficiency of Quick-sort: The efficiency of Quick-sort depends upon the selection of pivot element.

Best Case: In best case, consider the following two assumptions-

1. The pivot, which we choose, will always be swapped into the exactly the middle of the list. And also consider pivot will have an equal number of elements both to its left and right.
2. The number of elements in the list is a power of 2 i.e. n= 2^y.



This can be rewritten as, y=$\log_2$n.



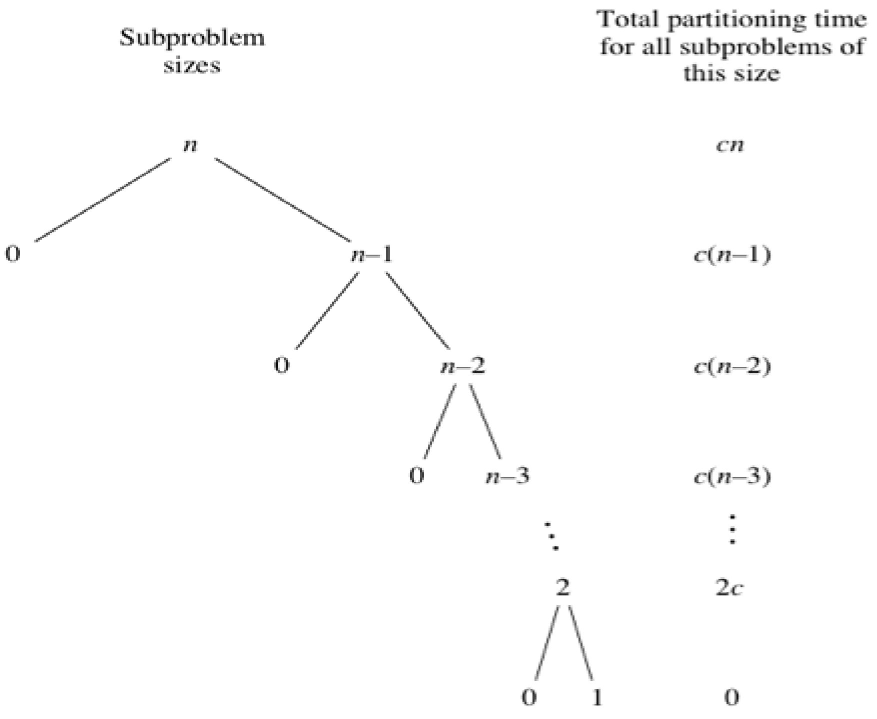
Pass	Number of Comparisons
1	n
2	2 * (n/2)
3	4 * (n/4)
.	.
.	.
y	x (n/x)

Thus, the total number of comparisons would be

O (n) + O (n) + O (n) +……..(y terms)
= O (n * y).

∴ Efficency in best case O(n log n) (∵ y=$\log_2$ n)

Worst Case: In worst case, assume that the pivot partition the list into two parts, so that one of the partition has no elements while the other has all the other elements.

