राष्ट्रीय प्रौद्योगिकी संस्थान दिल्ली

National Institute of Technology Delhi

(An autonomous Institute under the aegis of Ministry of HRD, Govt. of India)

DAA Assignment

Submitted by:

Amrit Raj
181210008
CSE 2nd Year

Utam Kumar
181210055
CSE 2nd Year

Submitted to:
Chandresh Kumar Maurya

Contents

- Divide and Conquer
- Backtracking
- Greedy
- Dynamic Programming
- Branch and Bound

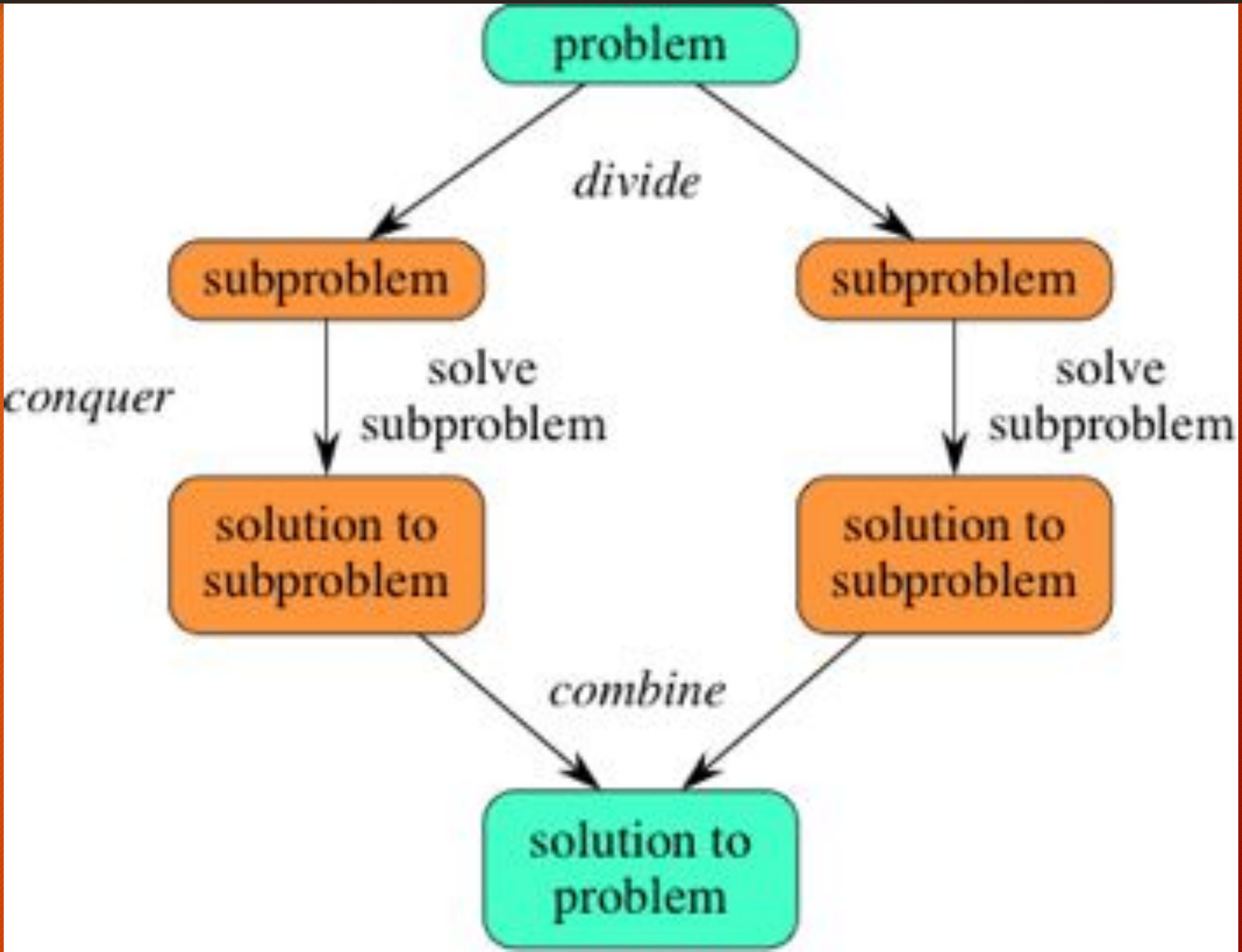
Divide and Conquer

Divide and Conquer is an algorithm design based on multi-branched recursion. Algorithm breaks problem into two or more smaller problems, until these become simple enough to solve directly

The most well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances.
2. Solve smaller instances recursively
3. Obtain solution to original(larger) instance by combining these solutions.

