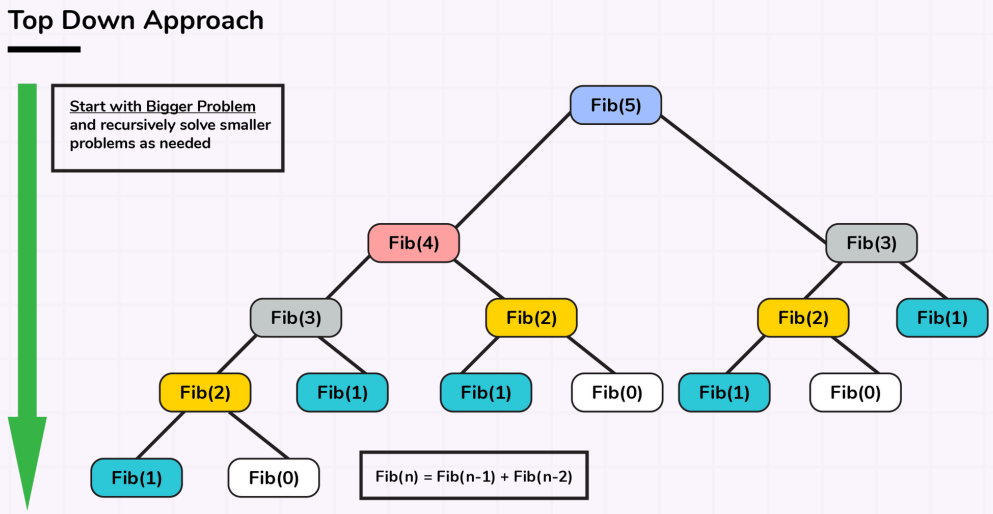
InterviewCafe
Where Learning Never Takes a Break!

Top Dynamic Programming Interview Questions & Explanations



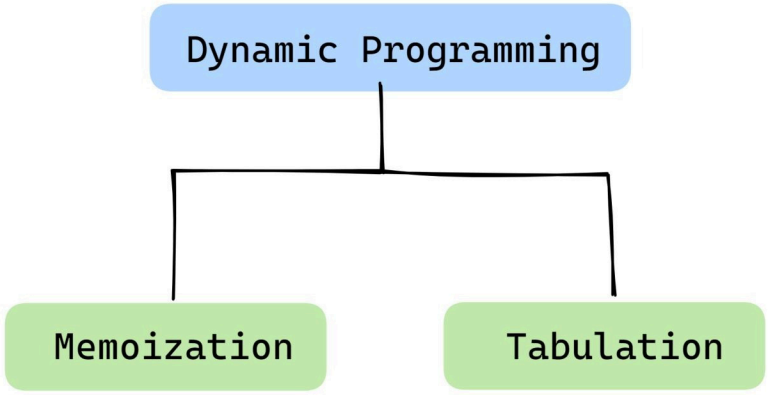
Santosh Kumar Mishra
Software Engineer at Microsoft • Author • Founder of InterviewCafe

Table of Contents

Dynamic Programming Intro.

- Dynamic Programming (DP) is a technique used to solve problems by breaking them down into smaller subproblems and reusing the solutions of those subproblems to solve the larger problem. DP is used when a problem has overlapping subproblems and optimal substructure. Overlapping subproblems mean that the solution to a subproblem can be used in multiple places within the problem. Optimal substructure means that the optimal solution to the problem can be constructed from the optimal solutions to its subproblems.

→ Techniques to solve Dynamic Programming Problems:



→ There are two different ways to store the values so that the values of a sub-problem can be reused. Here, will discuss two patterns of solving dynamic programming (DP) problems:

1. Tabulation: Bottom Up

2. Memoization: Top Down

- The key difference between tabulation and memoization is the order in which the subproblems are solved. In tabulation, the subproblems are solved in a specific order, starting from the smallest subproblems and building up to the largest subproblems. In memoization, the subproblems are solved in a more flexible order, based on which subproblems are needed to solve the current subproblem.

- In general, tabulation is more efficient for problems where the subproblems have a clear and natural order, while memoization is more efficient for problems where the subproblems can be solved in any order and there are many overlapping subproblems.

in Santosh Kumar Mishra

➜ How to solve a Dynamic Programming Problem?

- Overlapping Subproblems:

  When the solutions to the same subproblems are needed repetitively for solving the actual problem. The problem is said to have overlapping subproblems property.

- Optimal Substructure Property:

  If the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems then the problem is said to have Optimal Substructure Property.

➜ Steps to solve a Dynamic programming problem:

1. Identify if it is a Dynamic programming problem.
2. Decide a state expression with the Least parameters.
3. Formulate state and transition relationships.
4. Do tabulation (or memorization).

Fibonacci Number

→ The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Given n, calculate F(n).

```
Input: n = 2
Output: 1
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.
```

Approach 1: Recursion

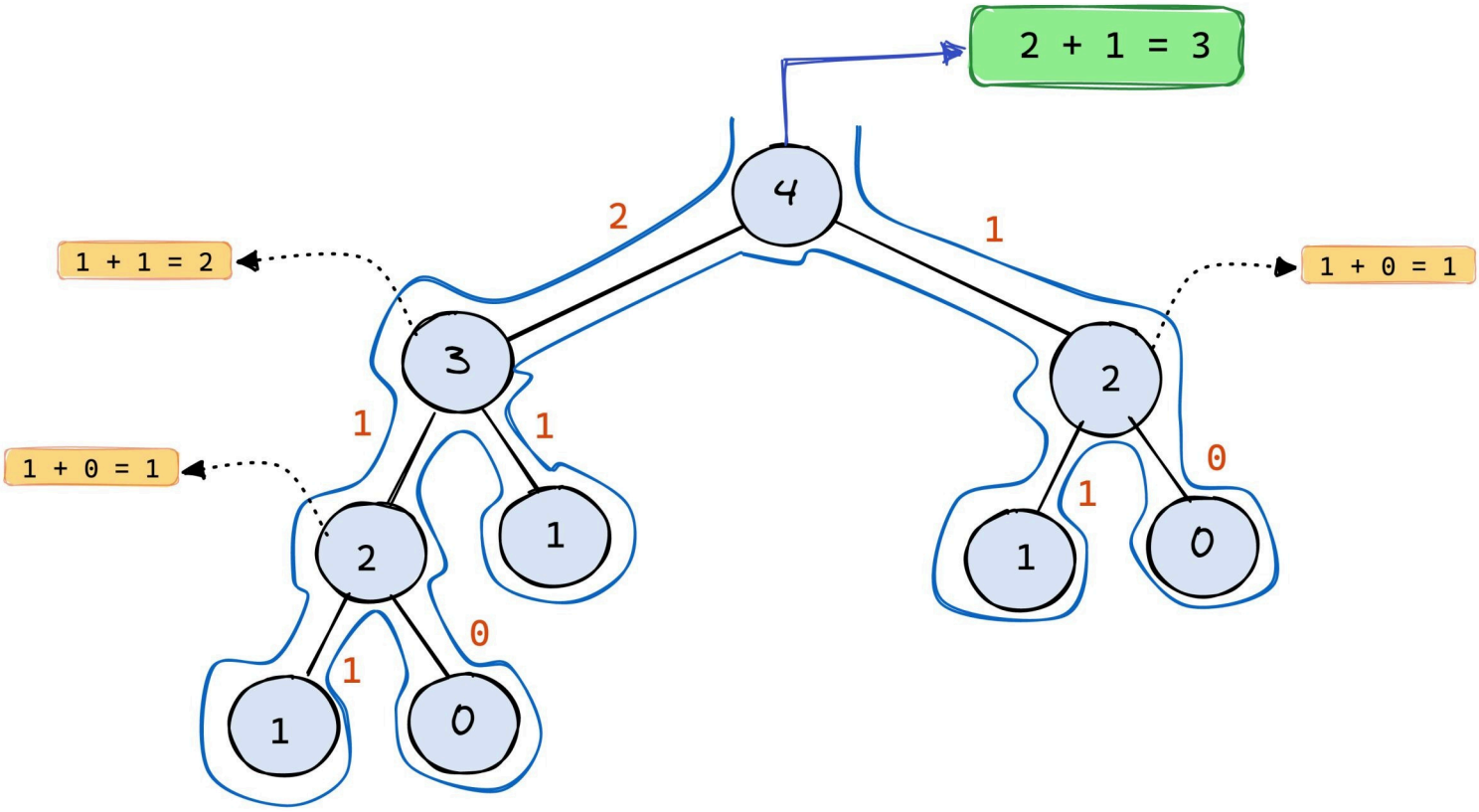
→ A simple method that is a direct recursive implementation mathematical recurrence relation is given above.

→ Check if the provided input value, N, is less than or equal to 1. If true, return N.

→ Otherwise, the function fib(int N) calls itself, with the result of the 2 previous numbers being added to each other, passed in as the argument. This is derived directly from the

```
F(n) = F(n-1) + F(n-2)
```

Santosh Kumar Mishra




```java
class Solution {
    public int fib(int n) {

        if(n<=1){
            return n;
        }

       return fib(n-1) + fib(n-2);
    }
}
```

Time Complexity: Exponential O(2^N), as every function calls two other functions.

Extra Space: O(n) if we consider the function call stack size.

Approach 2: Recursion with Memoization

→ Notice how in the previous solution we end up re-computing the solutions to sub-problems. We could optimize this by caching our solutions using memoization.

```java
class Solution {
    public int fib(int n) {

        int[] dp = new int[n+1];
        Arrays.fill(dp,-1);
        return fibNum(n,dp);
    }

    public int fibNum(int n,int[] dp){

        if(n <= 1) return n;
        if(dp[n] != -1) return dp[n];

        int temp = fibNum(n-1,dp) + fibNum(n-2,dp);
        dp[n] = temp;
        return dp[n];
    }
}
```

- Time Complexity: O(N)
- Space Complexity : O(N)

Approach 3: Space Optimized

→ We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

```java
public class fibonacci{

    public int fib(int n){
        int a = 0, b = 1, c;
        if (n == 0)
            return a;

        for (int i = 2; i <= n; i++){
            c = a + b;
            a = b;
            b = c;
        }
        return b;
    }

    public static void main (String args[]){
        int n = 9;
        System.out.println(fib(n));
    }
}
```

- Time Complexity: O(N)
- Space Complexity : O(1)

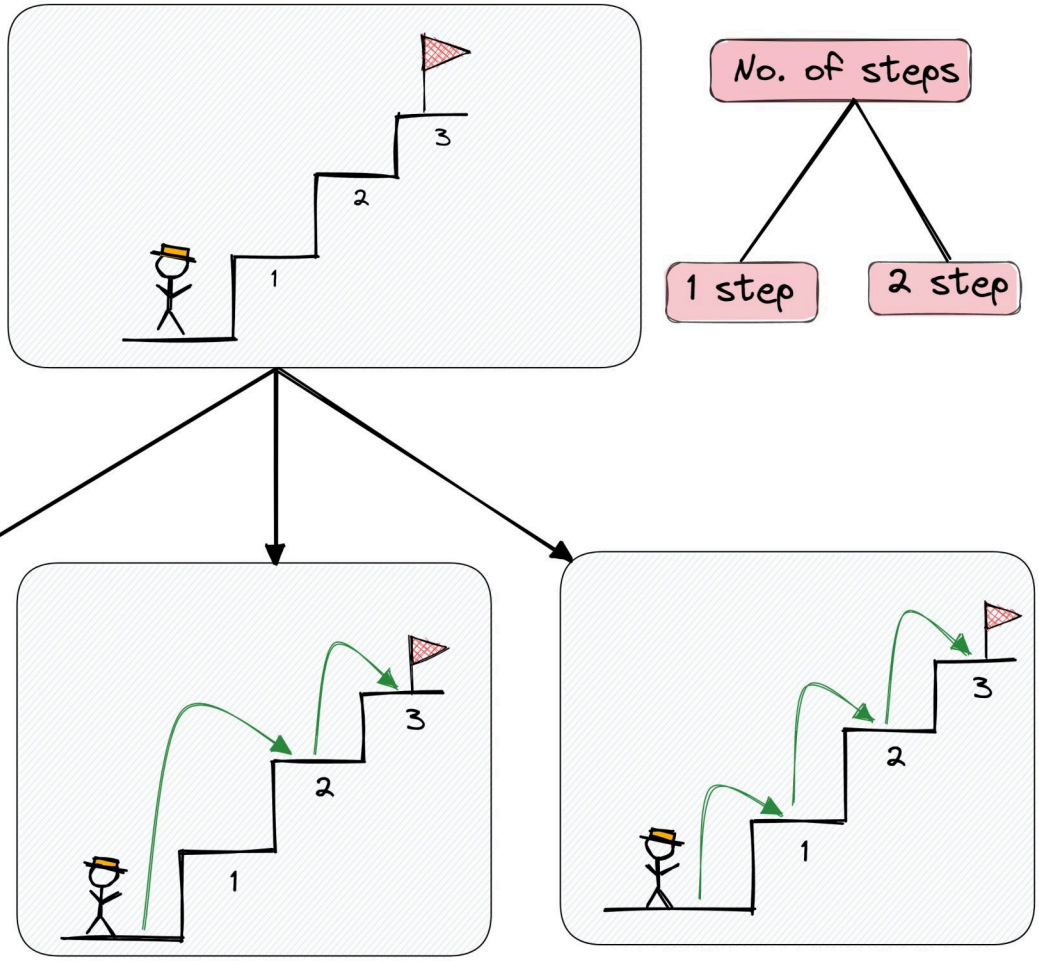
Climbing Stairs

→ You are climbing a staircase. It takes n steps to reach the top.

→ Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

```
Input: n = 2

Output: 2

Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

→ what are the number of ways to reach to the 3rd stair?



in Santosh Kumar Mishra

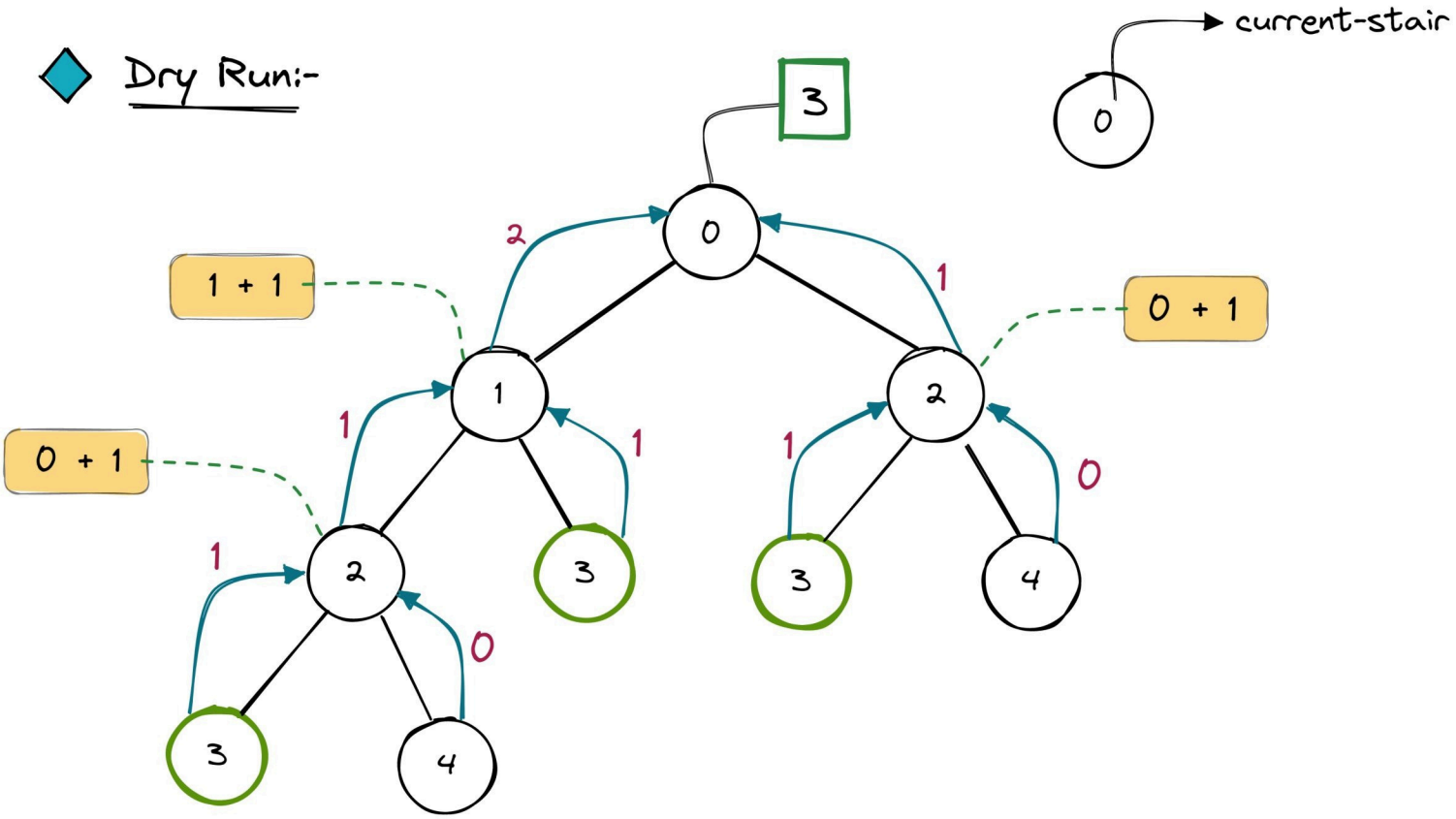
Approach-1 : Recursion

- In this brute force approach we take all possible step combinations i.e. 1 and 2, at every step.
- At every step we are calling the function climbStairs for step 1 and 2, and return the sum of returned values of both functions.

```
climbStairs(i,n) = climbStairs(i+1,n) + climbStairs(i+2,n)
```

- Where i defines the current step and n defines the destination step.

Dry Run:-



Base Conditions:-

```
if(currentStair == n)
    return 1;

if(currentStair > n)
    return 0;
```

Santosh Kumar Mishra

- Time-Complexity ⟶ O(2^n)
- Space-Complexity ⟶ O(n), The depth of the recursion tree can go upto n.

Approach 2: Recursion with Memoization

→ In the previous approach we are redundantly calculating the result for every step.

Instead, we can store the result at each step in memo `Array/Hashmap` and directly returning the result from the memo array whenever that function is called again.

→ In this way we are pruning recursion tree with the help of memo array and reducing the size of recursion tree upto n.

```java
public class Solution {
    public int climbStairs(int n) {

        int memo[] = new int[n + 1];
        return climb_Stairs(0, n, memo);
    }

    public int climb_Stairs(int i, int n, int memo[]) {

        if (i > n) return 0;
        if (i == n)  return 1;
        if (memo[i] > 0) return memo[i];

        memo[i] = climb_Stairs(i + 1, n, memo) + climb_Stairs(i + 2, n, memo);
        return memo[i];

    }
}
```

- Time-Complexity ⟶ O(n)
- Space-Complexity ⟶ O(n), The depth of the recursion tree can go upto n.

- Time-Complexity ⟶ O(2^n)
- Space-Complexity ⟶ O(n), The depth of the recursion tree can go upto n.

Approach 3: Bottom-Up Approach

→ As we can see this problem can be broken into subproblems, and it contains the optimal substructure property

One can reach ith step in one of the two ways:

1. Taking a single step from (i-1)th step.
2. Taking a step of 2 from (i-2)th step.

→ So, the total number of ways to reach ith is equal to sum of ways of reaching (i-1)th step and ways of reaching (i-2)th step.

```
dp[i] = dp[i-1] + dp[i-2]
```

```java
public class Solution {
    public int climbStairs(int n){

        if (n == 1) return 1;

        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;

        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];

    }
}
```

Santosh Kumar Mishra

- Time-Complexity ⟶ O(n), single loop upto n.
- Space-Complexity ⟶ O(n), dp array of size n is used.

Approach 4: Fibonacci Number (Space Optimization)

→ In the above approach we have used dp array where dp[i]=dp[i-1]+dp[i-2]. It can be easily analysed that dp[i] is nothing but ith fibonacci number.

Fib(n) = Fib(n-1) + Fib(n-2)

→ Now we just have to find nth number of the fibonacci series having 1 and 2 their first and second term respectively, i.e.

Fib(1) = 1 , Fib(2) = 2

```java
public class Solution {
    public int climbStairs(int n) {

        if (n == 1) {
            return 1;
        }
        int first = 1;
        int second = 2;

        for (int i = 3; i <= n; i++) {
            int third = first + second;
            first = second;
            second = third;
        }
        return second;
    }
}
```

- Time-Complexity ⟶ O(n)
- Space-Complexity ⟶ O(1)

Jump Game

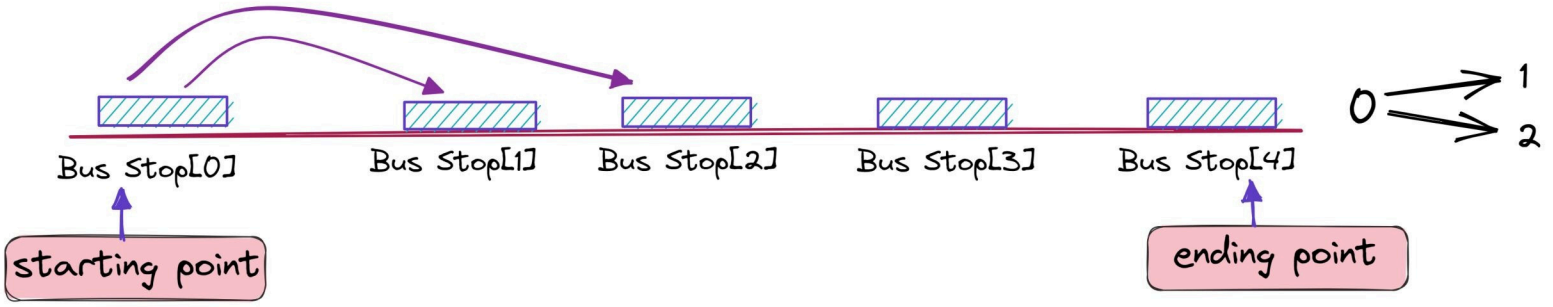
→ You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

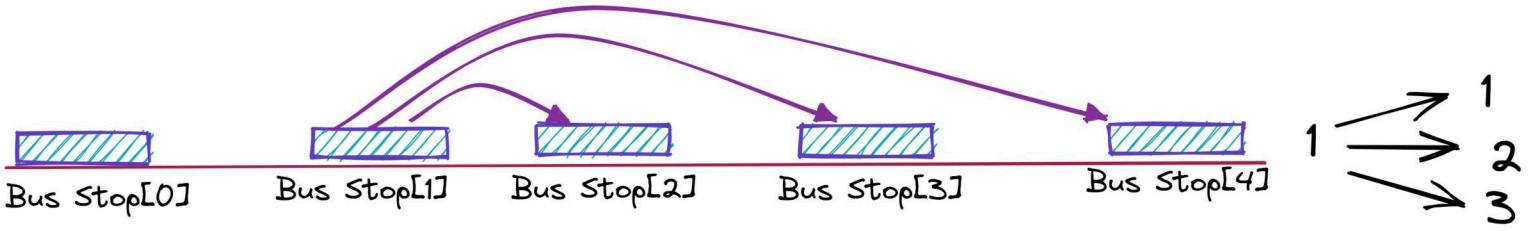
In this Question we have to reach to the last Index.

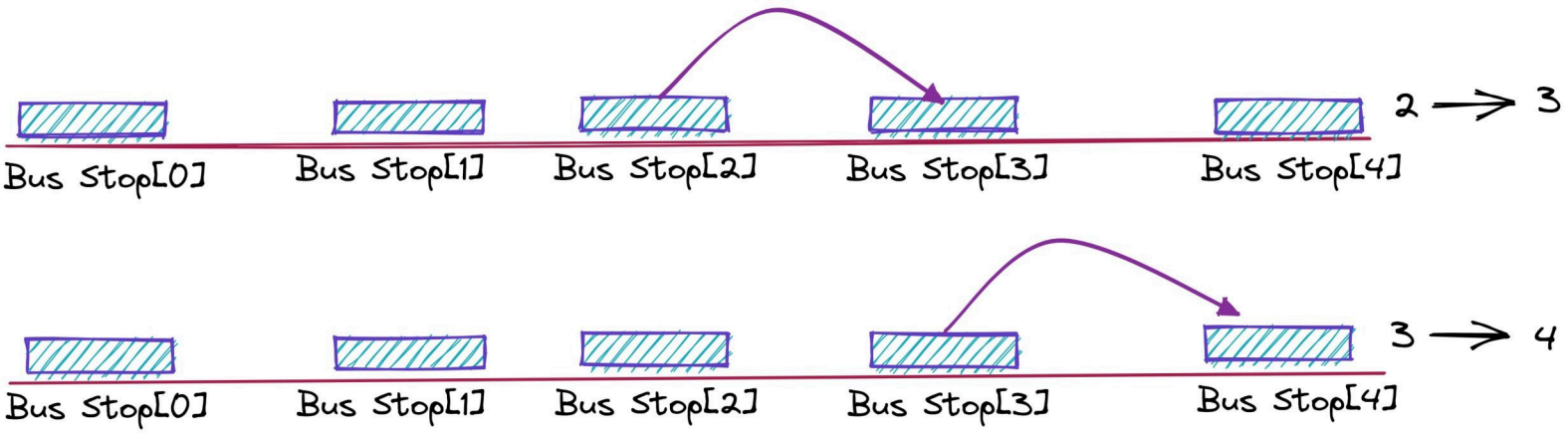
→ Reach at last Index, Return "true".
→ If not reach at last Index, Return "false".