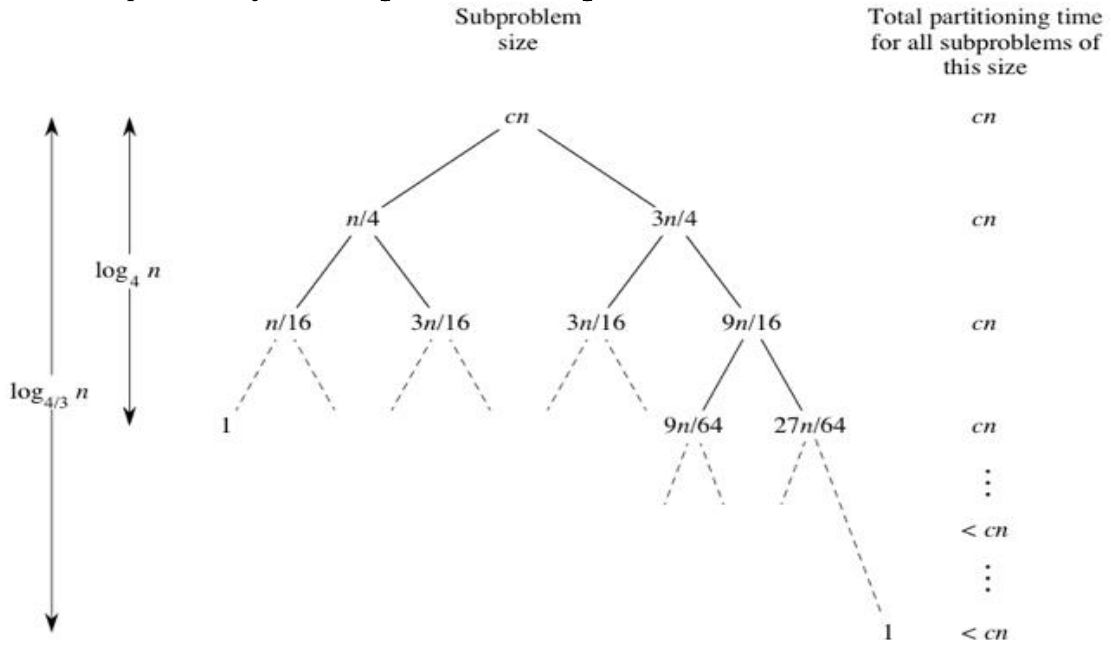


∴ Total number of comparisons will be-

$$(n-1)+(n-2)+(n-3)+\dots\dots\dots\dots+2+1=\frac{n(n-1)}{2}=O(n^2)$$

∴ Thus, the efficiency of quick-sort in worst case is O (n^2).

Average Case: Let cA(n) be the average number of key comparisons made by quick-sort on a list of elements of size n. assuming that the partitions split can happen in each position k(1≤k≤n) With the same probability 1/n, we get the following recurrence relation.



The left child of each node represents a sub-problem size 1/4 as large, and the right child represents a sub-problem size 3/4 as large.

There are $\log_{4/3} n$ levels, and so the total partitioning time is $O(n\log_{4/3}n)$. Now, there's a mathematical fact that

$\log_a n = \log_b n / \log_b a$

for all positive numbers a, b, and n. Letting a=4/3 and b=2, we get that $\log_{4/3} n = \log n / \log(4/3)$

Quick Sort	
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	$O(n^2)$

Space Complexity O(n)

Merge Sort:

Merge sort is based on divide-and-conquer technique. Merge sort method is a two phase process-

1. Dividing
2. Merging

Dividing Phase: During the dividing phase, each time the given list of elements is divided into two parts. This division process continues until the list is small enough to divide.

Merging Phase: Merging is the process of combining two sorted lists, so that, the resultant list is also the sorted one. Suppose A is a sorted list with **n** element and B is a sorted list with $\mathbf{n_2}$ elements. The operation that combines the elements of A and B into a single sorted list C with $n=n_1 + n_2$, elements is called merging.

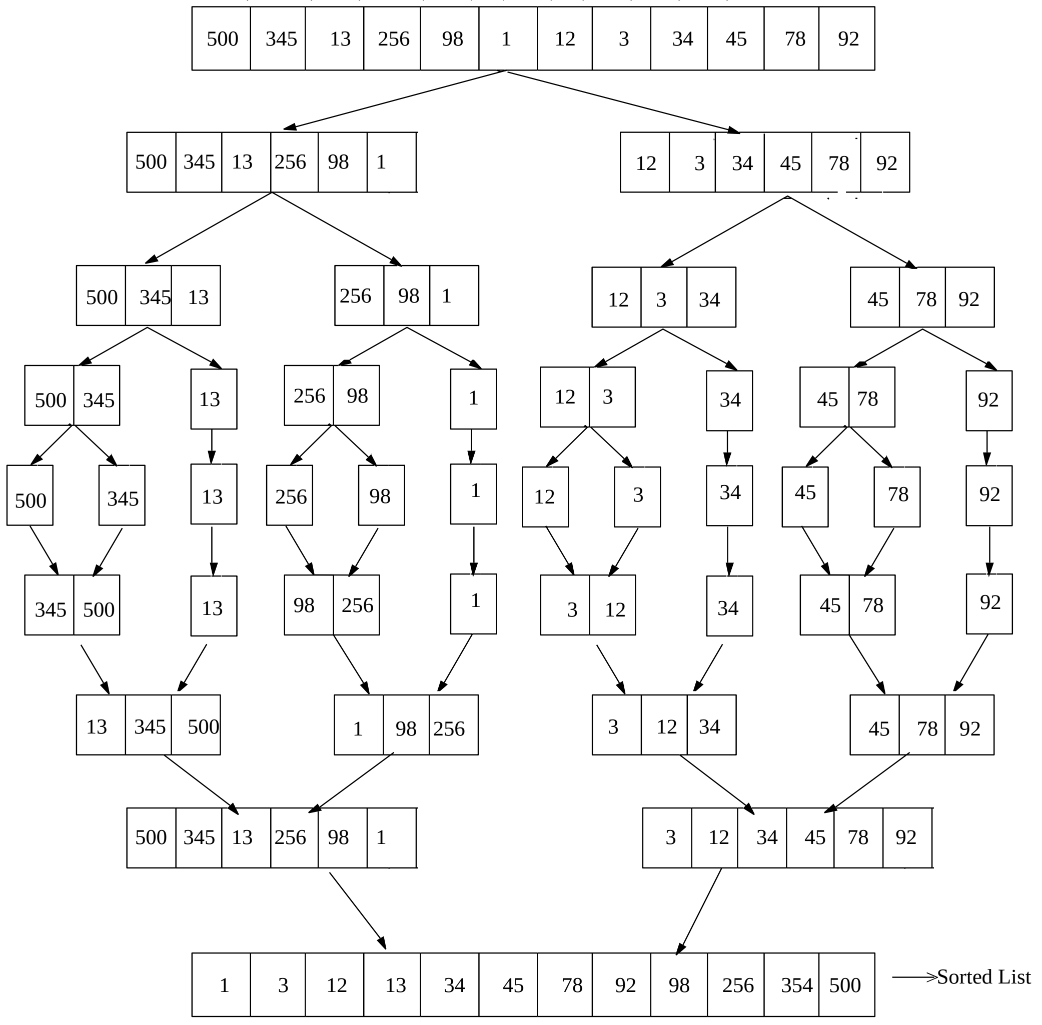
Algorithm-(Divide algorithm)

```
Algorithm Divide (a, low, high)
{
  // a is an array, low is the starting index and high is the end index of a

   If( low < high) then
     {
       Mid: = (low + high) /2;
       Divide( a, low, mid);
       Divide( a,  mid +1, high);
       Merge(a, low, mid, high);
     }

}
```

