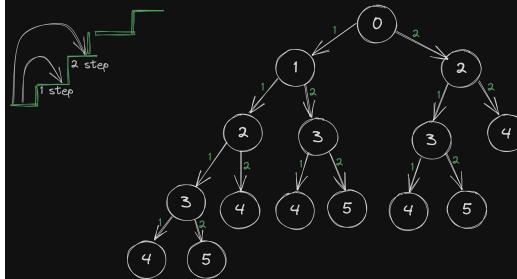


Note That when ever there is choices then we are going to use Recursion.

Ques- 70. Climbing Stairs?  
we have to return number of possible ways to reach n'th step.

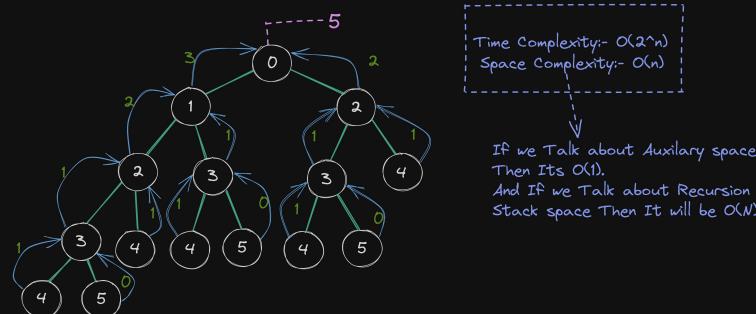
Note:- we have two possible ways either we can take one step or two step at a time.

$n = 4$   
No. of steps



Note:-  
1. If it reach to target step then return 1.  
2. If it reach beyond target step then, return 0.

Using Recursion only:-



Time Complexity:-  $O(2^n)$   
Space Complexity:-  $O(n)$

If we Talk about Auxiliary space  
Then Its  $O(1)$ .  
And If we Talk about Recursion  
Stack space Then It will be  $O(N)$

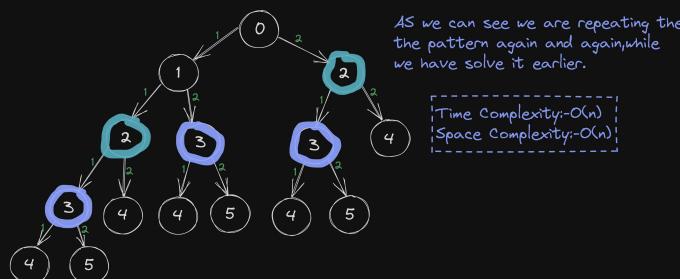
CODE IN JAVA:-

```
class Solution {
    public int climbStairs(int n) {
        return stair(0,n);
    }

    public int stair(int currStair, int targetStair){
        Base Case
        if(currStair == targetStair){           // 4 == 4, return 1;
            return 1;
        }
        if(currStair > targetStair){          // 5 > 4, return 0;
            return 0;
        }

        Recursive Call
        int oneStep = stair(currStair + 1, targetStair);
        int twoStep = stair(currStair + 2, targetStair);
        return oneStep + twoStep;           //It will return No. of possible Ways.
    }
}
```

Now, we Have to Optimize our Code to DP(Memoization):-



Code Using Memoization :-

```
class Solution {
    public int climbStairs(int n) {
        int[] dp = new int[n+1];           //Create an array
                                         //of n+1 size
        Arrays.fill(dp,-1);              //Fill whole array with -1.
        return stair(0,dp);
    }

    public int stair(int currStair, int targetStair,int[] dp){
        if(currStair == targetStair){     //Check if dp[currStair]
            return 1;                   //position is not equal
        }                                //to -1, then return it.
        if(currStair > targetStair){    //Check if dp[currStair]
            return 0;                   //position is not equal
        }

        if(dp[currStair]==-1){           //Check if dp[currStair]
            return dp[currStair];       //position is not equal
        }

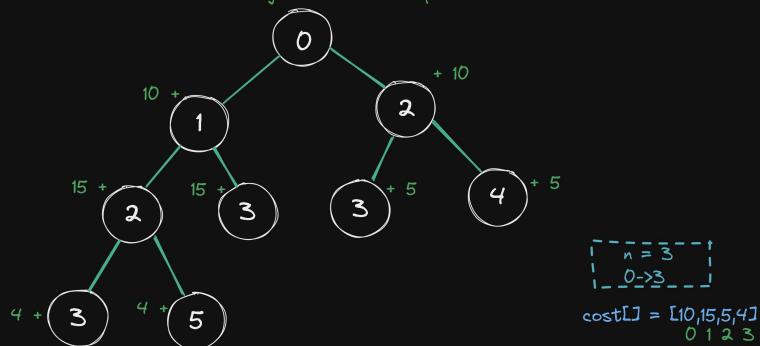
        int oneStep = stair(currStair + 1, targetStair,dp);
        int twoStep = stair(currStair + 2, targetStair,dp);

        dp[currStair] = oneStep + twoStep;
        return dp[currStair];
    }
}
```