Divide and Conquer Technique



Divide and Conquer Examples

- Searching – Binary Search
- Sorting – Merge Sort
- Sorting – Quick Sort
- Tree Traversal

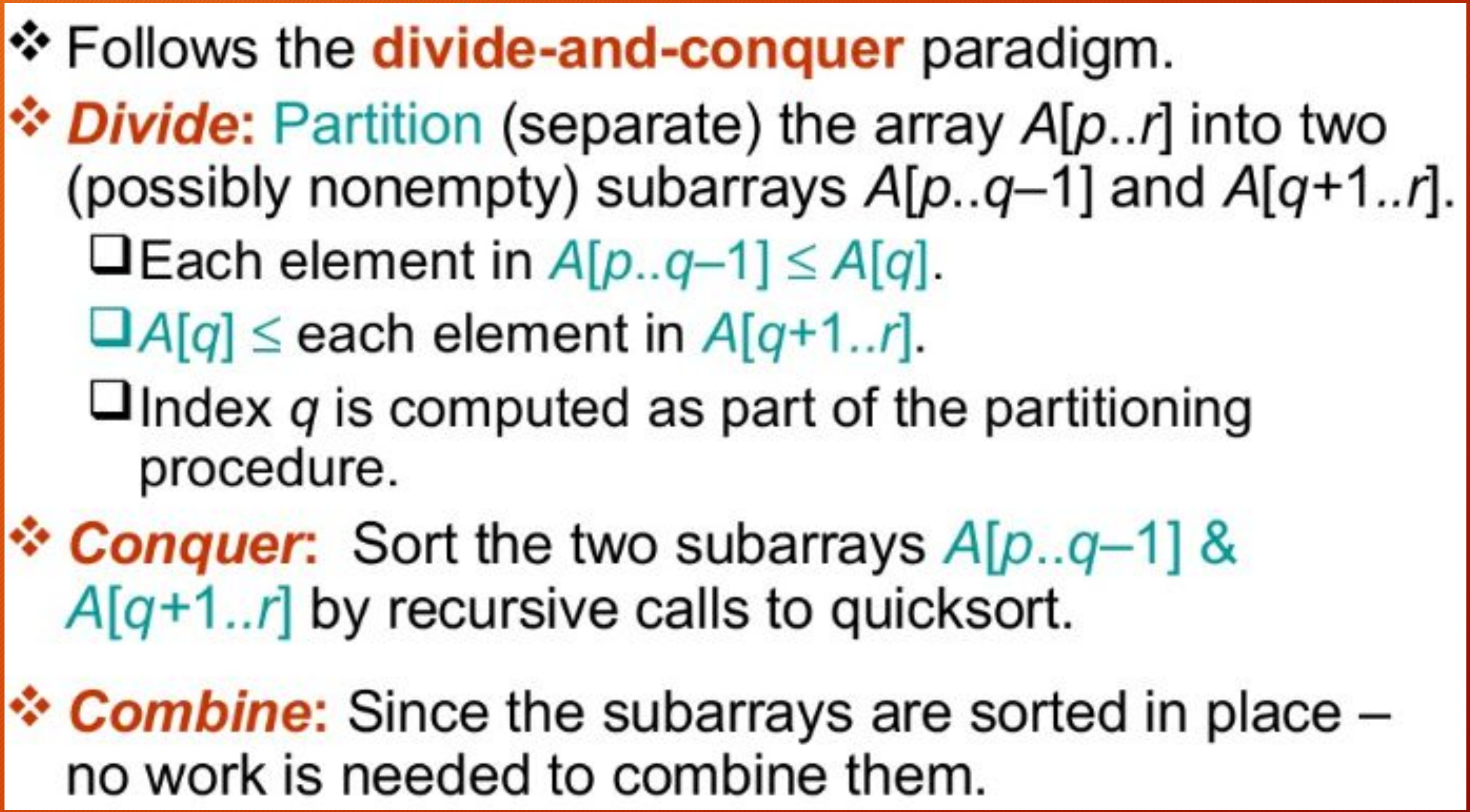
Divide and Conquer Recurrence: Master Theorem

$T(n) = aT(n/b) + f(n)$ where $f(n)=\Theta(n^k)$

1. $a < b^k$ $T(n) \in \Theta(n^k)$
2. $a = b^k$ $T(n) \in \Theta(n^k \log n)$
3. $a > b^k$ $T(n) \in \Theta(n^{\log b/\log a})$

Same results hold with $O(n)$ and $\Omega(n)$.

