• QUICK SORT (DVIDE AND CONQURE)

INTRODUCTION

The term ‘DIVIDE AND CONQUER’ is generally used in the sociological context. It generally refers to the strategy of breaking the unity in a given population in order to achieve some intended goal. It is easy to psychologically handle and manipulate a smaller group rather than handling a larger one. The divide and conquer strategy used in algorithms works only if there is a way to club together the solutions of the sub-problems. In contrast, the strategy used in sociological context does not require a strategy to merge the segregated population.

CONCEPT OF DIVIDE AND CONQUER

In order to accomplish a task using the above technique, the following steps need to be carried out:

Step 1 Divide the domain into SMALL (atomic level), where SMALL is the basic unit whose solution is known.

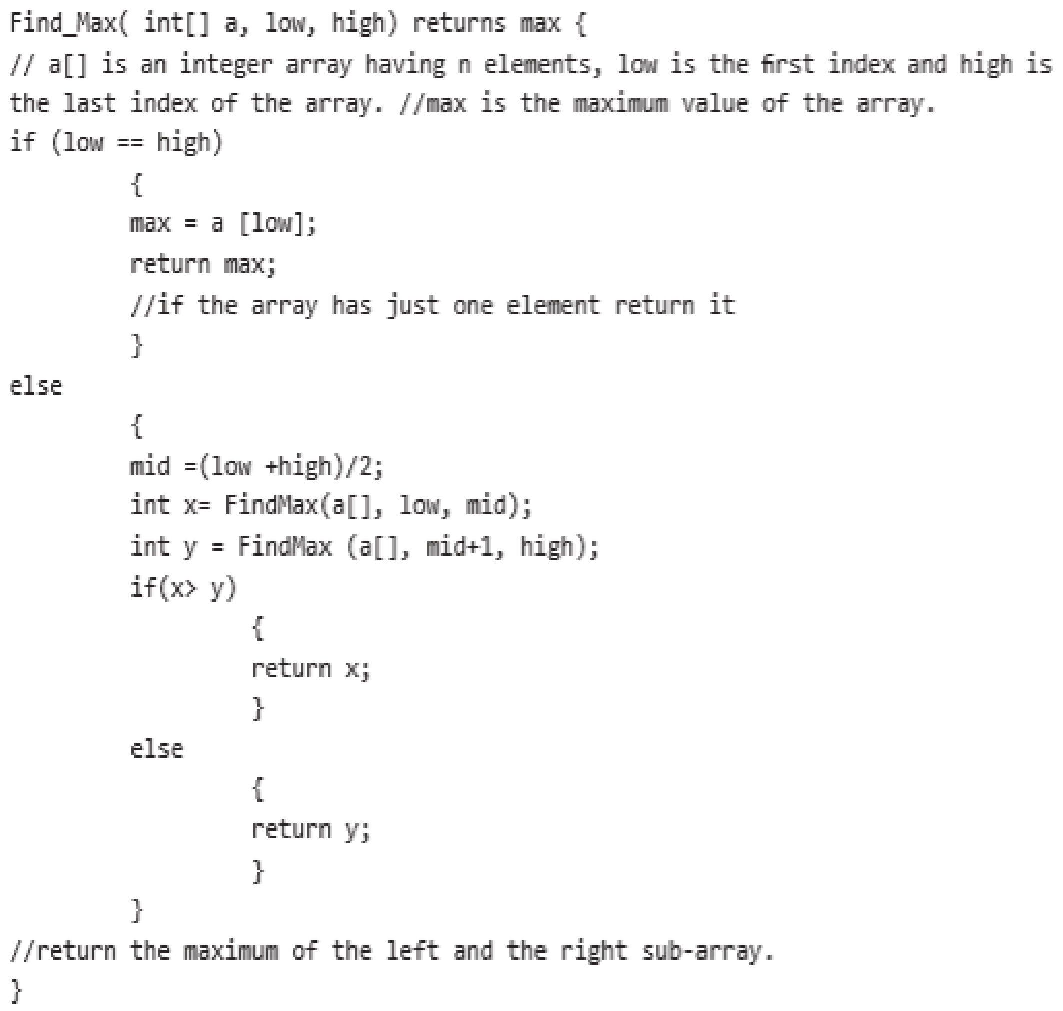
Step 2 Solve individual sub-problems using the solution of SMALL.