```
Input: nums = [2,3,1,1,4]

Output: true

Explanation: Jump 1 step from index 0 to 1, then 3 steps to
the last index.
```

● Take a Example of Bus :-
Like, from Bus-Stop[0] we can go to bus-stop[1] or bus-stop[2] because nums array contains 2 at bus stop[0]







Approach-1 : Recursion

→ Create a recursive function.

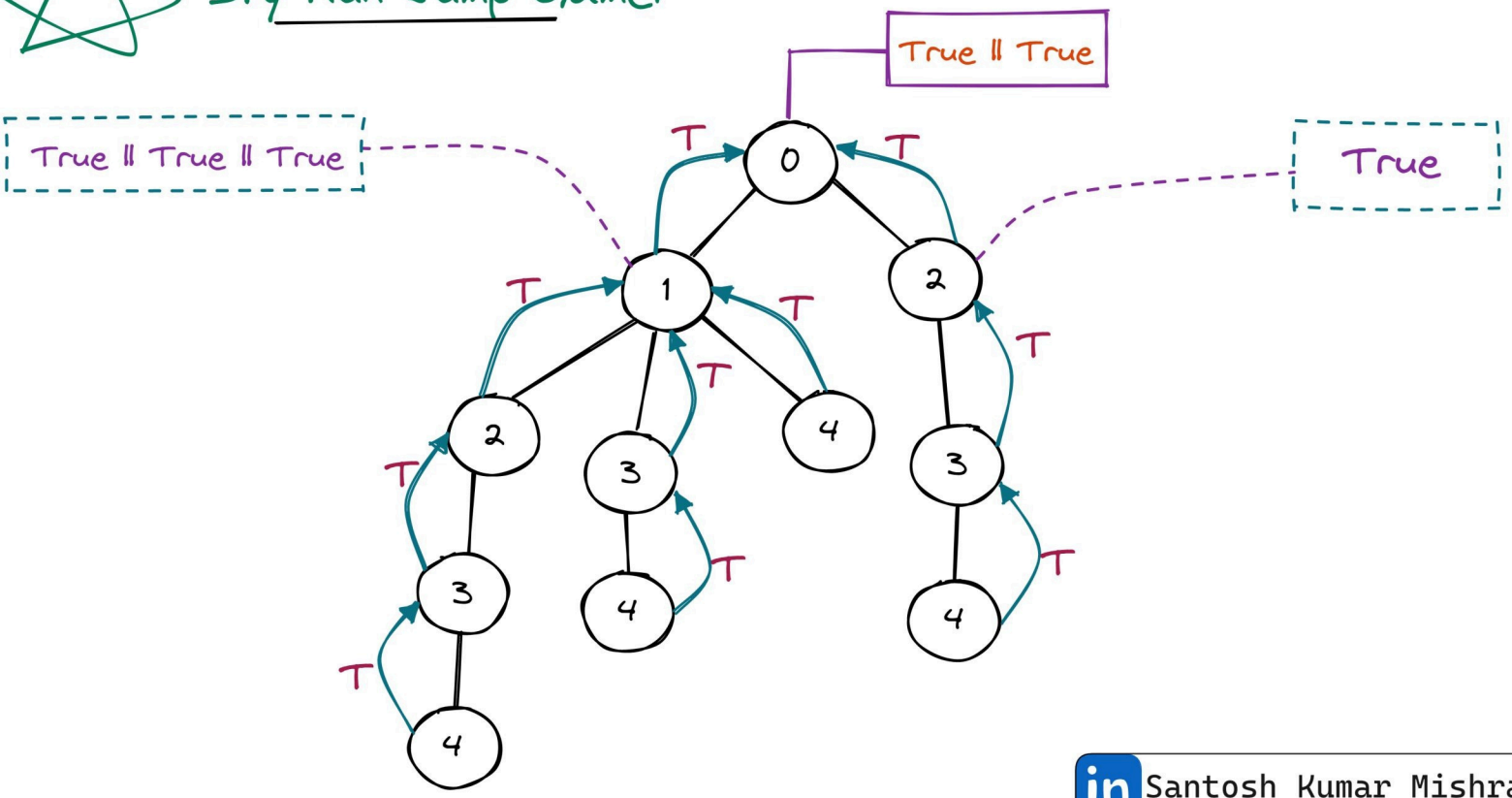
→ In each recursive call get all the reachable nodes from that index.

  → For each of the index call the recursive function.

  → Find the minimum number of jumps to reach the end from current index.

→ Return the minimum number of jumps from the recursive call.

Dry Run Jump Game:-



Approach 2: Recursion with Memoization

→ In the previous approach we are redundantly calculating the result for every step.

Instead, we can store the result at each step in memo `Array/Hashmap` and directly returning the result from the memo array whenever that function is called again.

→ In this way we are pruning recursion tree with the help of memo array and reducing the size of recursion tree upto n.

```java
class Solution {
    public boolean canJump(int[] nums) {
       int[] dp = new int[nums.length];
        Arrays.fill(dp, -1);
        return isPossible(nums,0,dp);
    }

    public boolean isPossible(int[] nums,int currentIndex,int[] dp){

        if(currentIndex >=nums.length-1){
            return true;
        }

        if(dp[currentIndex]!=-1){
            return dp[currentIndex] == 1;
        }

        int maxJump = nums[currentIndex];

        for(int jump = 1;jump<=maxJump;jump++){

            if(isPossible(nums,currentIndex+jump,dp)){
                dp[currentIndex] = 1;
                return true;
            }
        }
        dp[currentIndex] = 0;
        return false;
    }
}
```

- Time-Complexity ⟶ O(n)
- Space-Complexity ⟶ O(n), The depth of the recursion tree can go upto n.

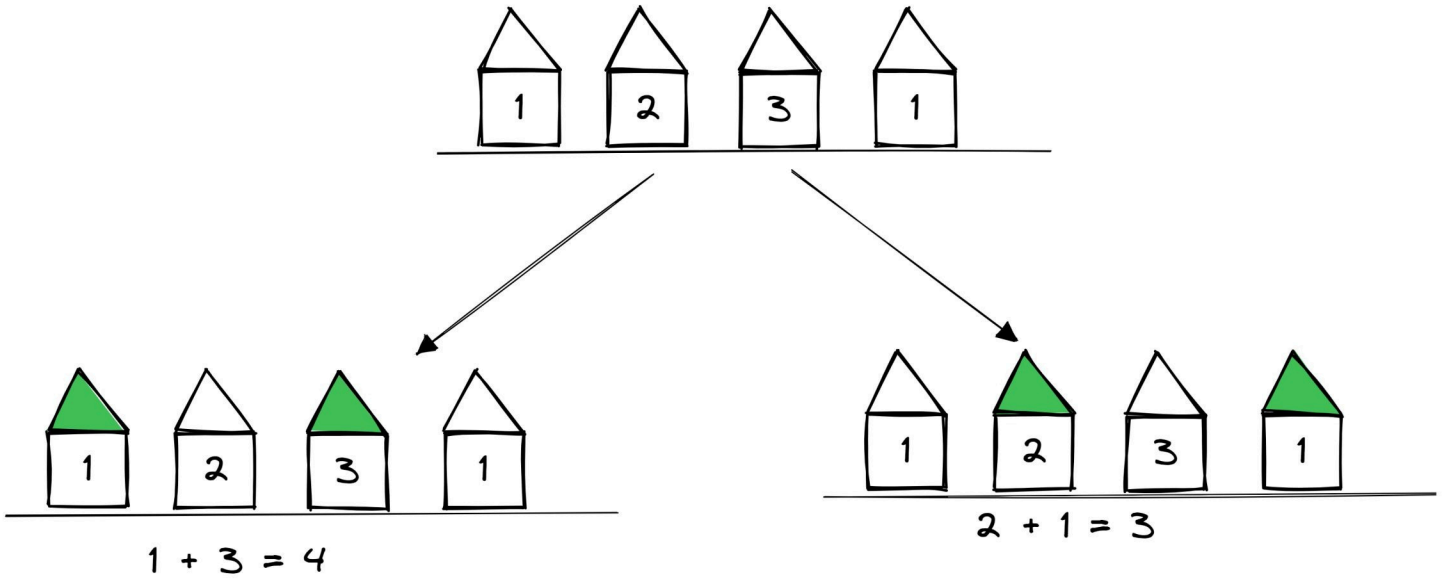
Santosh Kumar Mishra

House Robber

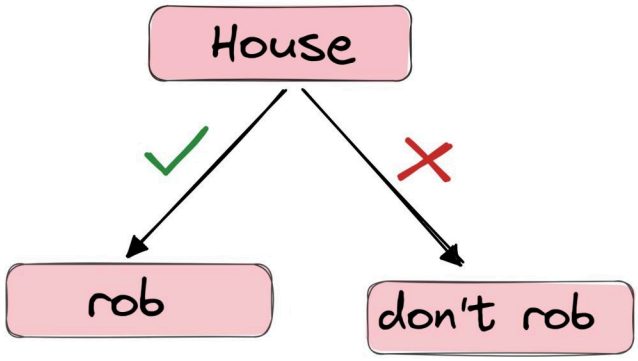
→ You are a professional robber planning to rob houses along a street.

→ if two adjacent houses were broken into on the same night.

```
Input: nums = [1,2,3,1]
Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3
(money = 3).
Total amount you can rob = 1 + 3 = 4.
```



→ There are only two ways :-

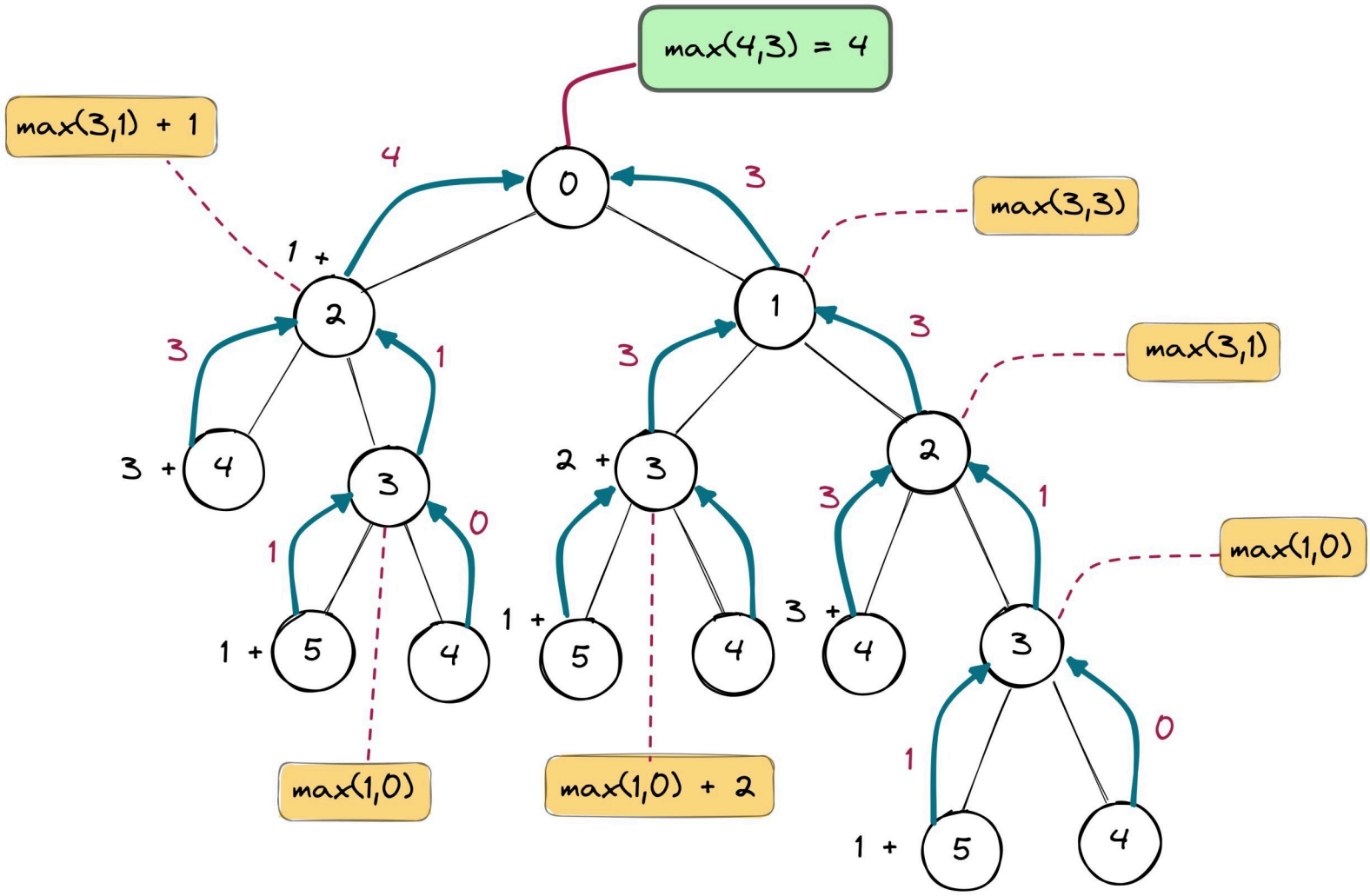


Approach-1 : Recursion

- If an element is selected then the next element cannot be selected.
- If an element is not selected then the next element can be selected.
  Use recursion to find the result for both cases.

Dry-Run

```
  0  1  2  3
[ 1, 2, 3, 1 ]
```



Time Complexity: O(2^N). Every element has 2 choices to pick and not pick.

Auxiliary Space: O(N). A recursion stack space is required.

Santosh Kumar Mishra

Approach 2: Recursion with Memoization

→ In the previous approach we are redundantly calculating the result for every step.

Instead, we can store the result at each step in memo `Array/Hashmap` and directly returning the result from the memo array whenever that function is called again.

→ In this way we are pruning recursion tree with the help of memo hashmap and reducing the size of recursion tree upto n.