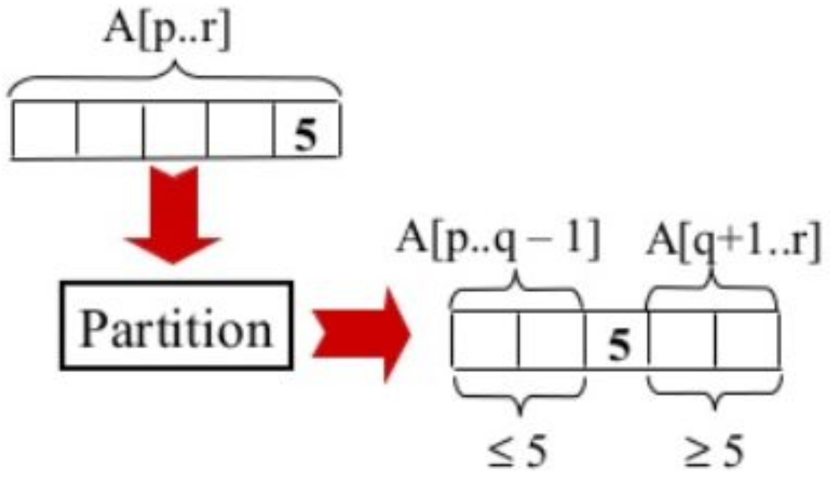
Quick Sort



- Follows the **divide-and-conquer** paradigm.
- ***Divide*:** Partition (separate) the array $A[p..r]$ into two (possibly nonempty) subarrays $A[p..q-1]$ and $A[q+1..r]$.
  - Each element in $A[p..q-1] \leq A[q]$.
  - $A[q] \leq$ each element in $A[q+1..r]$.
  - Index q is computed as part of the partitioning procedure.
- ***Conquer*:** Sort the two subarrays $A[p..q-1]$ & $A[q+1..r]$ by recursive calls to quicksort.
- ***Combine*:** Since the subarrays are sorted in place – no work is needed to combine them.

Quick Sort Pseudo Code

```
Quicksort(A, p, r)
    if $p < r$ then
        q := Partition(A, p, r);
        Quicksort(A, p, $q - 1$);
        Quicksort(A, $q + 1$, r)
    fi
```



```
Partition(A, p, r)
    x:= A[r],
    i :=p – 1;
    for j := p to $r - 1$ do
        if $A[j] \le x$ then
            i := $i + 1$;
            $A[i] \leftrightarrow A[j]$
        fi
    od;
    $A[i + 1] \leftrightarrow A[r]$;
    return $i + 1$
```

