Design and Analysis of Algorithm

Course Instructor

Pramod Mane, PhD

Faculty, Dept. CSE, NIT Raipur

BASIC TECHNIQUES FOR DESIGN OF EFFICIENT ALGORITHMS

There are basically 5 fundamental techniques which are used to design an algorithm efficiently:

1. **Divide-and-Conquer**
2. **Greedy method**
3. **Dynamic Programming**
4. **Backtracking**
5. **Branch-and-Bound**

1. DIVIDE-AND-CONQUER

Divide & conquer technique is a top-down approach to solve a problem.

The algorithm which follows divide and conquer technique involves 3 steps:

1. **Divide the original problem into a set of sub problems.**

2. **Conquer (or Solve) every sub-problem individually, recursive.**

3. **Combine the solutions of these sub problems to get the solution of original problem.**

GREEDY METHOD

- Greedy technique is used to solve an optimization problem.
- An Optimization problem is one in which, we are given a set of input values, which are required to be either maximized or minimized (known as objective function) w. r. t. some constraints or conditions.
- Greedy algorithm always makes the choice (greedy criteria) that looks best at the moment, to optimize a given objective function.
- That is, it makes a locally optimal choice in the hope that this choice will lead to a overall globally optimal solution.

GREEDY METHOD

- The greedy algorithm does not always guaranteed the optimal solution but it generally produces solutions that are very close in value to the optimal.

DYNAMIC PROGRAMMING

- Dynamic programming technique is similar to divide and conquer approach.

- Both solve a problem by breaking it down into a several sub problems that can be solved recursively.

- The difference between the two is that in dynamic programming approach, the results obtained from solving smaller sub problems are reused (by maintaining a table of results) in the calculation of larger sub problems.

DYNAMIC PROGRAMMING

- Thus dynamic programming is a Bottom-up approach that begins by solving the smaller sub-problems, saving these partial results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained.

- Reusing the results of sub-problems (by maintaining a table of results) is the major advantage of dynamic programming because it avoids the re-computations (computing results twice or more) of the same problem.

- Thus Dynamic programming approach takes much less time than naïve or straightforward methods, such as divide-and-conquer approach which solves problems in top-down method and having lots of re-computations.

DYNAMIC PROGRAMMING

- The dynamic programming approach always gives a guarantee to get a optimal solution.

BACKTRACK

- The term “backtrack” was coined by American mathematician D.H. Lehmer in the 1950s.

- Backtracking can be applied only for problems which admit the concept of a “partial candidate solution” and relatively quick test of whether it can possibly be completed to a valid solution.

- Backtrack algorithms try each possibility until they find the right one. It is a depth-first-search of the set of possible solutions.

BACKTRACK

- During the search, if an alternative doesn’t work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative.

- When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there.

- If there are no more choice points, the search fails.

BACKTRACK

- Branch-and-Bound (B&B) is a rather general optimization technique that applies where the greedy method and dynamic programming fail.

- B&B design strategy is very similar to backtracking in that a state-space-tree is used to solve a problem.

- Branch and bound is a systematic method for solving optimization problems. However, it is much slower.

BACKTRACK

- Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.

- The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated).

- Rather, a carefully selected criterion determines which node to expand and when, and another criterion tells the algorithm when an optimal solution has been found.

1. DIVIDE-AND-CONQUER

Divide & conquer technique is a top-down approach to solve a problem.