```java
class Solution {
    public int rob(int[] nums) {
        HashMap<Integer,Integer> map = new HashMap<>();
       return helper(0,nums,map);
    }

    public int helper(int curr,int[] nums, HashMap<Integer,Integer> map){

        if(curr >= nums.length) return 0;

        int key = curr;
        if(map.containsKey(key)){
            return map.get(key);
        }

        int rob = nums[curr] + helper(curr+2,nums,map);
        int notrob = helper(curr+1,nums,map);

        map.put(key,Math.max(rob,notrob));
        return map.get(key);
    }
}
```

- Time-Complexity ⟶ O(n)
- Space-Complexity ⟶ O(n), The depth of the recursion tree can go upto n.

Santosh Kumar Mishra

Approach 3: Bottom-Up Approach

→ The sub-problems can be stored thus reducing the complexity and converting the recursive solution to a Dynamic programming problem.

→ Tackle the following basic cases,

If the length of the array is 0, print 0.

If the length of the array is 1, print the first element.

If the length of the array is 2, print a maximum of two elements.

```java
class Solution {
    public int rob(int[] nums) {
        HashMap<Integer,Integer> map = new HashMap<>();
       return helper(0,nums,map);
    }

    public int helper(int curr,int[] nums, HashMap<Integer,Integer> map){

        if(curr >= nums.length) return 0;

        int key = curr;
        if(map.containsKey(key)){
            return map.get(key);
        }

        int rob = nums[curr] + helper(curr+2,nums,map);
        int notrob = helper(curr+1,nums,map);

        map.put(key,Math.max(rob,notrob));
        return map.get(key);
    }
}
```

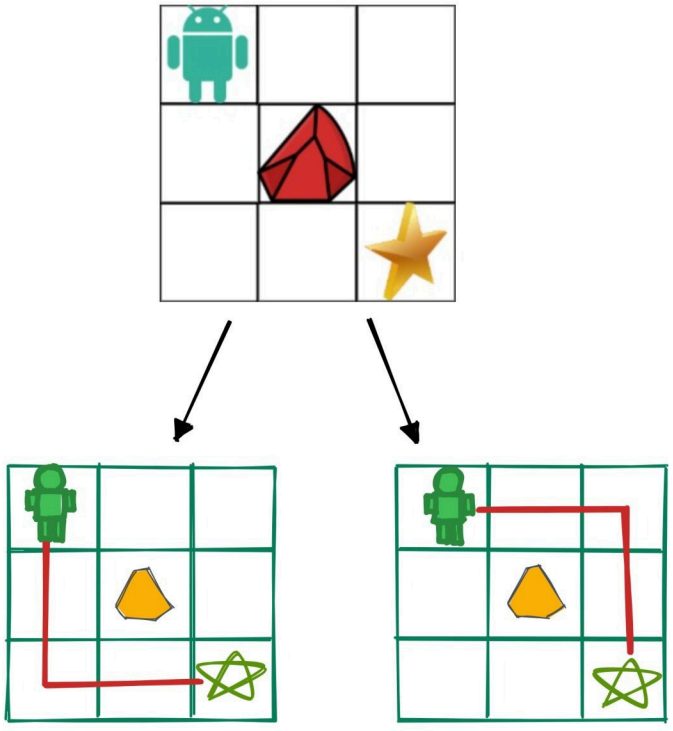
- Time-Complexity → O(n), single loop upto n.
- Space-Complexity → O(n), dp array of size n is used.

Santosh Kumar Mishra

Unique Paths

- A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).
- The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid
- we have to Reach to cornor grid at bottom-right, if there is an obstical so avoid that path.

```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2

Explanation: There is one obstacle in the middle of the 3×3 grid above.
There are two ways to reach the bottom-right corner:
1. Right → Right → Down → Down
2. Down → Down → Right → Right
```

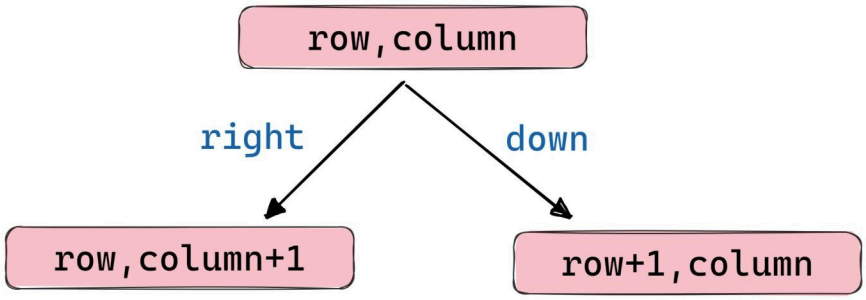


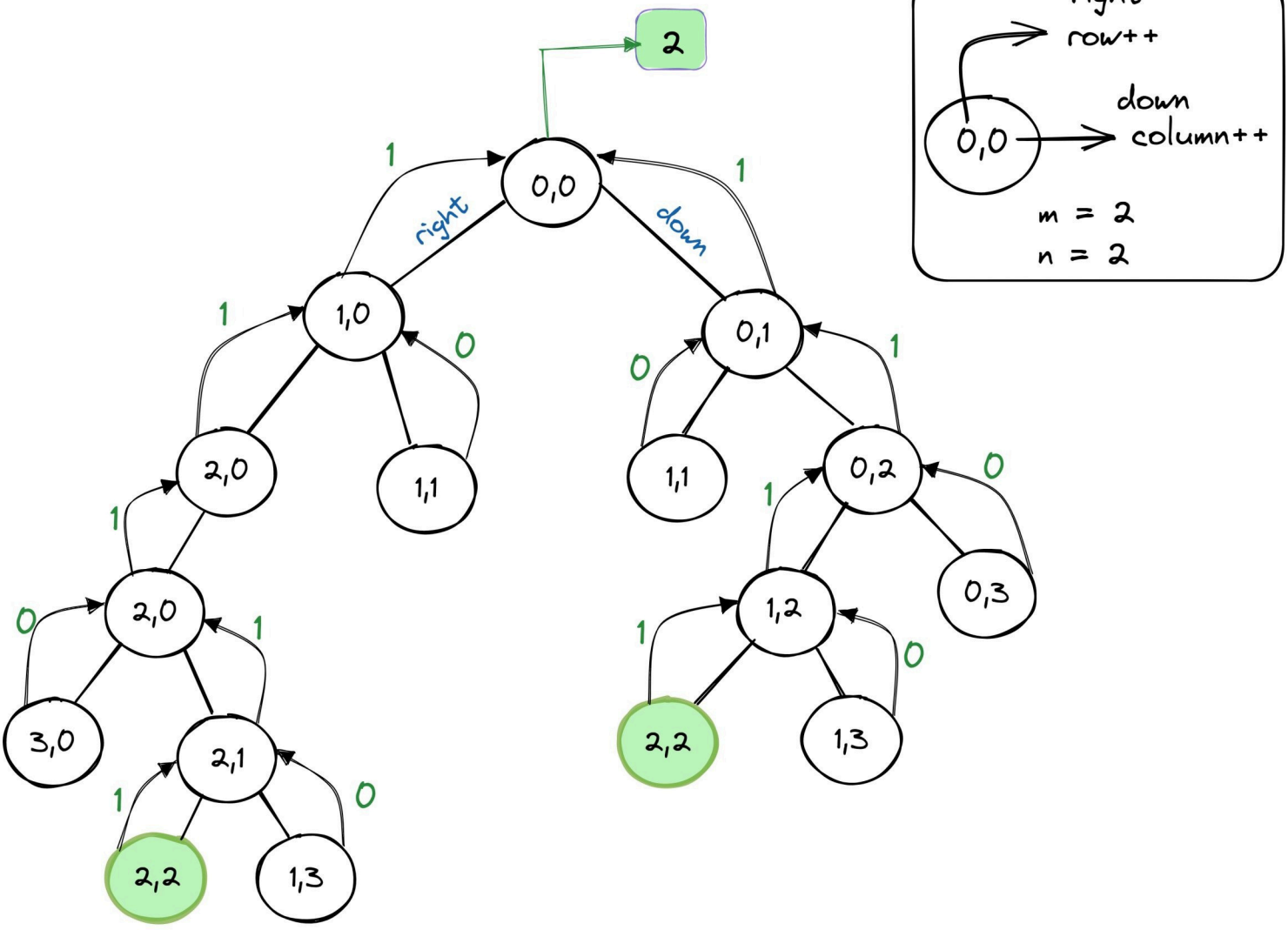
Santosh Kumar Mishra

Approach-1 : Recursion

→ Create two recursive call for right and down.

→ While moving through the grid, we can get some obstacles that we can not jump and the way to reach the bottom right corner is blocked.





- Time Complexity: O(2^m*n)
- Auxiliary Space: O(m*n)

in Santosh Kumar Mishra

Approach 2: Recursion with Memoization

→ In the previous approach we are redundantly calculating the result for every step.

Instead, we can store the result at each step in memo `Array/Hashmap` and directly returning the result from the memo array whenever that function is called again.

→ In this way we are pruning recursion tree with the help of memo Hashmap and reducing the size of recursion tree upto n.

```java
class Solution {
    public int uniquePathsWithObstacles(int[][] grid) {

        int m = grid.length;
        int n = grid[0].length;

        return totalWays(grid,m,n,0,0,new HashMap<String,Integer>());
    }

    public int totalWays(int[][] grid,int m,int n,int currentRow,int currentColumn,
    HashMap<String,Integer> memo){

        if(currentRow >= m || currentColumn >= n || grid[currentRow][currentColumn] == 1){
            return 0;
        }

        if(currentRow == m-1 && currentColumn == n-1){
            return 1;
        }

        String currentKey = currentRow + "," + currentColumn;
        if(memo.containsKey(currentKey)){
            return memo.get(currentKey);
        }

        int right = totalWays(grid,m,n,currentRow,currentColumn+1, memo);
        int down = totalWays(grid,m,n,currentRow+1,currentColumn, memo);