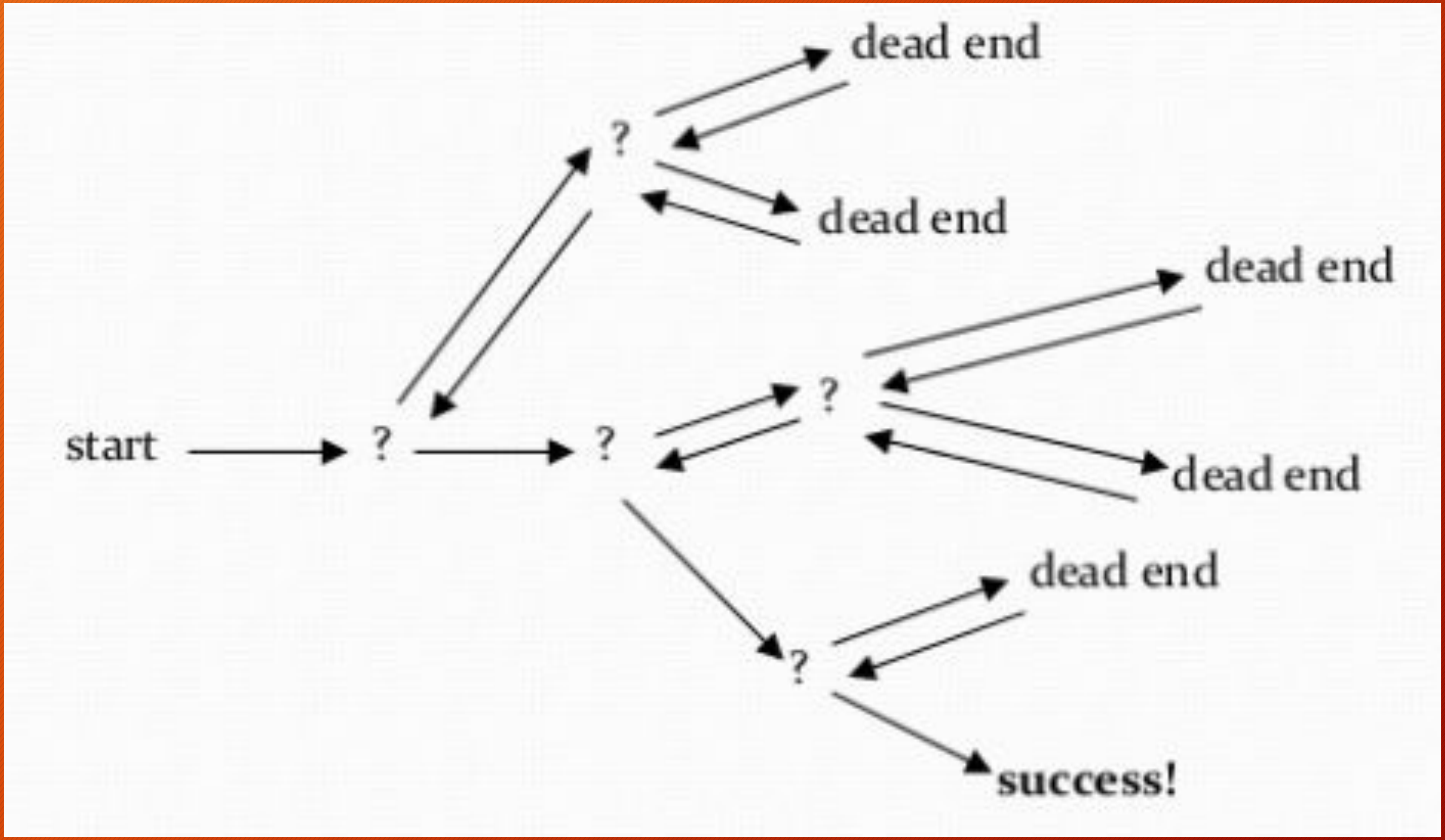
Backtracking

The principle of backtracking is to construct symbols as component at a time. And, then evaluate such partially constructed solutions.

Backtracking is a methodological way of trying out various sequences of decisions, until you find one that “works”.

Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead nodes, we go back (“backtrack”) to the node’s parent and proceed with the search on the next child.

Backtracking Technique

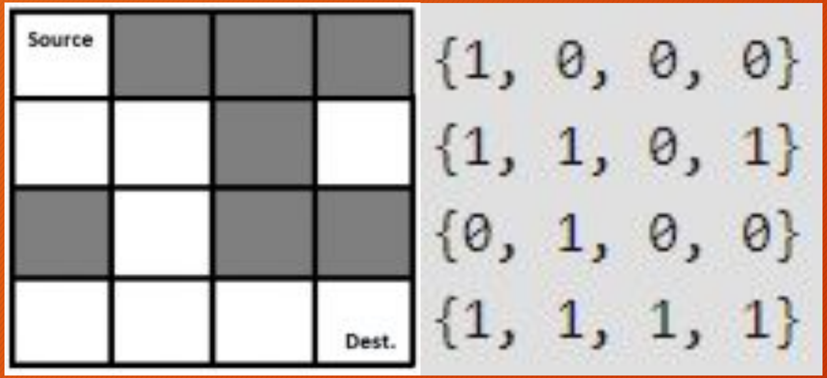


Backtracking Examples

- N-queen problem: Try to place N queens on an N * N board such that none of the queens can attack another queen
- Rat in a maze: A rat has to go from one corner(source) to opposite corner(destination) while some of the grids in the maze are dead end.
- The Knight's tour problem: The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.
- Sum of Subsets: Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K.

