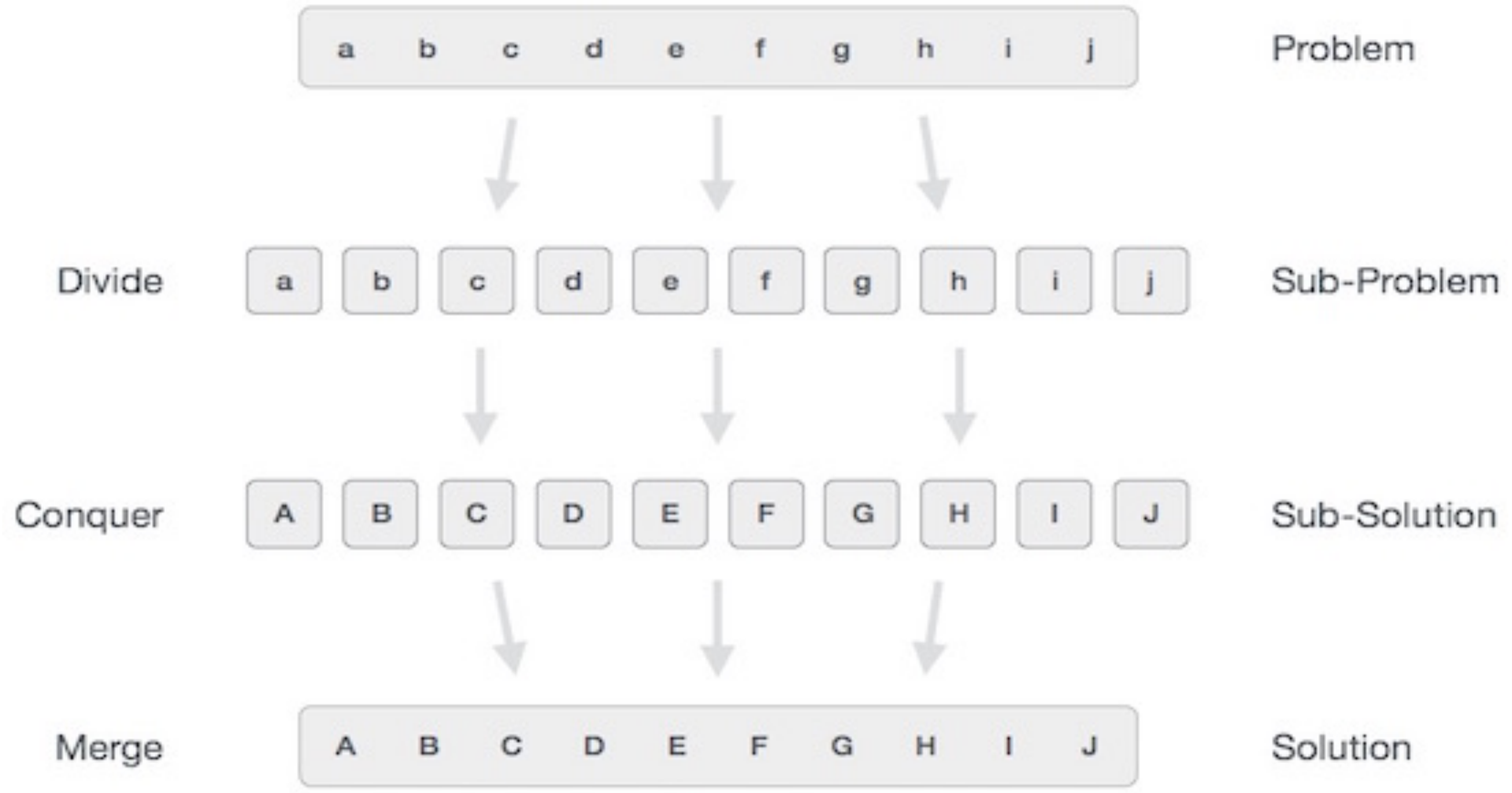
UNIT-2

- Introduction
- Divide and Conquer
- Maximum Subarray Problem

Introduction

- In divide and conquer method, the problem is divided into smaller sub-problems and then each problem is solved independently.
- When the subproblems is divided into even smaller sub-problems, it should reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Divide-and-conquer method in a three-step process:



Three parts of Divide-and-conquer method:

Divide: This involves dividing the problem into some sub problem.