        memo.put(currentKey,right + down);
        return memo.get(currentKey);
    }
}
```

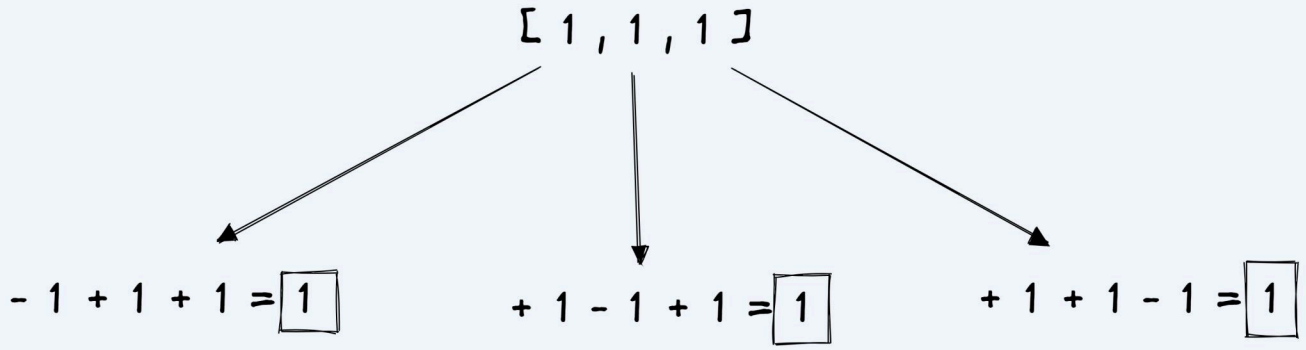
- Time Complexity: O(m*n)
- Auxiliary Space: O(m*n)

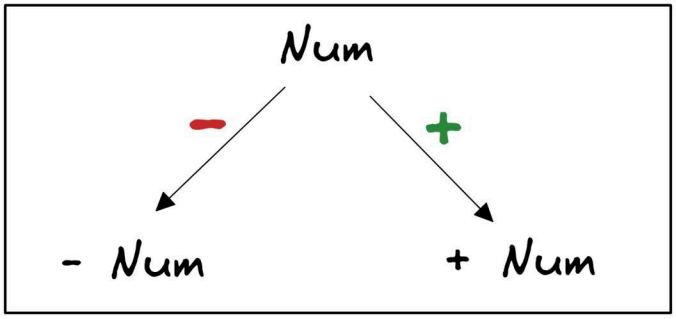
Target Sum

→ You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

nums[] = [1 , 1 , 1]

target = 1





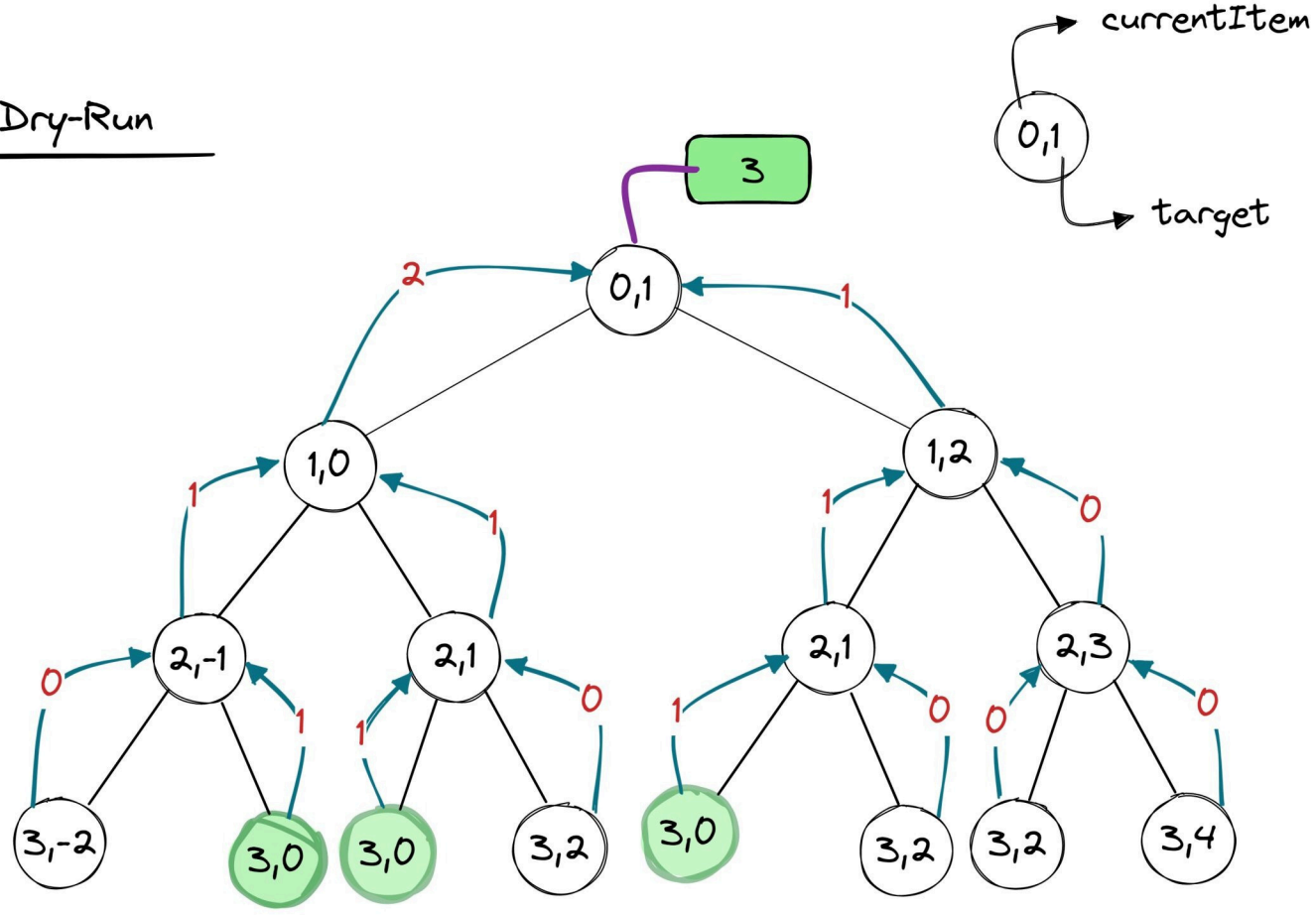
→ The most straightforward (and least efficient) solution is to explore every possibility using a backtracking algorithm.

Time Complexity: O(2^N)

Auxiliary Space: O(N), Stack space required for recursion

```java
class Solution {
    public int findTargetSumWays(int[] nums, int target){
        return totalWays(nums,target,0,0);
    }

    public int totalWays(int[] nums, int target,int index,int sum){

       if(index == nums.length){
           if(target == sum){
               return 1;
           }
           return 0;
       }

        int pos = totalWays(nums,target,index+1,sum + nums[index]);
        int neg = totalWays(nums,target,index+1,sum - nums[index]);

        return pos + neg;
  }
}
```

Dry-Run



Approach 2: Recursion with Memoization

→ Notice how in the previous solution we end up re-computing the solutions to sub-problems. We could optimize this by caching our solutions using memoization.

```java
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        HashMap<String,Integer> memo = new HashMap<>();
        return totalWays(nums,target,0,memo);
    }

    public int totalWays(int[] nums, int target,int currentIndex,HashMap<String,Integer> memo ){

       if(currentIndex >= nums.length && target != 0) return 0;
       if(currentIndex >= nums.length && target == 0) return 1;

        String currentKey =  target + "," +currentIndex;
        if(memo.containsKey(currentKey)){
            return memo.get(currentKey);
        }

        int positive = totalWays(nums,target-nums[currentIndex],currentIndex+1,memo);
        int negative = totalWays(nums,target+nums[currentIndex],currentIndex+1,memo);

        memo.put(currentKey,positive + negative);
        return memo.get(currentKey);
    }
}
```

- Time Complexity: O(N), where n is the items.
- Space Complexity : O(N) + O(N)

Coin Change

→ You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

→ Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

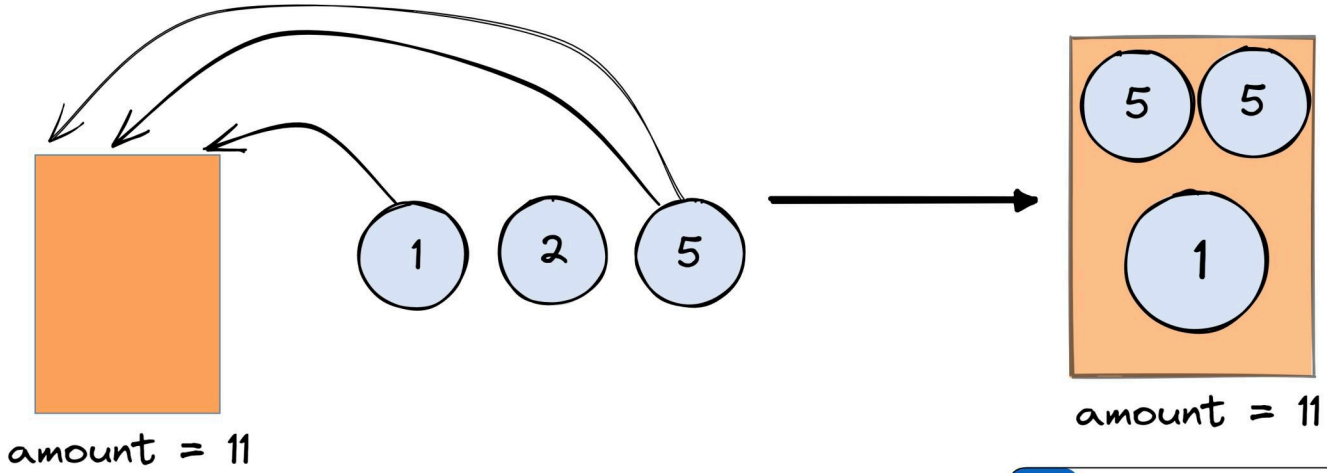
```
Input: coins = [1,2,5], amount = 11
Output: 3

Explanation: 11 = 5 + 5 + 1
```

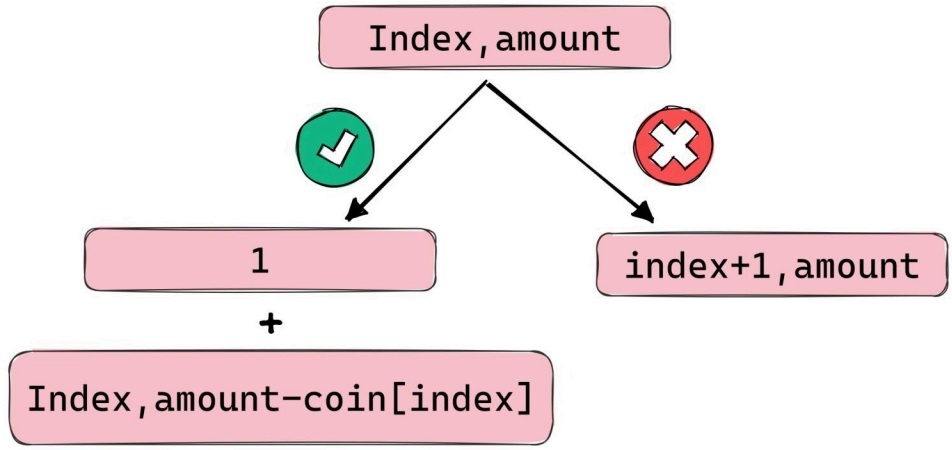
→ This Problem is based on un-bounded knapsack.

→ Suppose we have given some coins , we have to get the particular amount from minimum number of coins.

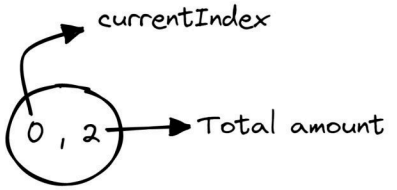
→ coins = [1,2,5] amount = 11

```
1 + 1 + 1 ..... + 1 = 11 (11 coins)
2 + 2 + 1 ..... + 1 = 11 (7 coins)
2 + 2 + 2 + 2 + 2 + 1 = 11 (6 coins)
5 + 5 + 1 = 11 (3 coins)
```



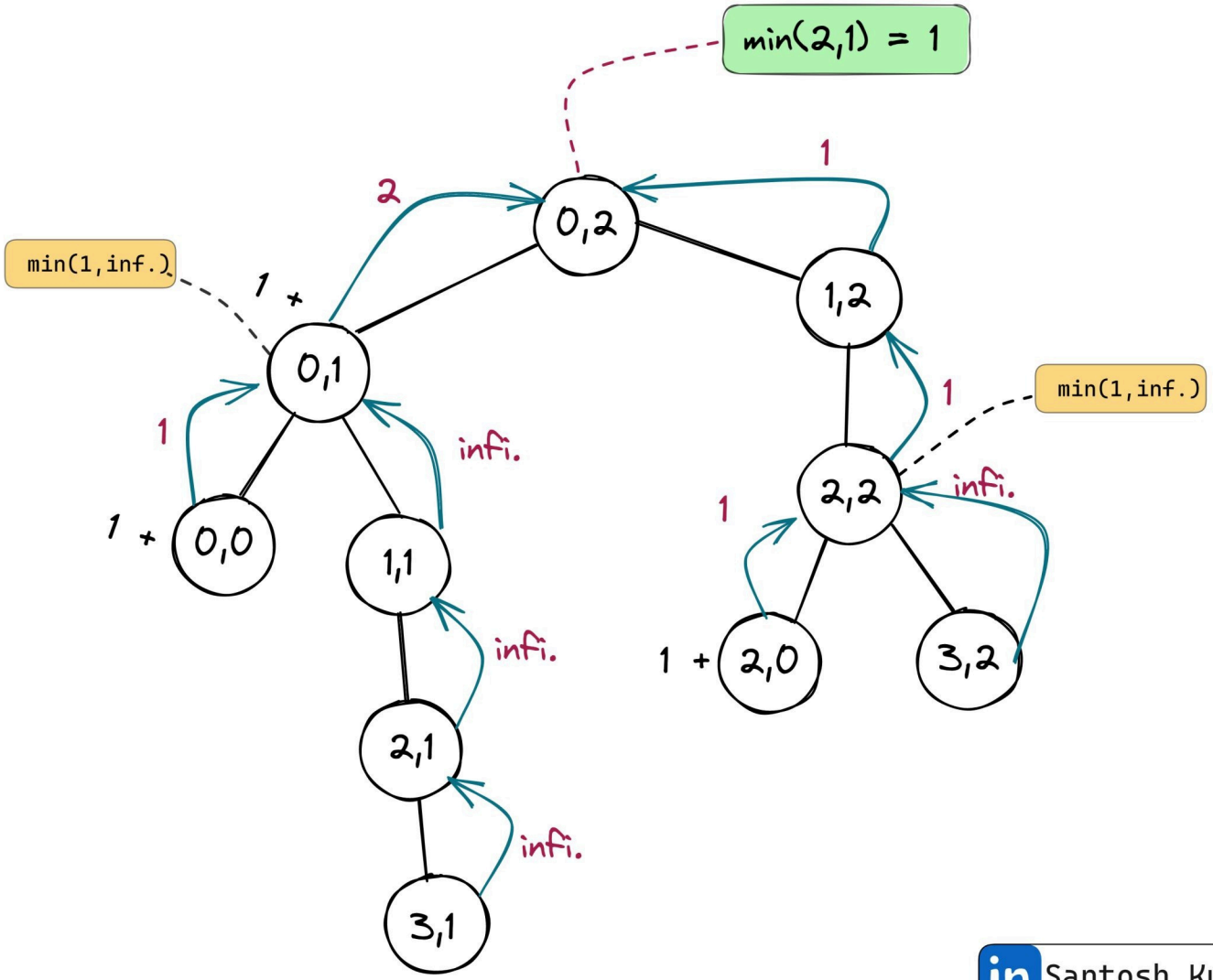


Dry Run :-



coins =	1	2	5
Index =	0	1	2

Time Complexity: O(2^amount)
Auxiliary Space: O(amount)



Approach 2: Recursion with Memoization

→ Using 2-D vector to store the Overlapping subproblems.

→ Traversing the whole array to find the solution and storing in the memoization table.
Using the memoization table to find the optimal solution.