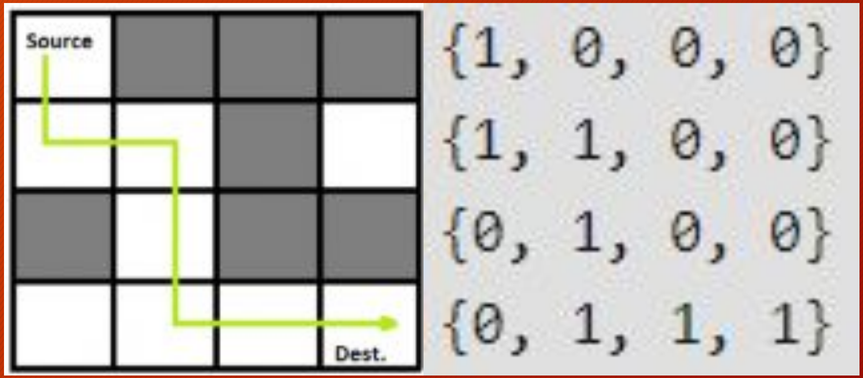
Rat in a maze problem

Input:
This algorithm will take the maze as a matrix.
In the matrix, the value 1 indicates the free space and 0 indicates the wall or blocked area.



```
{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}
```

INPUT



```
{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}
```

OUTPUT

Output:
It will display a matrix. From that matrix, we can find the path of the rat to reach the destination point.

Rat in a maze algorithm

```
isValid(x, y)
Input: x and y point in the maze.
Output: True if the (x,y) place is valid, otherwise false.
Begin
    if x and y are in range and (x,y) place is not blocked, then
        return true
    return false
End
```

```
solveRatMaze(x, y)
Input: The starting point x and y.
Output: The path to follow by the rat to reach the destination, otherwise false.
Begin
    if (x,y) is the bottom right corner, then
        mark the place as 1
        return true
    if isValidPlace(x, y) = true, then
        mark (x, y) place as 1
        if solveRatMaze(x+1, y) = true, then //for forward movement
            return true
        if solveRatMaze(x, y+1) = true, then //for down movement
            return true
        mark (x,y) as 0 when backtracks
        return false
    return false
End
```

Greedy

- Divide the concept into multiple steps(sub-problems).
- For each step, take best choice at the moment(Local Optimal)

  (Greedy Choice).
- A greedy algorithm always makes the choice that looks best at the moment.
- Hope: A local optimal choice will lead to a globally optimal solution.
  - For some problem, it works. For others, it don’t.

Greedy Algorithm

- Select the activity that ends first (smallest end time)
  - Intuition: it leaves the largest possible empty space for more activities

Greedy Choice: Select the next best activity (Local Optimal)

- Once selected an activity
  - Delete all non-compatible activities
  - They cannot be selected

- Repeat the algorithm for the remaining activities
  - Either using iterations or recursion

Sub-problem: We created one sub-problem to solve (Find the optimal schedule after the selected activity)

Greedy Algorithm Examples

- Activity Selection Problem: *Maximize* the number of jobs using a resource
- Huffman Encoding Problem: Encode the data in a file to *minimize* its size
- Knapsack Problem: Collect the *maximum* value of goods that fit in a given bucket
- Minimum Spanning Tree: Select the *smallest-weight* of edges to connect all nodes in a graph

Knapsack Problem

- Thief has a knapsack with maximum capacity W, and a set S consisting of n items.
- Each item i has some weight w_i and benefit value v_i (all w_i , v_i and W are integer values).
- Two types of knapsack problems:
  - 0-1 Knapsack: Items cannot be divided(Take it or leave it).
  - Fractional Knapsack: Items can be divided.

Knapsack Problem Algorithm

0-1 Knapsack:

- find x_i such that for all $x_i = \{0, 1\}$,

  $i = 1, 2, .., n$

  $\sum w_i x_i \leq W$ and

  $\sum x_i v_i$ is maximum

If $X_i = 1$, then item i will be taken

If $X_i = 0$, then item i will be skipped

Fractional Knapsack:

- find x_i such that for all $0 <= x_i <= 1$,

  $i = 1, 2, .., n$

  $\sum w_i x_i \leq W$ and

  $\sum x_i v_i$ is maximum

If $X_i = 0$, then item i will be skipped

If $X_i > 0$, then X_i fraction of item i will be taken

Dynamic Programming

Also known as dynamic optimization is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution.