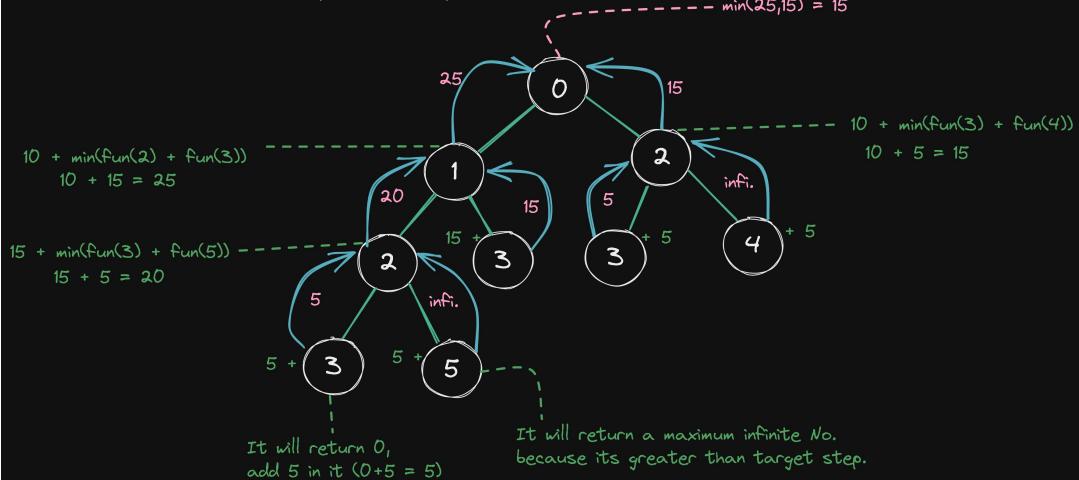
### Min. Cost Climbing Stairs

We have given an array of cost, cost[i] is cost of i'th step on stair.  
 Note:- One You have paid the cost you can either climb one or two step.

Note:- Everytime we move to left  
 we takes one step. And when ever  
 we move to right take two step.



Note :- we are paying after we visit that path. We don't need  
 to pay the cost if you are that step.



Time Complexity:-  $O(2^N)$

Space Complexity:-  $O(N)$

### Code:-

```

class Solution {
    public int minCostClimbingStairs(int[] cost) {
        HashMap<Integer, Integer> memo = new HashMap<Integer, Integer>();
        int a = minCost(cost, 0, cost.length, memo);
        int b = minCost(cost, 1, cost.length, memo);
        return Math.min(a, b);
    }

    public int minCost(int[] cost, int currStair, int targetStair, HashMap<Integer, Integer> memo) {
        if(currStair == targetStair) {
            return 0;
        }
        if(currStair > targetStair) {
            return 1001;
        }
        if(memo.containsKey(currStair)) {
            return memo.get(currStair);
        }
        int oneStep = cost[currStair] + minCost(cost, currStair+1, targetStair, memo);
        int twoStep = cost[currStair] + minCost(cost, currStair+2, targetStair, memo);
        memo.put(currStair, Math.min(oneStep, twoStep));
        return memo.get(currStair);
    }
}

```

## 198. House Robber.

- We cannot rob two adjacent houses.