```java
class Solution {
    public int coinChange(int[] coins, int amount) {

        int[][] dp = new int[coins.length+1][amount+1];
        int ans = maxProfit(coins,0,amount,dp);

        if(ans == 100000) return -1;
        return ans;
    }

    public int maxProfit(int[] coins,int currentIndex,int amount,int[][] dp){

        if(amount == 0) return 0;
        if(currentIndex == coins.length) return 100000;

        if(dp[currentIndex][amount]!=0){
            return dp[currentIndex][amount];
        }

        int consider = 100000;
        if(coins[currentIndex] <=amount){
           consider = 1 + maxProfit(coins,currentIndex,amount-coins[currentIndex],dp);
        }
        int notconsider = maxProfit(coins,currentIndex+1,amount,dp);

        dp[currentIndex][amount] = Math.min(consider,notconsider);
        return  Math.min(consider,notconsider);
    }
}
```

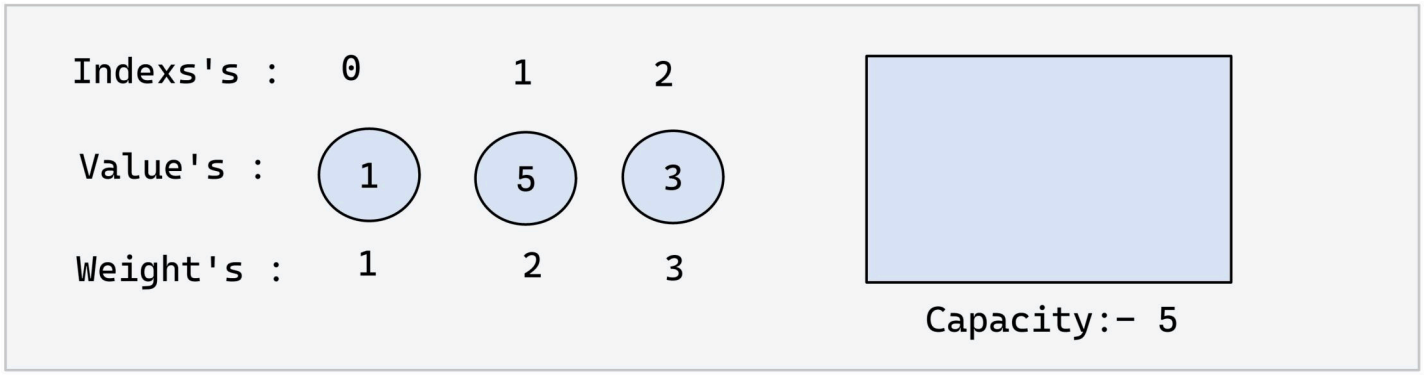
- Time Complexity: O(S*n), where S is the amount, n is denomination count.
- Space Complexity : O(S), where S is the amount to change We use extra space for the memoization table.

Santosh Kumar Mishra

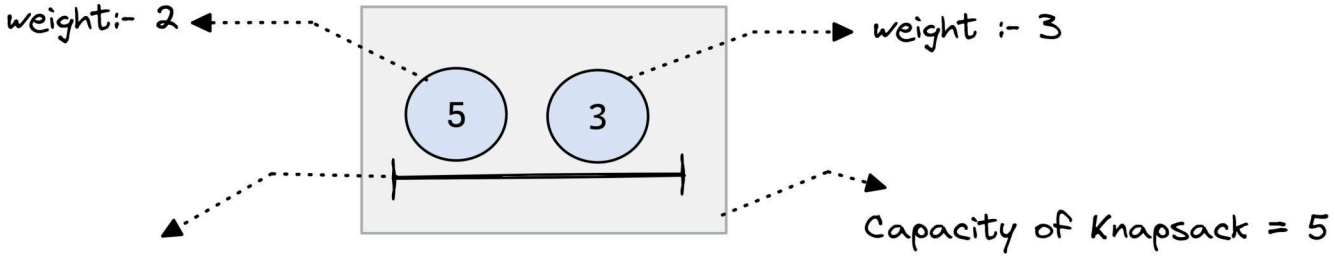
0-1 Knapsack

- Given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.
- In other words, given two integer arrays val[0..N-1] and wt[0..N-1] which represent values and weights associated with N items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that the sum of the weights of this subset is smaller than or equal to Capacity.
- You cannot break an item, either pick the complete item or don't pick it (0-1 property)

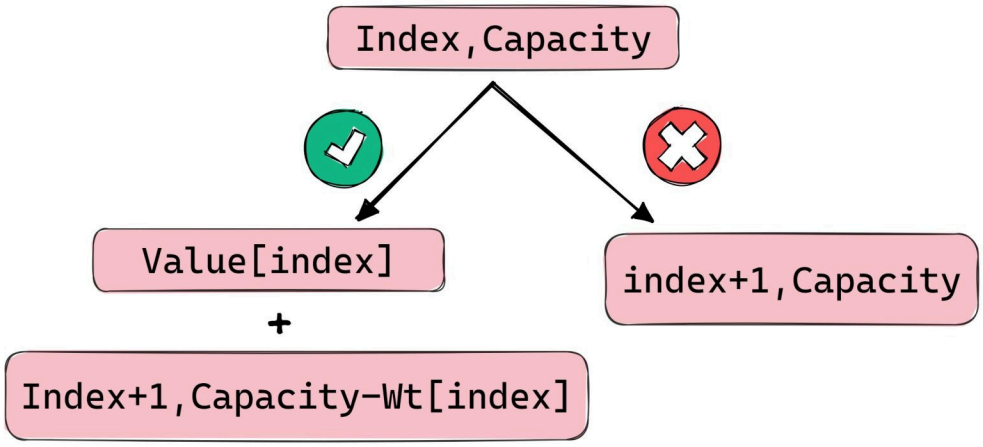
```
Input: N = 3, Capacity = 5
values[] = [ 1 , 5 , 3 ]
weight[] = [ 1 , 2 , 3 ]
Output: 2
```



- We have to create max. profit from this knapsack with capacity 5.
- So,from the above diagram we can say that the max. profit is 8, how?



Santosh Kumar Mishra

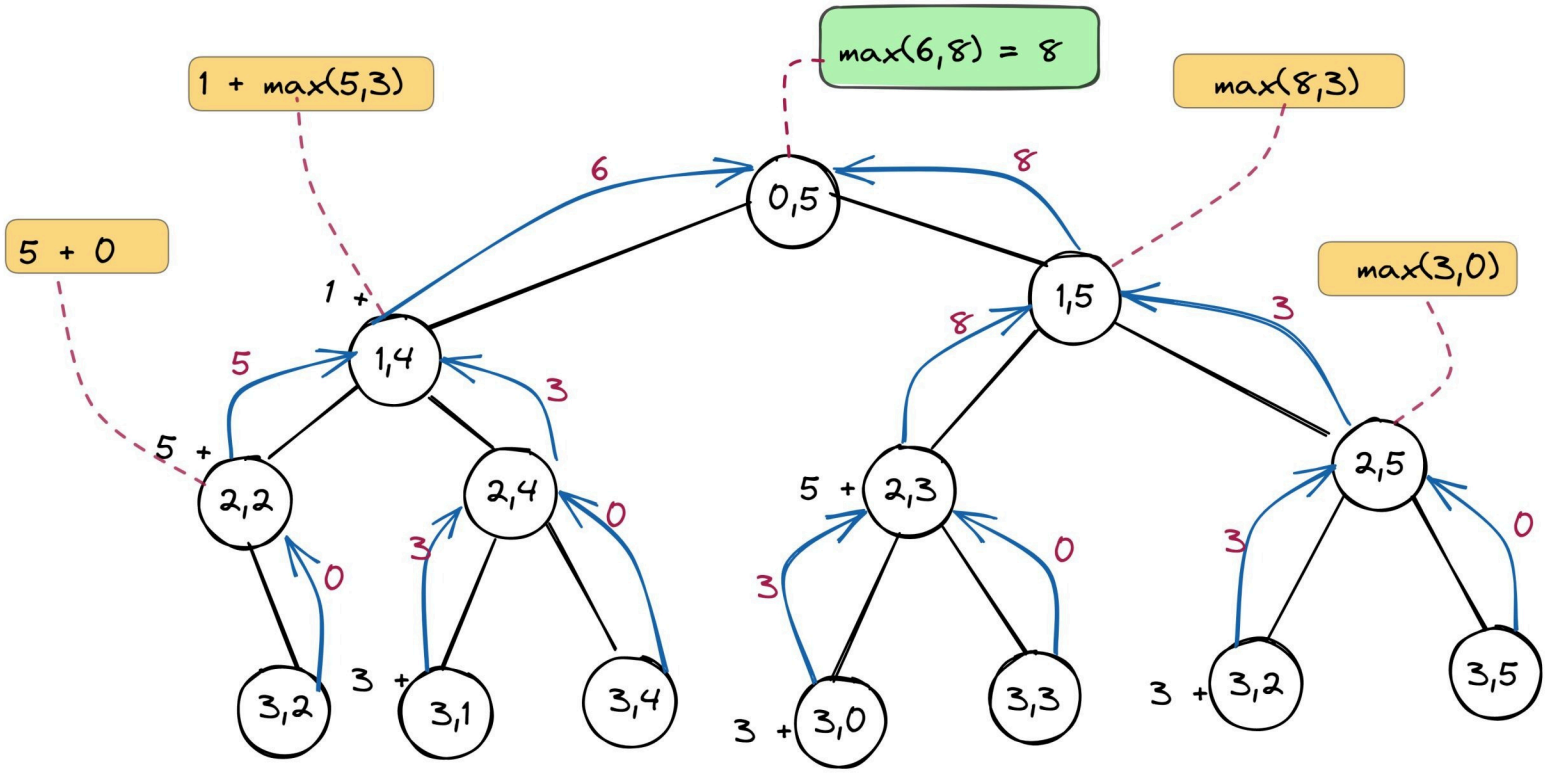


Time Complexity: O(2N)

Auxiliary Space: O(N), Stack space required for recursion

Dry-Run

```
Capacity = 5
values[] = [ 1 , 5 , 3 ]
weight[] = [ 1 , 2 , 3 ]
```



Approach 2: Recursion with Memoization

→ We can use memoization to overcome the overlapping sub-problems. To reiterate, memoization is when we store the results of all the previously solved sub-problems and return the results from memory if we encounter a problem that has already been solved.