Step 3 Combine the sub-solutions to get the final answer.

In order to understand the above process, let us consider an example shown in Algorithm below. An array of n numbers is given and it is required to find the maximum element of the array using divide and conquer approach.

ALGORITHM: MAX

Input: an array a[], consisting of n elements. Output: The maximum element of the array, MaX. Strategy: Divide the array into two parts and continue dividing the sub-parts till one element remains in the array. For example, an array of eight elements would be divided into two arrays of four elements. The two arrays would then be divided into four arrays of two elements each and in the last step; the four arrays would be divided into eight arrays of one element each. Now the largest element from two arrays (containing a single element) would be the solution of the sub-problem. The procedure is repeated for the rest of the elements also. After this step, four elements would remain. now, these four elements would be paired in groups of two, to give two elements which are greatest in their respective groups. This is followed by the selection of the largest element.



```
Find_Max( int[] a, low, high) returns max {
// a[] is an integer array having n elements, low is the first index and high is
the last index of the array. //max is the maximum value of the array.
if (low == high)
        {
        max = a [low];
        return max;
        //if the array has just one element return it
        }
else
        {
        mid =(low +high)/2;
        int x= FindMax(a[], low, mid);
        int y = FindMax (a[], mid+1, high);
        if(x> y)
                {
                return x;
                }
        else
                {
                return y;
                }
        }
//return the maximum of the left and the right sub-array.
}
```

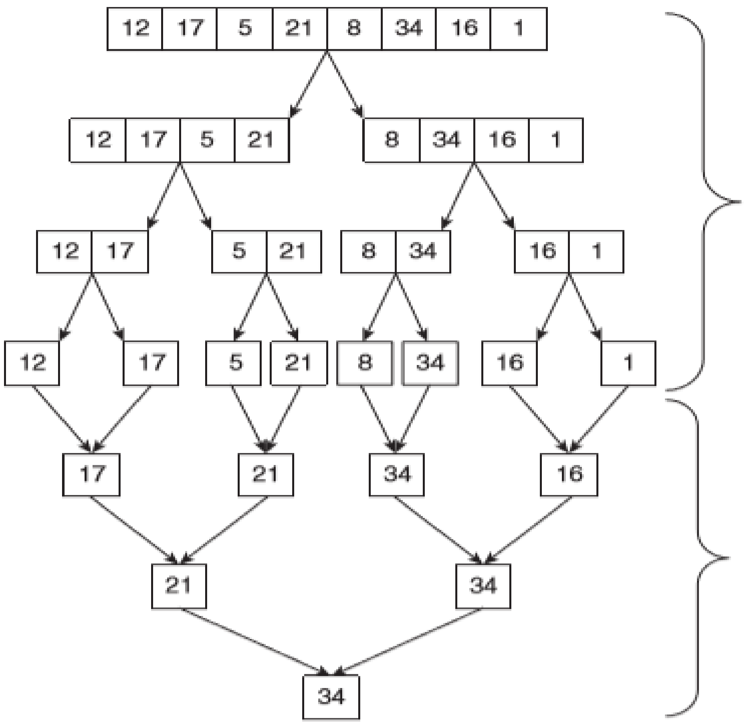


Figure 1: The implementation to finding the maximum element by divide and conquer

COMPLEXITY: Initially, there are n elements, after the first iteration; there would be n/2 elements in each array. The process stops when a single element remains in the array. This would require $\log_2 n$ steps. After this a single element is selected in each step. So, the complexity is proportional to log2n, i.e.,

$$T(n) = O(\log_2 n)$$

APPLICATIONS: The above strategy can also be used to calculate the minimum from the array. In order to calculate the minimum value, instead of the maximum, choose the minimum at each step. Figure2, illustrates the process of selection of minimum element using divide and conquer approach. The rest of the chapter relies heavily on recursive equations and their solutions. Therefore, it is important to understand the procedures of solving them.

