Divide-and-conquer algorithm

Divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting, multiplying large numbers, finding the closest pair of points, and syntactic analysis.

Designing efficient divide-and-conquer algorithms can be difficult. As in mathematical induction, it is often necessary to generalize the problem to make it amenable to a recursive solution. The correctness of a divide-and-conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

The divide-and-conquer paradigm is often used to find an optimal solution of a problem. Its basic idea is to decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem. Problems of sufficient simplicity are solved directly.

The name "divide and conquer" is sometimes applied to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list. These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular. Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide-and-conquer algorithm".

A divide and conquer algorithm is a strategy of solving a large problem by using the following steps.

- Divide: Divide the given problem into sub-problems using recursion.
- Conquer: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
- Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

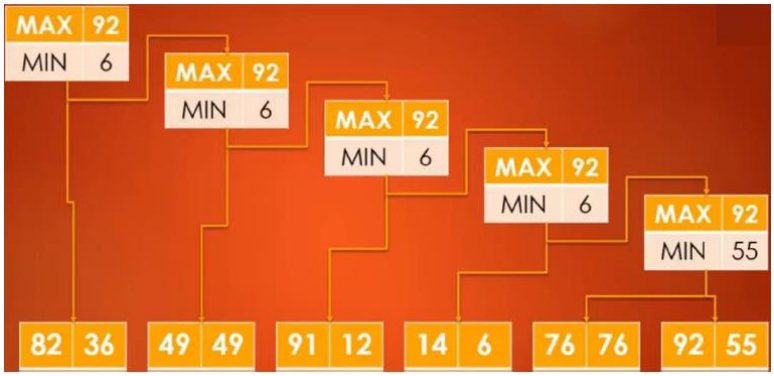
For example, to sort a given list of n natural numbers, split it into two lists of about n/2 numbers each, sort each of them in turn, and interleave both results appropriately to obtain the sorted version of the given list (see the picture). This approach is known as the merge sort algorithm.

```
DAC(a, i, j)
{
        if(small(a, i, j))
                return(Solution(a, i, j))
        else
                m  = divide(a, i, j)            // f1(n)
                b  = DAC(a, i, mid)             // T(n/2)
                c  = DAC(a, mid+1, j)           // T(n/2)
                d  = combine(b, c)              // f2(n)
        return(d)
}
```

This is a recurrence relation for the above program.

O(1) if n is small
T(n) = f1(n) + 2T(n/2) + f2(n)

Example: Find the maximum and minimum element in a given array.



Input : {70, 250, 50, 80, 140, 12, 14}
Output : The minimum number in a given array is : 12

Approach: To find the maximum and minimum element from a given array is an application for divide and conquer. In this problem, we will find the maximum and minimum elements in a given array. In this problem, we are using a divide and conquer approach(DAC) which has three steps divide, conquer and combine.

For Maximum: We are using the recursive approach to find maximum where we will see that only two elements are left and then we can easily using condition, i.e. if(a[index]>a[index+1]).

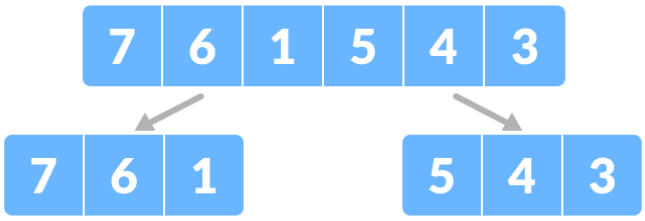
```
int[] findMinMax(int A[], int start, int end)
{
        int max;
        int min;
        if ( start == end ) {
                max = A[start]
                min = A[start]
        }
        else if ( start + 1 == end ) {
                if ( A[start] < A[end] ) {
                        max = A[end]
                        min = A[start]
                }
                else {
                        max = A[start]
                        min = A[end]
                }
        }
        else {
                int mid = start + (end - start)/2
```

```
            int left[] = findMinMax(A, start, mid)
            int right[] = findMinMax(A, mid+1, end)

            if ( left[0] > right[0] )   max = left[0]
            else                        max = right[0]

            if ( left[1] < right[1] )   min = left[1]
            else                        min = right[1]
        }

        // By convention, we assume ans[0] as max and ans[1] as min
        int ans[2] = {max, min};
        return ans
}
```