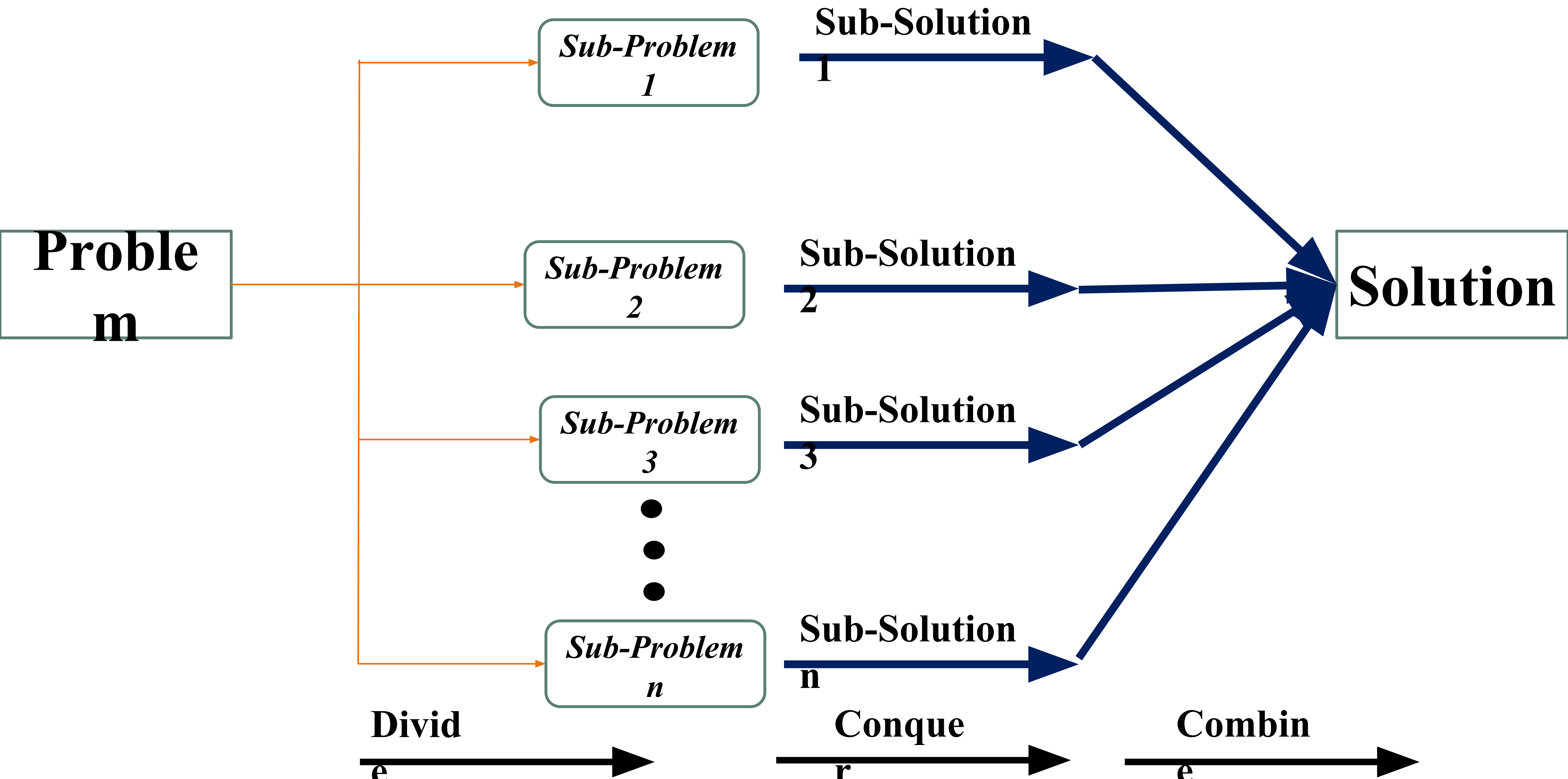
The algorithm which follows divide and conquer technique involves 3 steps:

1. **Divide the original problem into a set of sub problems.**

2. **Conquer (or Solve) every sub-problem individually, recursive.**

3. **Combine the solutions of these sub problems to get the solution of original problem.**

1. DIVIDE-AND-CONQUER



1. DIVIDE-AND-CONQUER

Step 1: **Divide** the given big problem into a number of sub-problems that are similar to the original problem but smaller in size. A sub-problem may be further divided into its sub-problems. A Boundary stage reaches when either a direct solution of a sub-problem at some stage is available or it is not further sub-divided. When no further sub-division is possible, we have a direct solution for the sub-problem.

Step 2: **Conquer** (Solve) each solutions of each sub-problem (independently) by recursive calls; and then

Step 3: **Combine** the solutions of each sub-problems to generate the solutions of original problem.

1. DIVIDE-AND-CONQUER

An algorithms which follow the divide-and-conquer strategy have the following recurrence form:

T(n)=

Where

1. T(n) is running time for problem size n

2. If the problem size is small enough (say, $n \leq c$ for some constant c), we have a base case.

1. DIVIDE-AND-CONQUER

3. The brute-force (or direct) solution takes constant time: $\Theta(1)$

4. Otherwise, suppose that we divide into *a* sub-problems, each 1/b of the size of the original problem of size n.

5. Suppose each sub-problem of size n/b takes time to solve and since there are a sub-problems so we spend total time to solve sub-problems.

6. D(n) is the cost(or time) of dividing the problem of size n.

7. C(n) is the cost (or time) to combine the sub-solutions.