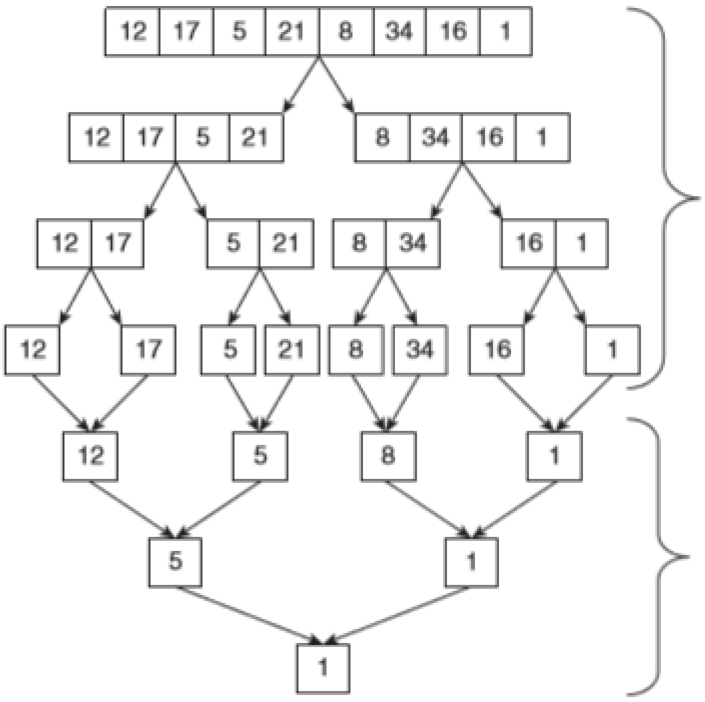


Figure 2 finding minimum element by divide and conquer approach

Example:

1. Divide
2. Conquer
3. Combine

X=[30, 10, 35, 50, 88, 22, 64] **(35,88)**

35 88

(30, 35) 30, 10, 35 60, 88, 22, 63 **(88,63)**

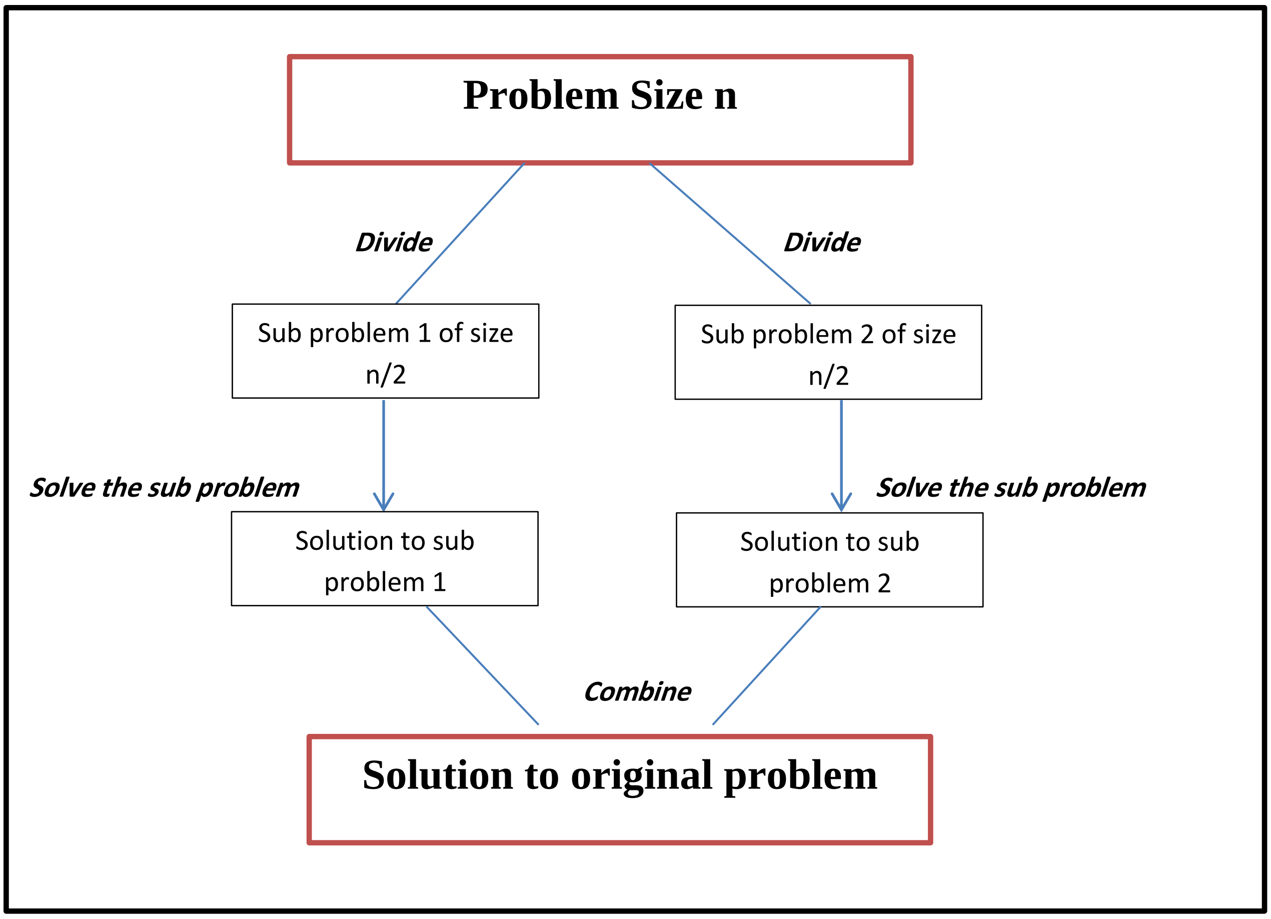
30 35 88 63

30, 10 35 60, 88 22, 63

Break a problem into roughly equal sited, sub problem, solve separately, combine results.

So, the steps include the following:

1. ***Divide problem into smaller parts.***
2. **Independently solve the parts.**
3. **Combine these solutions to get overall solution.**



Problem Size n
Divide
Divide
Sub problem 1 of size n/2
Sub problem 2 of size n/2
Solve the sub problem
Solve the sub problem
Solution to sub problem 1
Solution to sub problem 2
Combine
Solution to original problem