Dynamic programming Algorithm

- Characterize *optimal substructure*
- *Recursively* define the value of an optimal solution
- Compute the value *bottom up*
- (if needed) *Construct* an optimal solution

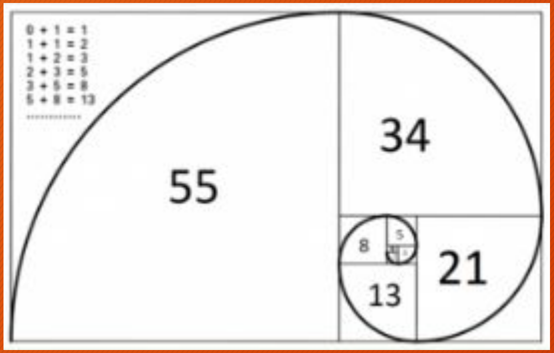
Dynamic programming Examples

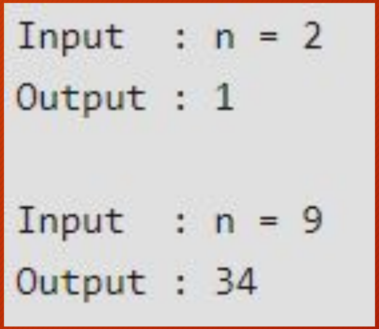
1. Fibonacci Numbers: Given an number n, print n’th Fibonacci Number.
2. Friend pairing problem: Given n friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.
3. Ugly Numbers: Given a number n, the task is to find n’th Ugly Number.
4. Largest divisible pairs subset: Given an array of n distinct elements, find length of the largest subset such that every pair in the subset is such that the larger element of the pair is divisible by smaller element.

Fibonacci Number Problem

The Fibonacci numbers are the numbers in the following integer sequence.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation





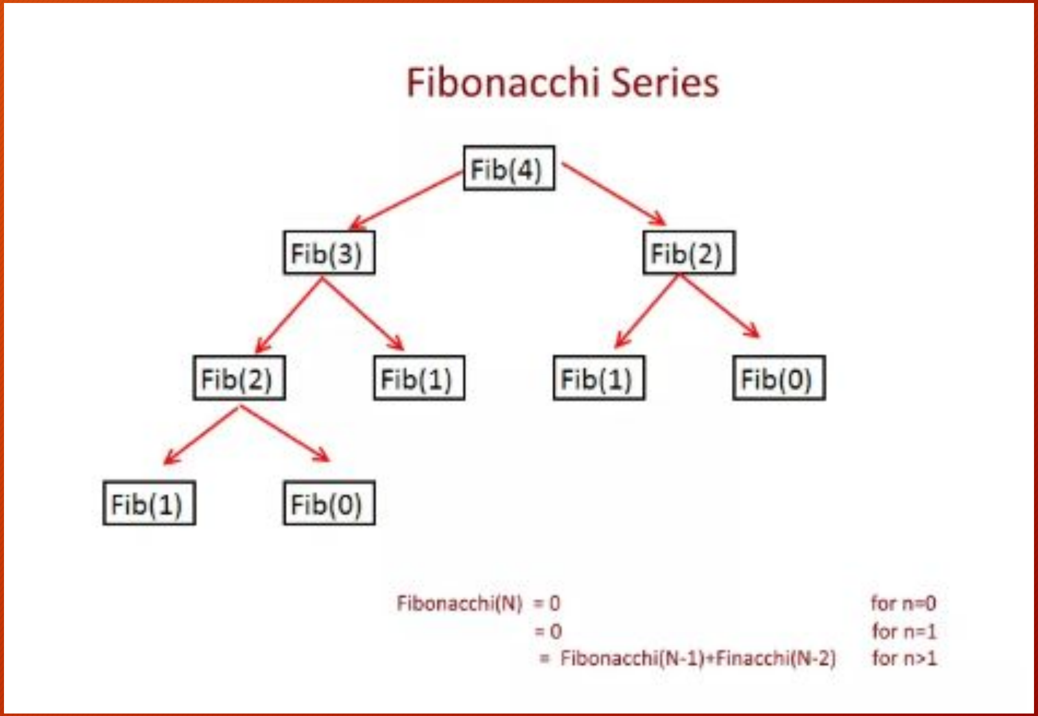
```
Input  : n = 2
Output : 1

Input  : n = 9
Output : 34
```

If n = 0, then fib() should return 0. If n = 1, then it should return 1. For n > 1, it should return Fn-1 + Fn-2