Conquer: Sub problem by calling recursively until sub problem solved.

Combine: The Sub problem Solved so that we will get find problem solution.

The following are some standard algorithms that follows Divide and Conquer algorithm.

1. Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of the middle element, else recurs for the right side of the middle element.
2. Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3.Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4.Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.

5.Strassen's Algorithm is an algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3).

Divide And Conquer 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)
}
```

Recurrence Relation for DAC algorithm :

This is recurrence relation for above program.

O(1) if n is small

$T(n) = f1(n) + 2T(n/2) + f2(n)$

Example:

To find the maximum in a given array.

Input: { 100,70, 60, 50, 80, 10, 1}

Output: The maximum number in a given array is : 100

For Maximum:

The recursive approach is used to find maximum where only two elements are left using condition i.e. if(a[index]>a[index+1].)

In a program line a[index] and a[index+1]) condition will ensure only two elements in left.

```
if(index >= l-2)
{
if(a[index]>a[index+1])
{
// (a[index] // the last element will be maximum in a given array.
}
else
{
//(a[index+1] // last element will be maximum in a given array.
} }
```

Recursive function to check the right side at the current index of an array.

```
max = DAC_Max(a, index+1, l); // Recursive call


// Right element will be maximum.
if(a[index]>max)
return a[index];
// max will be maximum element in a given array.
else
return max;
}
```

• Maximum Subarray Problem

Generic problem:

Find a maximum subarray of A[low...high] with initial call: low = 1 and high = n

DC strategy:

1. Divide A[low...high] into two subarrays of as equal size as possible by finding the midpoint mid
2. Conquer:
   (a) finding maximum subarrays of A[low...mid] and A[mid + 1...high]
   (b) finding a max-subarray that crosses the midpoint
3. Combine: returning the max of the three

Correctness:

This strategy works because any subarray must either lie entirely in one side of midpoint or cross the midpoint.

- <u>Maximum Subarray Problem</u>

Given an array **arr[ ]** of integers, the task is to find the maximum sum sub-array among all the possible sub-arrays.
Examples:

Input: *arr[ ] = {-2, 1, -3, 4, -1, 2, 1, -5, 4}*
Output: *6*
{4, -1, 2, 1} is the required sub-array.
Input: *arr[ ] = {2, 2, -2}*
Output: *4*

• Maximum Subarray Problem

Divide and conquer algorithms generally involves dividing the problem into sub-problems and conquering them separately. For this problem, a structure is used to store the following values:

1. Total sum for a sub-array.
2. Maximum prefix sum for a sub-array.
3. Maximum suffix sum for a sub-array.
4. Overall maximum sum for a sub-array.(This contains the max sum for a sub-array).

- <u>Maximum Subarray Problem</u>

  - During the recursion(Divide part) the array is divided into 2 parts from the middle. The left node structure contains all the above values for the left part of array and the right node structure contains all the above values. Having both the nodes, now we can merge the two nodes by computing all the values for resulting node.
  - The max prefix sum for the resulting node will be maximum value among the maximum prefix sum of left node or left node sum + max prefix sum of right node or total sum of both the nodes (which is possible for an array with all positive values).

• Maximum Subarray Problem

- Similarly the max suffix sum for the resulting node will be maximum value among the maximum suffix sum of right node or right node sum + max suffix sum of left node or total sum of both the nodes (which is again possible for an array with all positive values).
- The total sum for the resulting node is the sum of both left node and right node sum. Now, the max subarray sum for the resulting node will be maximum among prefix sum of resulting node, suffix sum of resulting node, total sum of resulting node, maximum sum of left node, maximum sum of right node, sum of maximum suffix sum of left node and maximum prefix sum of right node.

- <u>Maximum Subarray Problem</u>

    - **Time Complexity:** The recursive function generates the following recurrence relation.
      T(n) = 2 * T(n / 2) + O(1)