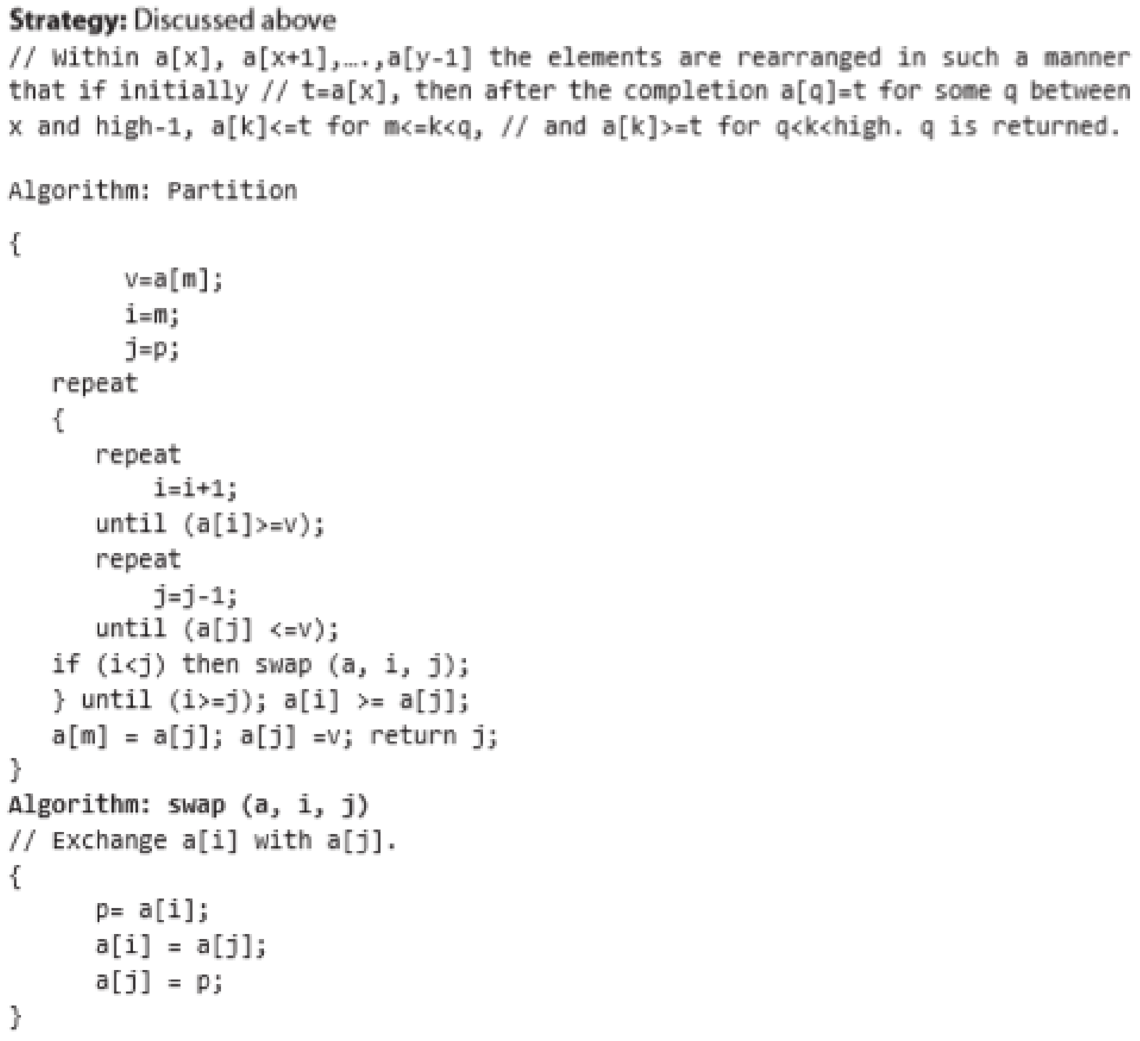
Figure: Show the Divide and Conquer approach

QUICK SORT

In quick sort, the pivot is compared with the first and the last element of the list. The pointer which moves from left to right, starting from the first element, would henceforth be called i. The pointer which moves from right to left, starting from the last element of the list would henceforth be called j. The pivot is compared with a[i], where a[] denotes the array. If a[i] is less than the pivot, i is incremented. The process continues till a[i] remains less than pivot. The pivot is then compared with a[j], where a[] denotes the array. If a[j] is greater than the pivot, j is decremented. The process continues till a[i] remains less than pivot. When a[i] > pivot and a[j] < pivot, they are swapped. The process stops when i become greater than j. At this point, the pivot is placed at the position denoted by the index where i become greater than j. This step places pivot at its appropriate position. After this step, the elements less than pivot will be to the left of pivot and those greater than pivot will be to the right of the pivot. The left sub-array and the right sub-array will now undergo the above procedure. Finally, a sorted array is obtained. The following algorithm presents a formal approach to quick sort.

ALGORITHM: Partition (a, x, y)

Input: an array: a[ ], low: the first index of the array, high: the last index of the array.
Output: a sorted array



Strategy: Discussed above

```
// Within a[x], a[x+1],….,a[y-1] the elements are rearranged in such a manner
that if initially // t=a[x], then after the completion a[q]=t for some q between
x and high-1, a[k]<=t for m<=k<q, // and a[k]>=t for q<k<high. q is returned.

Algorithm: Partition

{
        v=a[m];
        i=m;
        j=p;
   repeat
   {
      repeat
          i=i+1;
      until (a[i]>=v);
      repeat
          j=j-1;
      until (a[j] <=v);
   if (i<j) then swap (a, i, j);
   } until (i>=j); a[i] >= a[j];
   a[m] = a[j]; a[j] =v; return j;
}
Algorithm: swap (a, i, j)
// Exchange a[i] with a[j].
{
      p= a[i];
      a[i] = a[j];
      a[j] = p;
}
```

```
Algorithm: QuickSort (low, high)
{
   // Sorts the elements a[low]… a[high] into ascending order;
   // a [n+1] is considered to be defined and must be >= all the elements in a
   [1: n].
   if (low<high) then // If there are more than one element
   {
      // divide array into two sub arrays.
          j= Partition (a, low, high);
             // j is the position of the partition element.
      Quicksort (low, j-1);
      QuickSort (j+1, high);
      }
}
```