Fibonacci Number Problem Algorithm

```
int memo[] = empty
int fib(n) {
if(memo[n] != empty)
return memo[n];
if(n==1 || n==2)
return 1;
memo[n] = fib(n-1)+fib(n-2);
return memo[n];
}
```



Branch And Bound

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

Branch-and-bound usually applies to those problems that have finite solutions, in which the solutions can be represented as a sequence of options. The first part of branch-and-bound branching requires several choices to be made so that the choices will branch out into the solution space. In these methods, the solution space is organized as a treelike structure. Branching out to all possible choices guarantees that no potential solutions will be left uncovered. But because the target problem is usually NP-complete or even NP-hard, the solution space is often too vast to traverse.

Branch And Bound

An important advantage of branch-and-bound algorithms is that we can control the quality of the solution to be expected, even if it is not yet found

Branch And Bound Algorithm

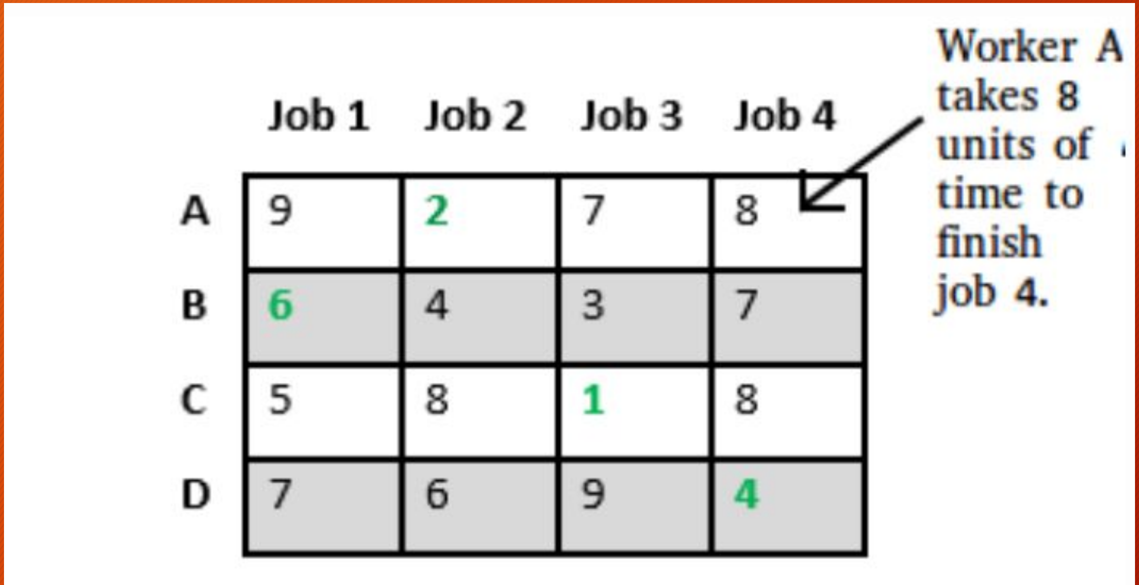
- Search the tree using a breadth -first *(FIFO branch and bound)*.
- search the tree as in a bfs, but replace the FIFO queue with a stack *(LIFO branch and bound)*.
- Replace the FIFO queue with a priority queue *(least-cost (or max priority) branch and bound)*. The priority of a node p in the queue is based on an estimate of the likelihood that the answer node is in the subtree whose root is p.
- Space required is *O(number of leaves)*.
- For some problems, solution are at different levels of the tree (e.g., 16 puzzle).

Branch And Bound Algorithm Examples

- 0/1 Knapsack using Branch and Bound
- Implementation of 0/1 Knapsack using Branch and Bound
- 8 puzzle Problem using Branch And Bound
- Job Assignment Problem using Branch And Bound
- N Queen Problem using Branch And Bound
- Traveling Salesman Problem using Branch And Bound