The merging algorithm is as follows-

```
Algorithm Merge( a, low, mid, high)
   {
            L:= low;
              H:= high;
               J:= mid +1;
              K:= low;

              While (low ≤ mid AND  j ≤  high) do
               {
                          If (a[low < a[j]) then
                           {
                              B[k] = a[low];
                              K:= k+1;
                              Low:= low+1;
                           }
                           Else
                          {
                             B[k]= a[j];
                             K: = k+1;
                             J: = j+1;
                          }
              }

            While (low ≤ mid) do
            {
               B[k]=a[low];
               K: = k+1;
               Low: =low + 1;
             }

            While (j ≤ high) do
            {
               B[k]=a[j];
               K: = k+1;
               j: =j + 1;
             }

             //copy elements of b to a
             For i: = l to n do
             {
               A[i]: =b[i];
             }
}
```

Ex: Let the list is: - 500, 345, 13, 256, 98, 1, 12, 3, 34, 45, 78, 92.



The merge sort algorithm works as follows-

Step 1: If the length of the list is 0 or 1, then it is already sorted, otherwise,

Step 2: Divide the unsorted list into two sub-lists of about half the size.

Step 3: Again sub-divide the sub-list into two parts. This process continues until each element in the list becomes a single element.

Step 4: Apply merging to each sub-list and continue this process until we get one sorted list.

Efficiency of Merge List: Let 'n' be the size of the given list/ then the running time for merge sort is given by the recurrence relation.

$$T(n) = \begin{cases} a & \text{if } n=1, \text{ a is a constant} \\ 2T(n/2) + Cn & \text{if } n>1, \text{ C is constant} \end{cases}$$

Assume that 'n' is a power of 2 i.e. $n=2^k$.

This can be rewritten as $k=\log_2 n$.

Let $T(n) = 2T(n/2) + Cn \longrightarrow \textcircled{1}$

We can solve this equation by using successive substitution.

Replace n by n/2 in equation, $\textcircled{1}$,we get

$$T(n/2) = 2T(n/4) + \frac{Cn}{2} \text{ ——— } \textcircled{2}$$

$$\text{Thus,}\quad T(n) = 2\left(2T(n/4) + \frac{Cn}{2}\right) + Cn$$

$$= 4T(n/4) + 2Cn$$

$$= 4T\left(2T(n/8) + \frac{Cn}{4}\right) + 2Cn$$

$$\cdots$$

$$\cdots$$

$$\cdots$$

$$= 2^{k} T(1) + KCn \left(\because k = \log_2 n\right)$$

$$= a\, n + Cn \log n$$

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

Worst case	O(*n* log *n*)
Best case	O(*n* log *n*)
Average case	O(*n* log *n*)
Space Complexity	O(2*n*)