COMPLEXITY: In the average case, the array would be divided into two arrays of equal size. Each sub-array will be divided into two arrays of number of elements half of the parent. The process continues till just one element remains in the child array. The situation is depicted in the following diagram. The process stops when

$$\frac{n}{2^i} = 1$$

i.e., $n = 2^i$

or, $i = \log_2 n$

Since, there are $\log_2 n$ levels, the complexity of algorithm becomes $O(n \times \log_2 n)$.

Figure 9.4 depicts the tree corresponding to the situation.

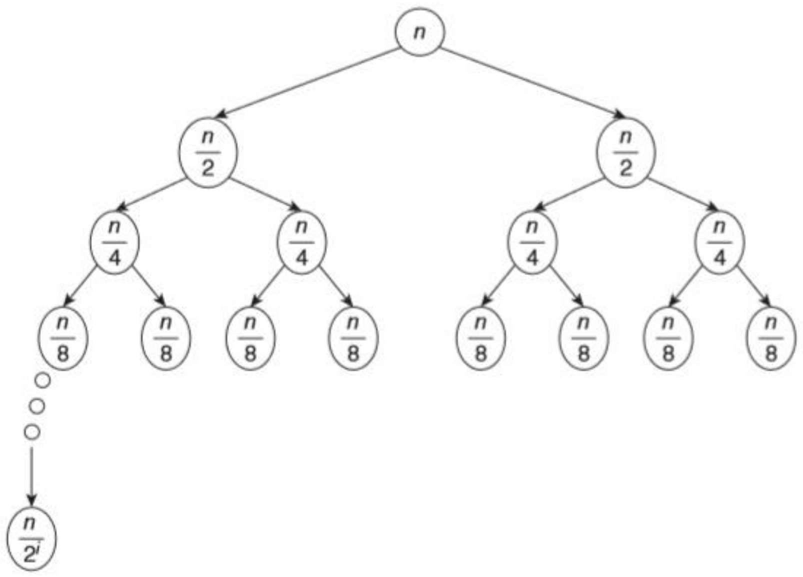


Figure 3: average-case complexity of quick sort

Worst-case Complexity If partition leads to segregation of the array into two parts, one having (n − 1) elements and the other having 1 element (in the next iteration also the pattern is repeated). This happens when the array is already sorted. In that case, the array would be divided into two parts and the first part will always have a single element. The situation is depicted in the following diagram. The number of comparisons in this case would be

$$T(n) = (n-1) + T(n-1),\ T(1) = 1$$

i.e., $T(n) = (n-1) + (n-2) + \cdots + 1$

or, $T(n) = n \times \frac{n-1}{2}$

or, $T(n) = O(n^2)$

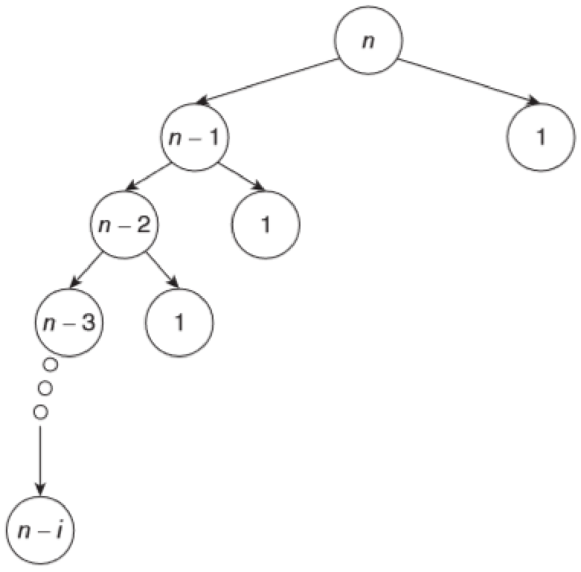


Figure 4: depicts the above division