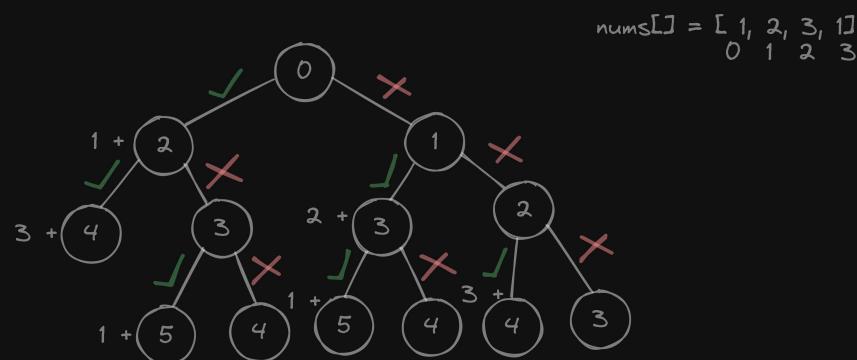
Example 1:

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

Output: 4

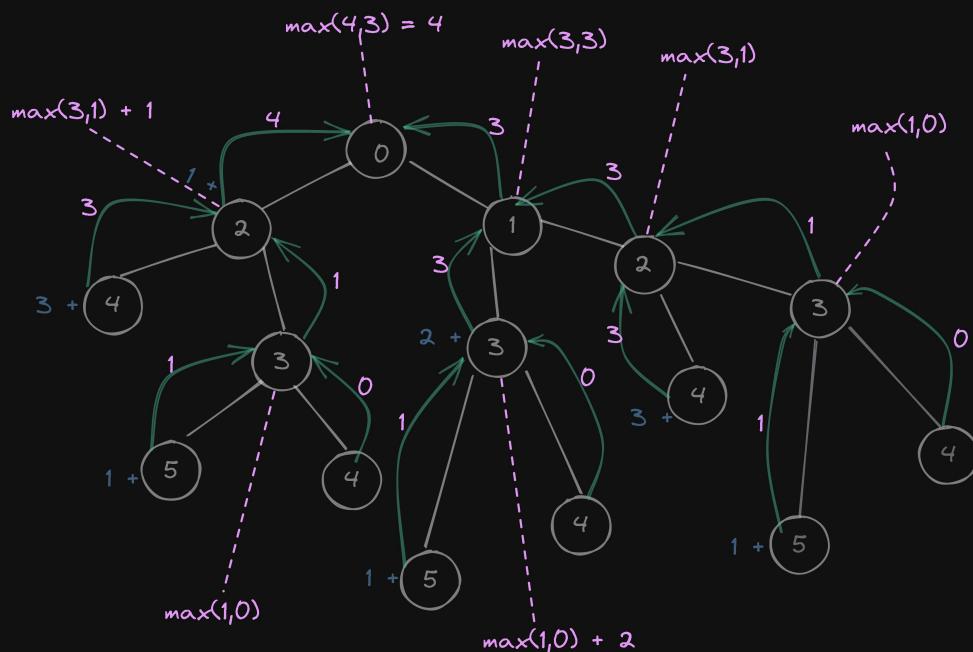
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob =  $1 + 3 = 4$ .



$\begin{matrix} \text{nums}[0] &= [1, 2, 3, 1] \\ 0 & 1 & 2 & 3 \end{matrix}$

If recursion reaches indices 4 or 5, return 0 (because indices 4 and 5 do not exist in the input array).