Job Assignment Problem

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized



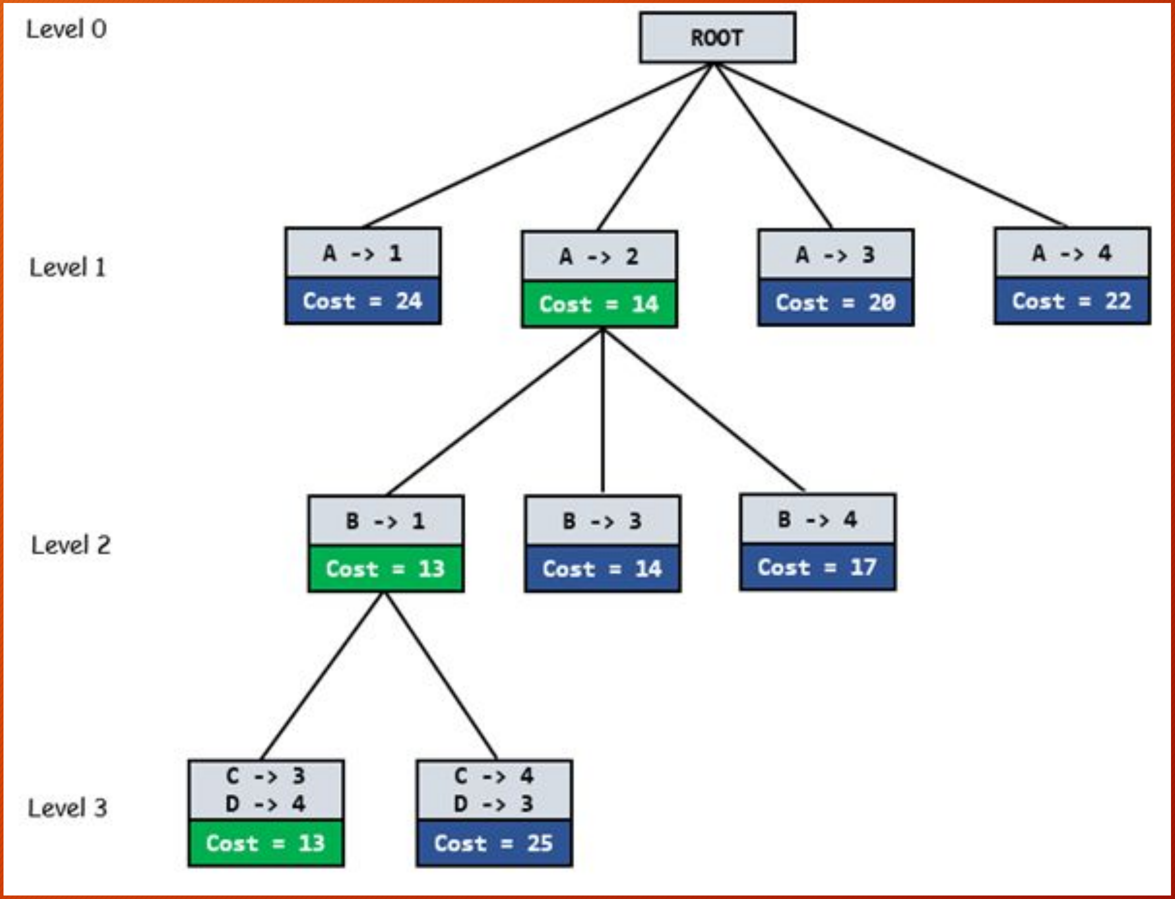
	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Worker A takes 8 units of time to finish job 4.

An example job assignment problem. Green values show optimal job assignment that is A-Job4, B-Job1 C-Job3 and D-Job4

Job Assignment Problem

Below diagram shows complete search space diagram showing optimal solution path in green



Job Assignment Algorithm

```
/* findMinCost uses Least() and Add() to maintain the
   list of live nodes

   Least() finds a live node with least cost, deletes
   it from the list and returns it

   Add(x) calculates cost of x and adds it to the list
   of live nodes

   Implements list of live nodes as a min heap */

// Search Space Tree Node
node
{
   int job_number;
   int worker_number;
   node parent;
   int cost;
}
// Input: Cost Matrix of Job Assignment problem
// Output: Optimal cost and Assignment of Jobs
```

Job Assignment Algorithm

```
algorithm findMinCost (costMatrix mat[][])
{
   // Initialize list of live nodes(min-Heap)
   // with root of search tree i.e. a Dummy node
   while (true)
   {
      // Find a live node with least estimated cost
      E = Least();
      // The found node is deleted from the list
      // of live nodes
      if (E is a leaf node)
      {
         printSolution();
         return;
      }
     for each child x of E
     {
         Add(x); // Add x to list of live nodes;
         x->parent = E; // Pointer for path to root
     }
   }
}
```