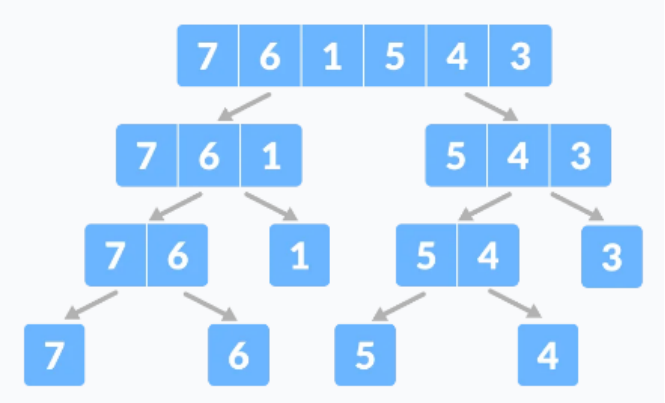
Recurrence: T(n) = 2 T(n/2) + 2

Example: Sort an array using the divide and conquer approach, merge sort.
Let the given array be:

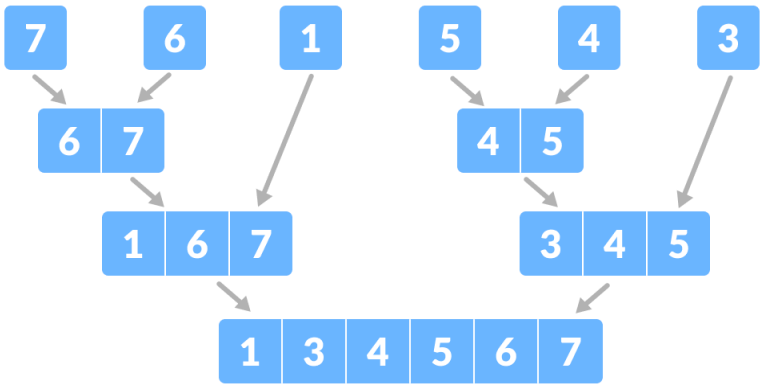
Divide the array into two subparts.



Again, divide each subpart recursively into two halves until you get individual elements.



Now, combine the individual elements in a sorted manner. Here, conquer and combine steps go side by side.



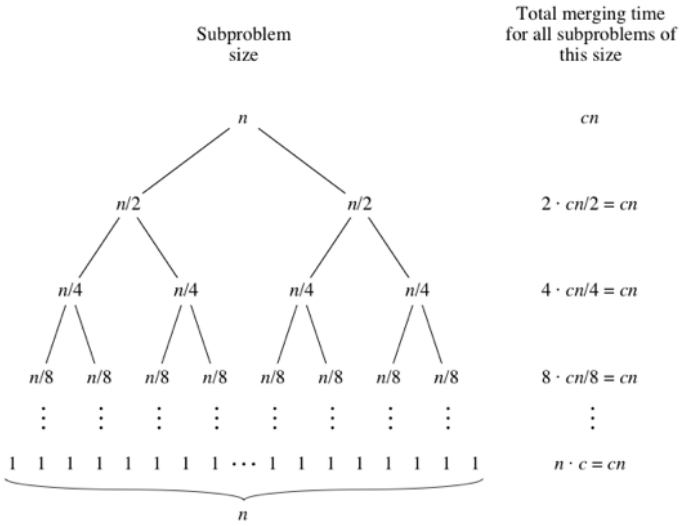
Time Complexity:

T(n) = aT(n/b) + f(n)
= 2T(n/2) + O(n)

Where,

a = 2 … Each time, a problem is divided into 2 subproblems
n/b = n/2 ... Size of each sub problem is half of the input
f(n) = Time taken to divide the problem and merging the subproblems

T(n) = 2T(n log n) + O(n)
≈ O(n log n)



Now we know how long merging takes for each subproblem size. The total time for mergeSort is the sum of the merging times for all the levels. If there are l levels in the tree, then the total merging time is $l.cn$.