Code using Recursion:-

- <https://leetcode.com/submissions/detail/566673896/>

Time Complexity:-  
 $O(2^n)$

Space Complexity:-  
 $O(n)$

Code using Memoization + HashMap:-

- <https://leetcode.com/problems/house-robber/submissions/>

Time Complexity:-  
 $O(n)$

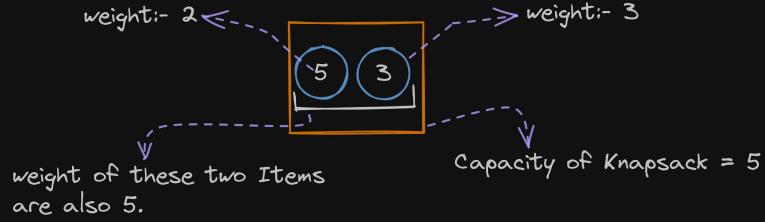
Space Complexity:-  
 $O(n)$

## 0-1 Knapsack

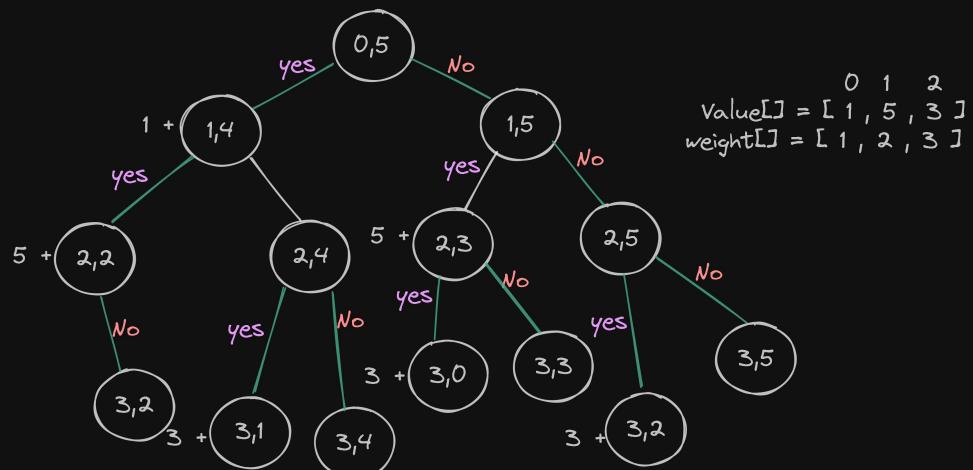


We Have to create max. profit from this Knapsack with capacity 5.

So, from the above diagram we can say that the max. profit is 8,  
 How?

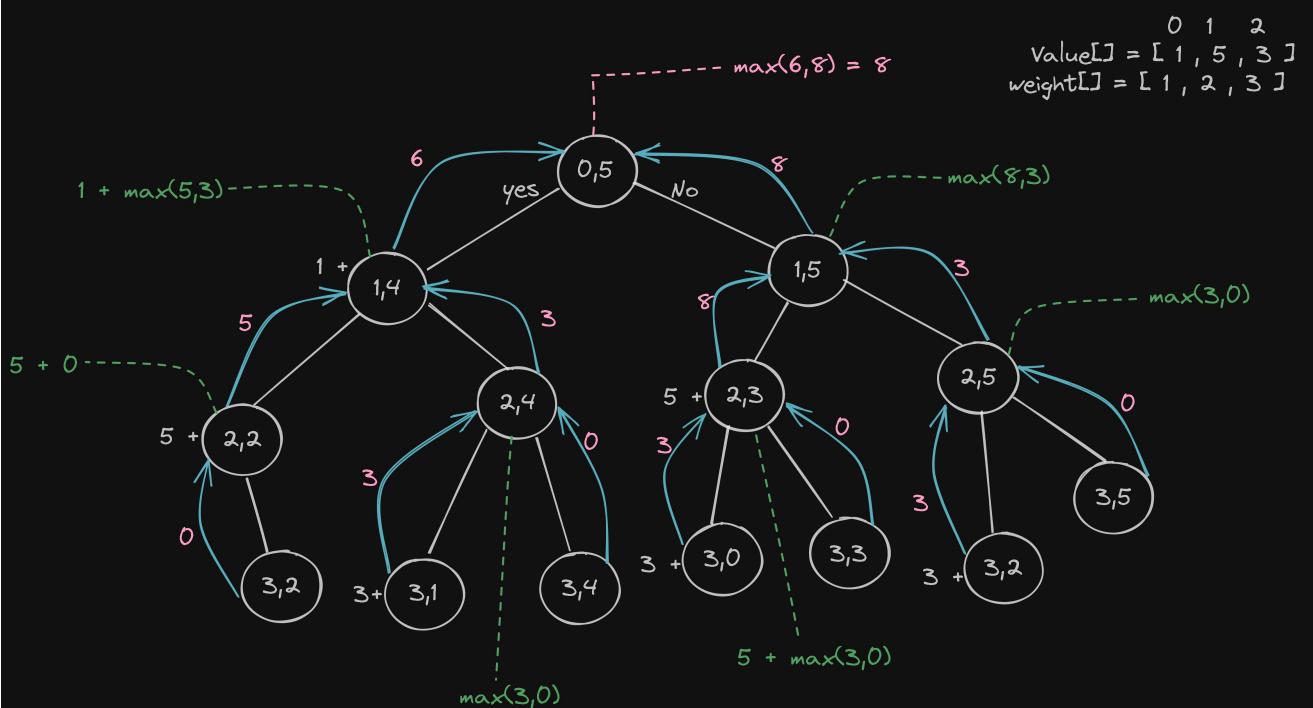


we have two option weather to take element in knapsack or not



\*  $n = 0$   
 \* capacity = 5

Note:- when ever  $n > \text{value.length}$  return 0,  
 when ever capacity become -negative don't create that branch.