```java
class Solution {
    public int knapSack(int capacity, int weight[], int value[], int n) {

        HashMap<String,Integer> memo = new HashMap<>();
         return maxProfit(weight,value,capacity,0,n,memo);
    }

    public static int maxProfit(int weight[],int value[],int capacity,int currentItem,int n,
        HashMap<String,Integer> memo){

        if(currentItem == n) return 0;

        int consider = 0;

        String key = capacity + "," + currentItem;

        if(memo.containsKey(key)){
            return memo.get(key);
        }

        if(weight[currentItem] <= capacity){
           consider = value[currentItem] +  maxProfit(weight,value,
           capacity - weight[currentItem],currentItem+1,n,memo);
        }
        int notconsider =  maxProfit(weight,value,capacity,currentItem+1,n,memo);

        memo.put(key,Math.max(consider,notconsider));
        return memo.get(key);
    }
}
```

- Time Complexity: O(n*Capacity), where n is the items.
- Space Complexity : O(n*Capacity + N), where N is the Recursive Stack Space.

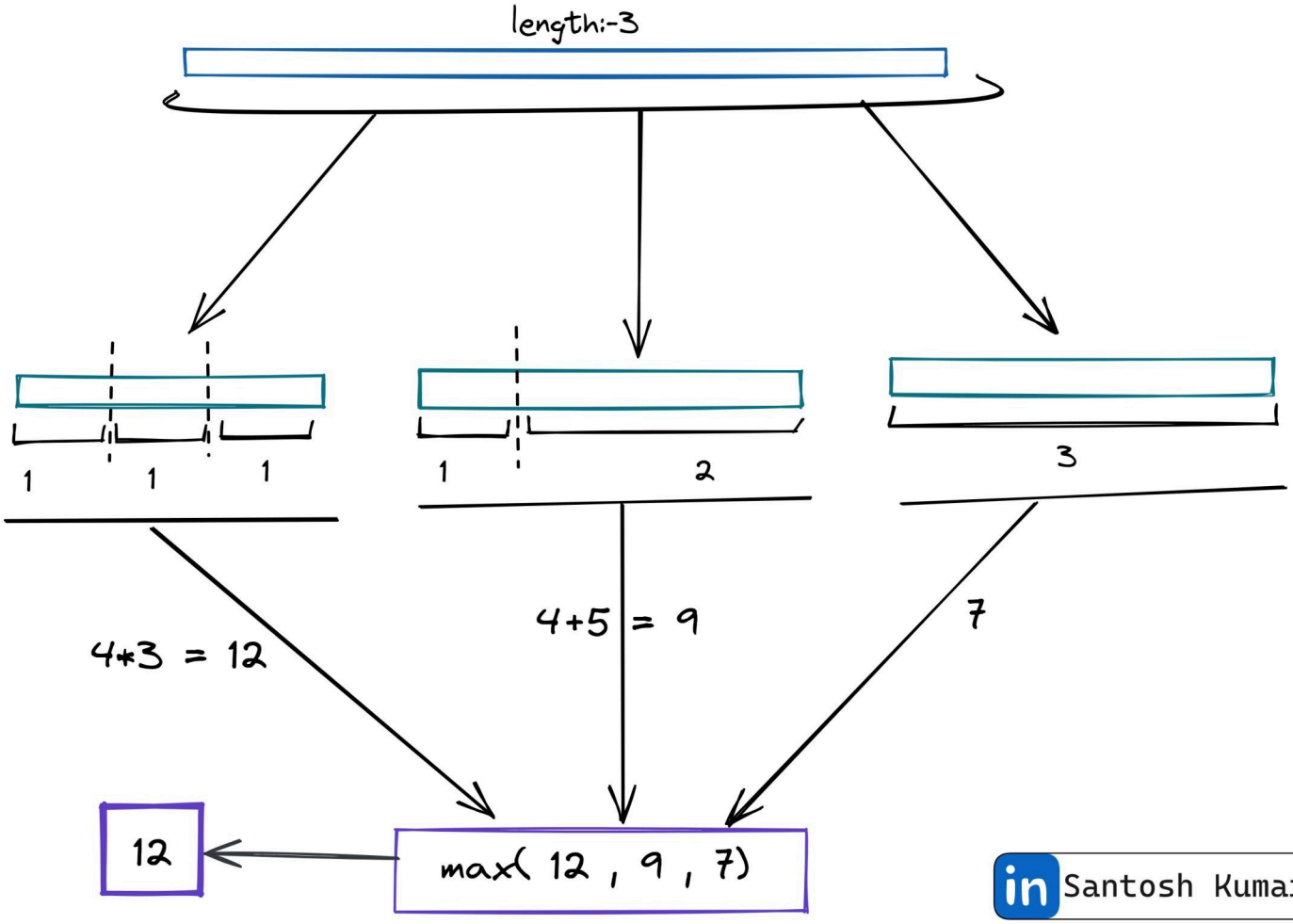
Un-Bounded KnapSack

- Given a knapsack weight W and a set of n items with certain value val[i] and weight wt[i], we need to calculate the maximum amount that could make up this quantity exactly.
- This is different from classical Knapsack problem, here we are allowed to use unlimited number of instances of an item.

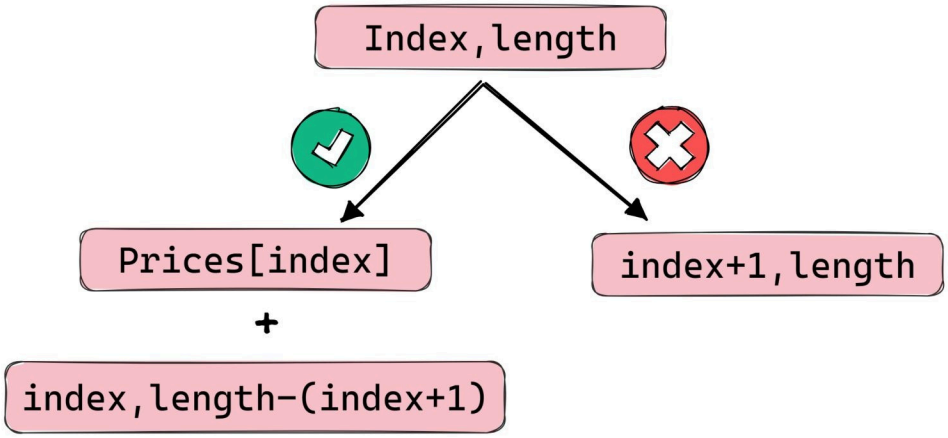
Unbounded Knapsack ⟶ Repetition of items allowed

Ques. Suppose a rod of Length 3 , we have to cut that rod to get maximum profit,The price is mention below for each cut:-

```
index's: =  0  1  2  3
price[] = [ 4, 5, 7, 2]
cut's: =    1  2  3  4
```

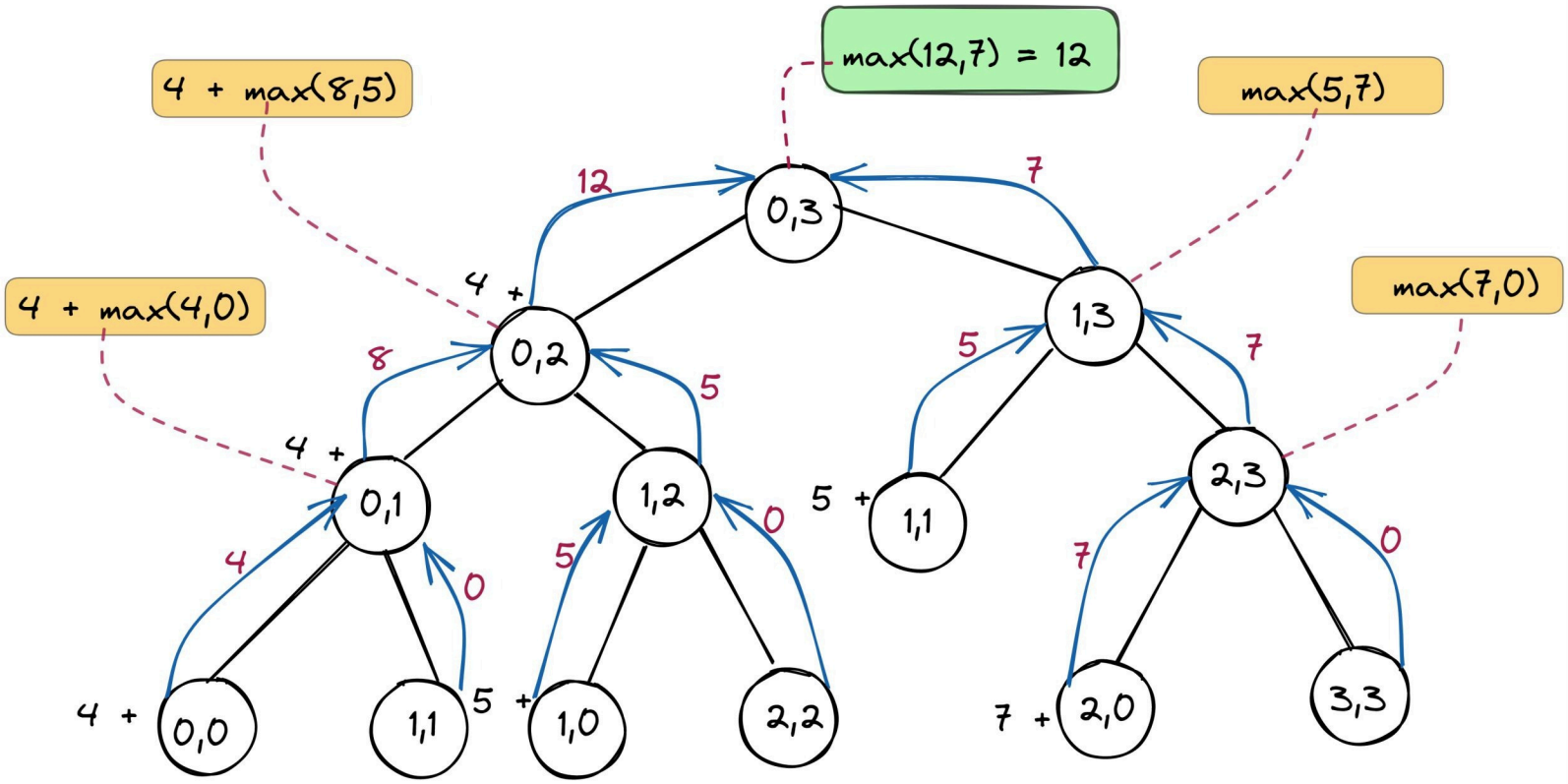


→ There are only two ways either we are going to cut that rod or we are not going to cut the rod.



Rode cutting Problem: -

```
index's: =  0  1  2  3
price[] = [ 4, 5, 7, 2]
cut's: =    1  2  3  4
```



Approach 2: Recursion with Memoization

→ In the previous approach we are redundantly calculating the result for every step.

Instead, we can store the result at each step in memo `Array/Hashmap` and directly returning the result from the memo array whenever that function is called again.

→ In this way we are pruning recursion tree with the help of memo array and reducing the size of recursion tree upto n.

```java
public int maxProfit(int price[], int currentIndex,int l,int[][] dp){

        if(l == 0) return 0;
        if(currentIndex + 1 > l) return 0;

        if(dp[l][currentIndex]!=-1){
            return dp[l][currentIndex];
        }

        int consider = price[currentIndex] + maxProfit( price,currentIndex,l - (currentIndex + 1), dp );
        int notconsider = maxProfit(price,currentIndex+1,l,dp);

        dp[l][currentIndex] = Math.max(consider,notconsider);

