Divide & Conquer Approach

I] General Method

- This strategy works for algorithms that are recursive in structure i.e to solve a problem, algorithms recursively calls themselves one or more time to deal with closely related subproblems.

- In this,

" Problem is broken into several subproblems, that are similar to the original problem but smaller in size, solve the subproblem recursively, and then combine these solutions to create a solution to the original problem".

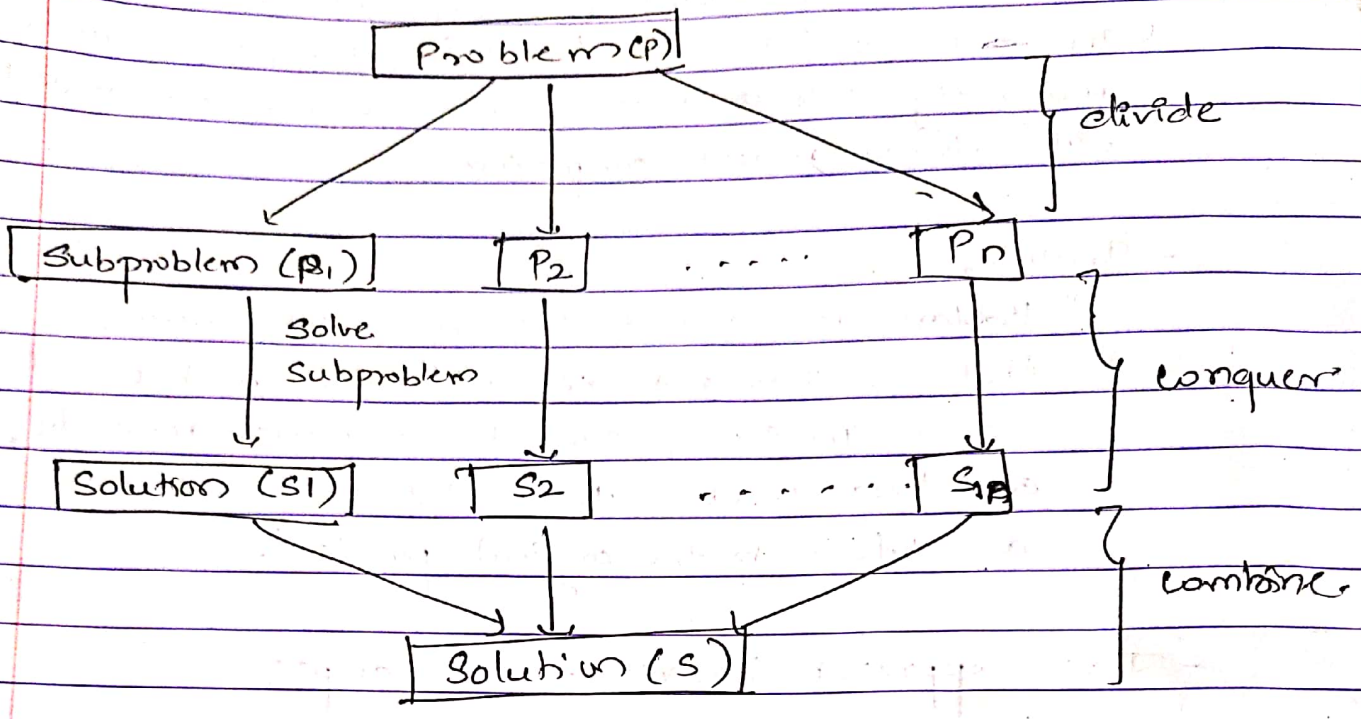
- This approach involves three steps :-

1) Divide - Problem into number of subproblems that are smaller instances of the same problem.

2) Conquer - solve the subproblems by solving them recursively. If subproblem is small just solve it in a straightforward manner.

3) Combine - Join the solutions of subproblems to produce solution to the original problem.

Example — Merge sort, Quick Sort, Binary Search, Min Max Array, Strassens Matrix, Convex hull etc.



- General Algorithm structure

```
Algorithm DAndC (P)
{
1.    if Small (P) then return S(P);
2.    else
      {
3.       divide P into smaller instances P1, P2 ... Pk ; k>=1
4.       Apply DAndC to each subproblems.
5.       return Combine ( S(P1); S(P2) .... S(Pk));
      }
}
```

II] Analysis of Merge Sort

a) Merge Sort

- This algorithm follows divide and conquer approach.
- Divide : Divides n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- Conquere : Sort the two subseque recursively.
- Combine : Merge the two sorted sequence to produce the sorted answer.

Algorithm :

i) MERGE PROCEDURE

i/p : low, mid, high indices of a global array a[], where a[] has two sorted subsequences a[low .... mid] and a[mid+1 ... high]

o/p : Sorted sequence a[low ... high]

Algorithm uses an auxiliary global array b[].

```
MERGE (low, mid, high)
{
1.    h = low; i = low; j = mid+1;

2.    while((h ≤ mid) and (j ≤ high)) do
          // Both condition ensures none of the array is completed.
      {
3.        if (a[h] ≤ a[j]) then
4.            b[i++] = a[h++];
5.        else    b[i++] = a[j++];
      }
6.    if (h > mid) then    // First array is saturated
7.        for k=j to high do    } → copy remaining
8.            b[i++] = a[k++]   } elements of 2nd array.

9.    else                 // 2nd array is saturated
10.       for k=h to mid do     } → copy remaining
11.           b[i++] = a[k++]   } elements of 1st array.

12.   for k=low to high         } → copying from aux array
13.       a[k] = b[k]           } to orginal array.
}
```

(ii) MERGE SORT PROCEDURE

i/p : low, high - indices of unsorted global array a[low, ... high]

o/p : Sorted sequence a[low ... high]

```
MergeSort (low, high)
{
1.    if (low < high) then
2.    {
3.        mid = ⌊(low + high)/2⌋;
4.        MergeSort (low, mid);
5.        MergeSort (mid+1, high);
          Merge (low, mid, high)
      }
}
```

(iii) Initial call to MergeSort will be @ MergeSort(1, a.length)

(b) Analysis of Merge Sort

Divide : It computes middle index takes constant time ie $\Theta(1)$.

Conquer : In this, we recursively solves 2 subproblems of size n/2, which contributed to $2T(n/2)$ to the running time.

Combine : The MERGE Procedure takes $\Theta(n)$ times where $n = high - low + 1$, since loops will take 'n' iterations.

∴ Recurrence for the running time T(n) of merge sort is,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solving this recurrence using Master Method

Step 1 : comparing with standard form.

i.e $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$a = 2 \quad b = 2 \quad f(n) = n$

$\log_b a = \log_2 2 = 1$

$n^{\log_b a} = n^1$

Step 2 : Identifying case.

$f(n) = n$

$= n^{\log_b a}$

$= \Theta(n^{\log_b a})$

By case 2,

$T(n) = \Theta(n^{\log_b a} \log n)$

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

Example :

a[] = { 5, 4, 6, 1, 3, 8, 2, 7 }