        return  dp[l][currentIndex];
}
```

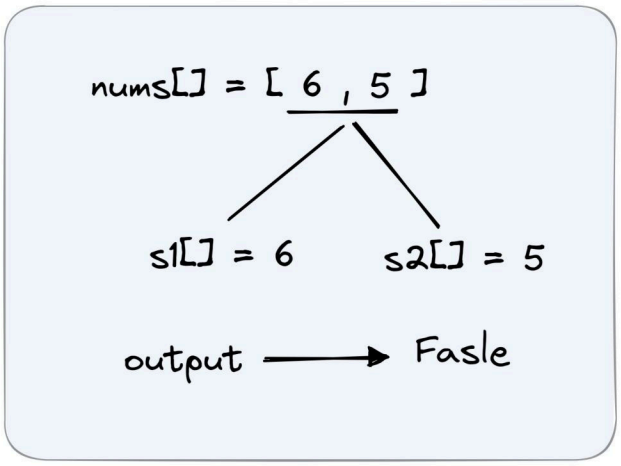
- Time Complexity: O(n^2)
- Auxiliary Space: O(n^2) + O(n)

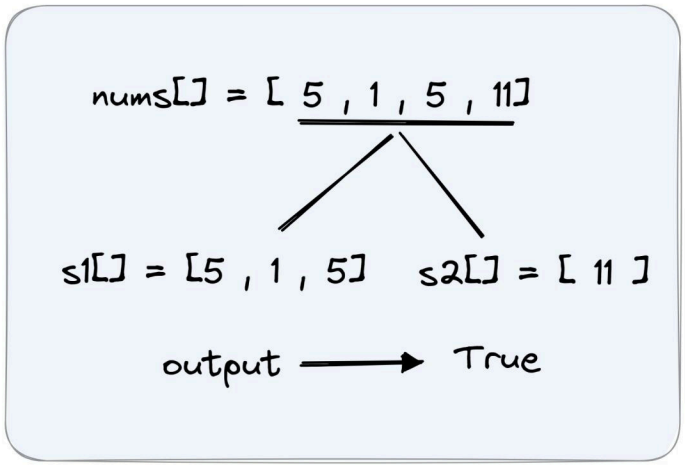
Partition Equal Subset Sum

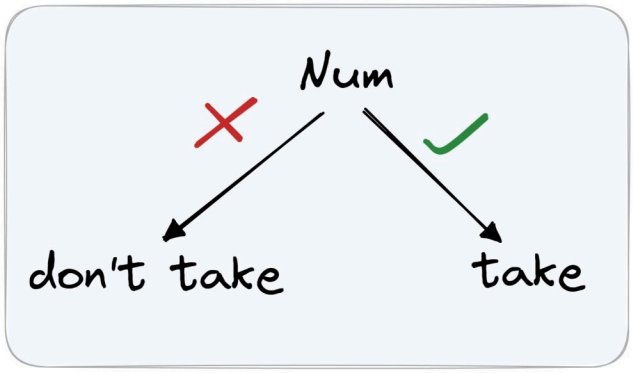
→ Array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

1st check - if its sum is odd, then return false

2nd check - if its sum is even , make recursive call.

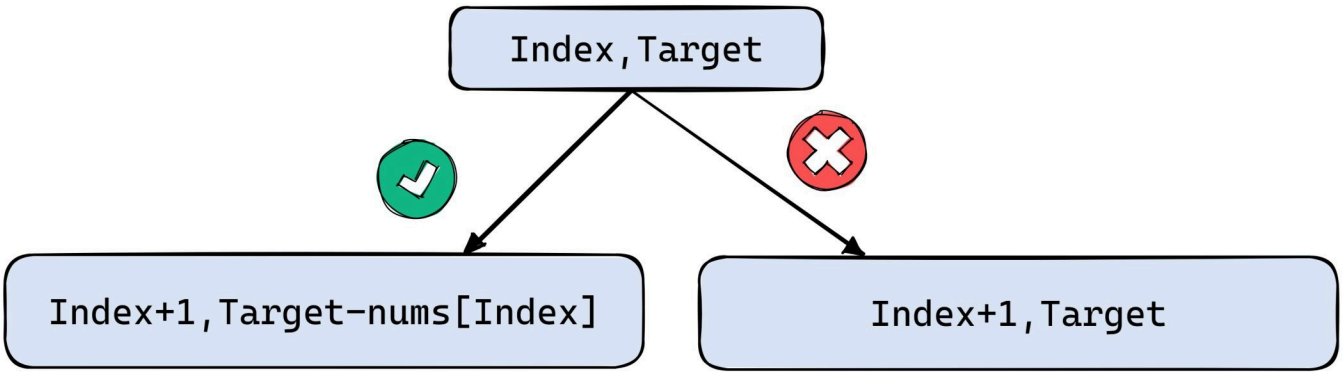






→ The following are the two main steps to solve this problem:

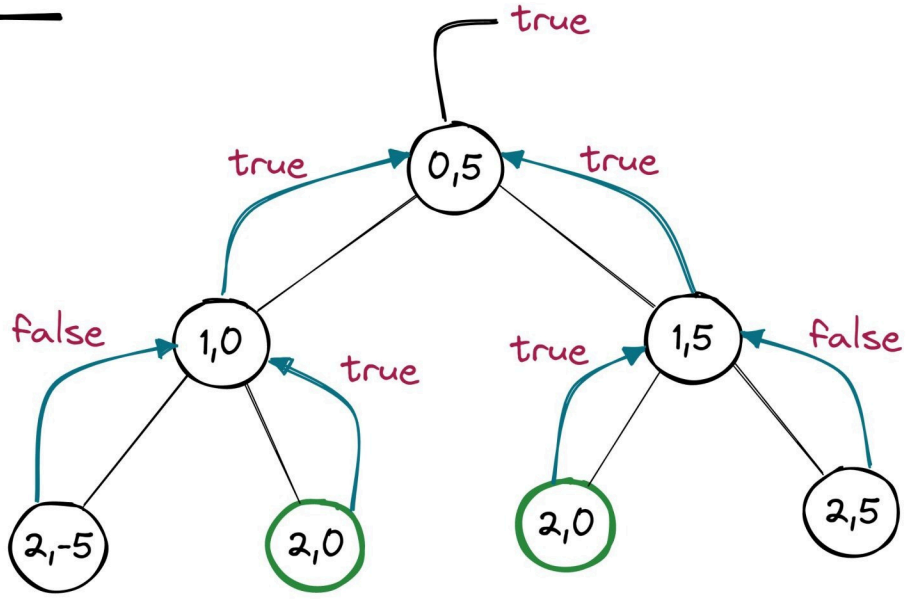
1. Calculate the sum of the array. If the sum is odd, there can not be two subsets with an equal sum, so return false.
2. If the sum of the array elements is even, calculate sum/2 and find a subset of the array with a sum equal to sum/2.
3. The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.



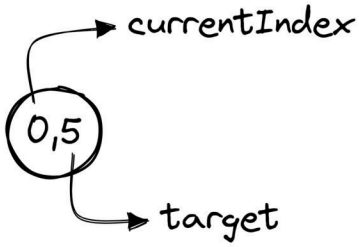
→ Time Complexity: O(2N) In the worst case, this solution tries two possibilities (whether to include or exclude) for every element.

→ Auxiliary Space: O(N). Recursion stack space

Dry-Run




```
  0  1
nums[] = [ 5 , 5 ]
```



Approach 2: Recursion with Memoization

→ Notice how in the previous solution we end up re-computing the solutions to sub-problems. We could optimize this by caching our solutions using memoization.

```java
class Solution {
    public boolean canPartition(int[] nums) {

       HashMap<String,Boolean> memo = new HashMap<String,Boolean>();
        int sum = 0;
        for(int num : nums) sum += num;

        if(sum%2!=0){ return false; }
        else return maxProfit(nums,0,sum/2,memo);
    }

    public boolean maxProfit(int[] nums,int currentItem,int n, HashMap<String,Boolean> memo)
{
        if(n == 0) return true;
        if(currentItem >= nums.length && n == 0) return true;
        if(currentItem >= nums.length && n != 0) return false;

        String key = currentItem + "," + n;

        if(memo.containsKey(key)){
            return memo.get(key);
        }

       boolean selectItem = maxProfit(nums,currentItem+1,n - nums[currentItem],memo);
        if(selectItem == true){
            return true;
        }
       boolean notselectItem = maxProfit(nums,currentItem+1,n,memo);

        memo.put(key,selectItem || notselectItem);
       return memo.get(key);
    }
}
```

- Time Complexity: O(sum*N)
- Space Complexity : O(sum*N)

The Art Of Data Structures and Algorithms

Visual Aids and Detailed Dry Runs for Easy Understanding

The Art of Structures Algorithms

ate Quick Reference Guide

₹399 ~~₹999~~

sh Kumar Mishra

gineer at Microsoft, Author

@iamsantoshmishra

+91-9701101993

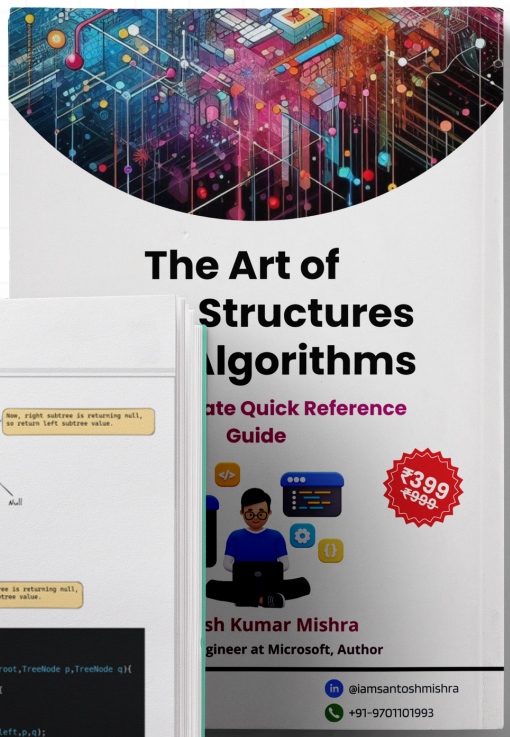
Questions Explained from Brute Force to Optimized Solutions

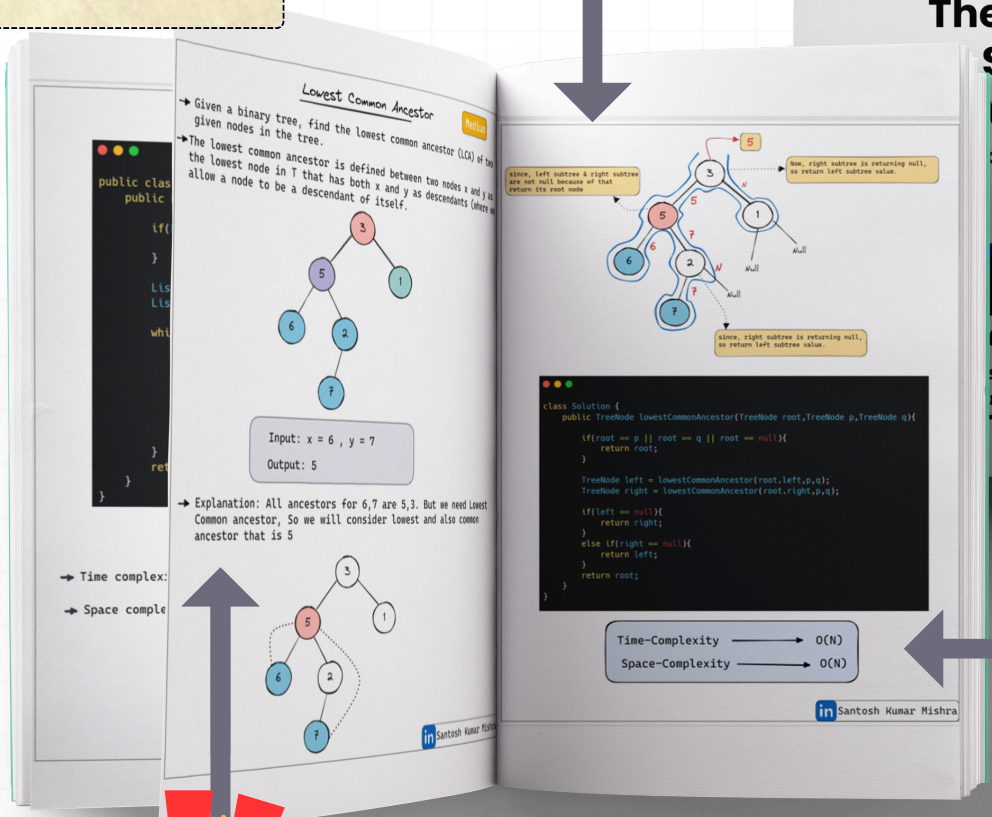
Perfect for Beginners to Advanced Learners

BUY NOW

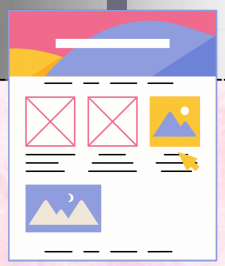
The Art Of Data Structures and Algorithms

One-Time Purchase with Lifetime Access





Comprehensive Coverage of Important DSA Questions



Expertly Designed by Industry Professionals

The Art of Data Structures and Algorithms

Readers' Reviews and Insights

Prem Kumar

Software Engineer

I've never seen a book that covers DSA questions so thoroughly! Santosh breaks down each problem from the basic brute-force solution to the most optimized version, using proper diagrams and dry runs. It's a fantastic resource if you struggle with understanding complex algorithms.

Saumya Awasthi

Software Engineer

If you're preparing for interviews, this is the book you need. It contains all the key DSA problems explained with easy-to-follow graphics. The step-by-step approach, from brute force to optimized solutions, along with well-done dry runs, makes difficult concepts very digestible.

Archy Gupta

Software Engineer

Thank you, Santosh, for sending me a copy of this book. It's truly amazing! The book contains all the important DSA questions, and each one is explained from brute-force to optimized solutions with very clear dry runs and diagrams. It's the perfect resource for mastering DSA concepts and acing coding interviews.

The Art of Data Structures and Algorithms
Readers' Reviews and Insights

Aishwarya Tripathi
Software Engineer

"This book was a game-changer for my Amazon interview preparation. The collection of important DSA problems and their step-by-step solutions from brute-force to optimized helped me build a solid understanding. The graphical dry runs made complex concepts easy to follow. I highly recommend this book if you're aiming for top tech companies!

Parth Chaturvedi
Software Engineer

"The structured explanations in this book played a huge role in my Amazon interview success. The dry runs and graphical representations provided a clear understanding of complex problems, and the progression from brute-force to optimal solutions was exactly what I needed. This book is a must-read for serious interview prep.

Namrata Tiwari
Software Engineer

"This book's strength lies in its ability to break down complex problems into simple, understandable steps. The questions are explained starting from brute-force solutions and are then optimized with clear, graphical dry runs. It's an essential guide for interview prep and mastering DSA.

And Many More Success Stories from Satisfied Readers!

The Art of Data Structures and Algorithms

The Ultimate Quick Reference Guide

BUY NOW

Buy From Gumroad **Buy From Topmate**