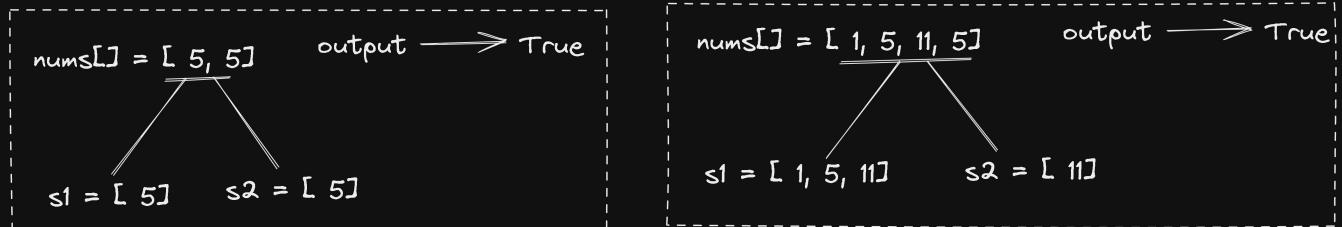
## Partition Equal Subset Sum :-

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

2nd check:- if sum is even then make recursive call.

So, if given array sum is 10.

$$s1 + s2 = \text{sum}(\text{nums});$$

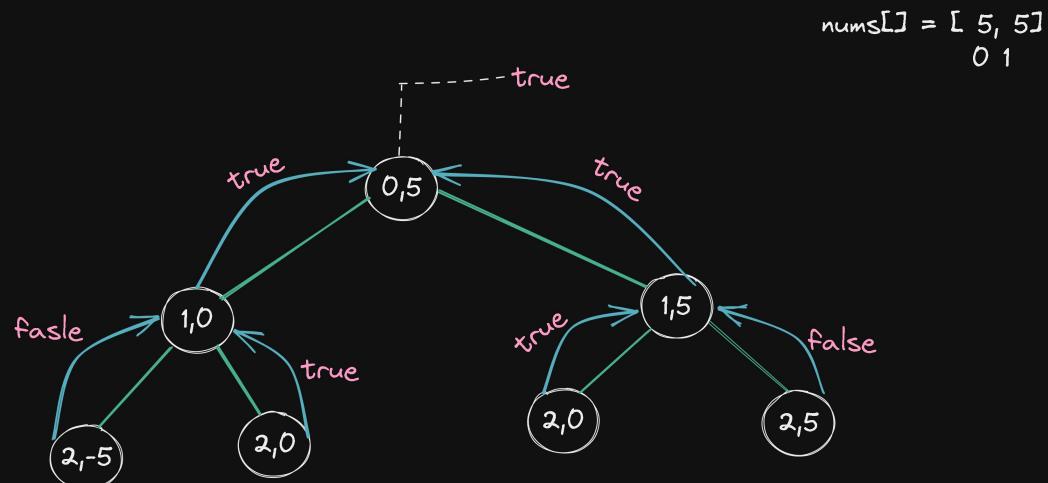


$$\text{sum}(s1) + \text{sum}(s2) = \text{sum}(\text{nums})$$

$$\text{sum}(s1) + \text{sum}(s2) = \text{sum} = (\text{nums})$$

$$2\text{sum}(s1) = \text{sum}(\text{nums})$$

$$\text{sum}(s1) = \text{sum}(\text{nums})/2;$$



NOTE:- If our left half give's us "true" , then just "return true" from there no need to check for the right half then.

Like, if we get 5 from left half , that's means that we diffenately going to get 5 from other half as well.

"All DP problem can be solve using backtracking. But, we cannot solve backtracking problem using DP. "

And Note that Knapsack is Only Valid for positive input.

## 494. Target Sum

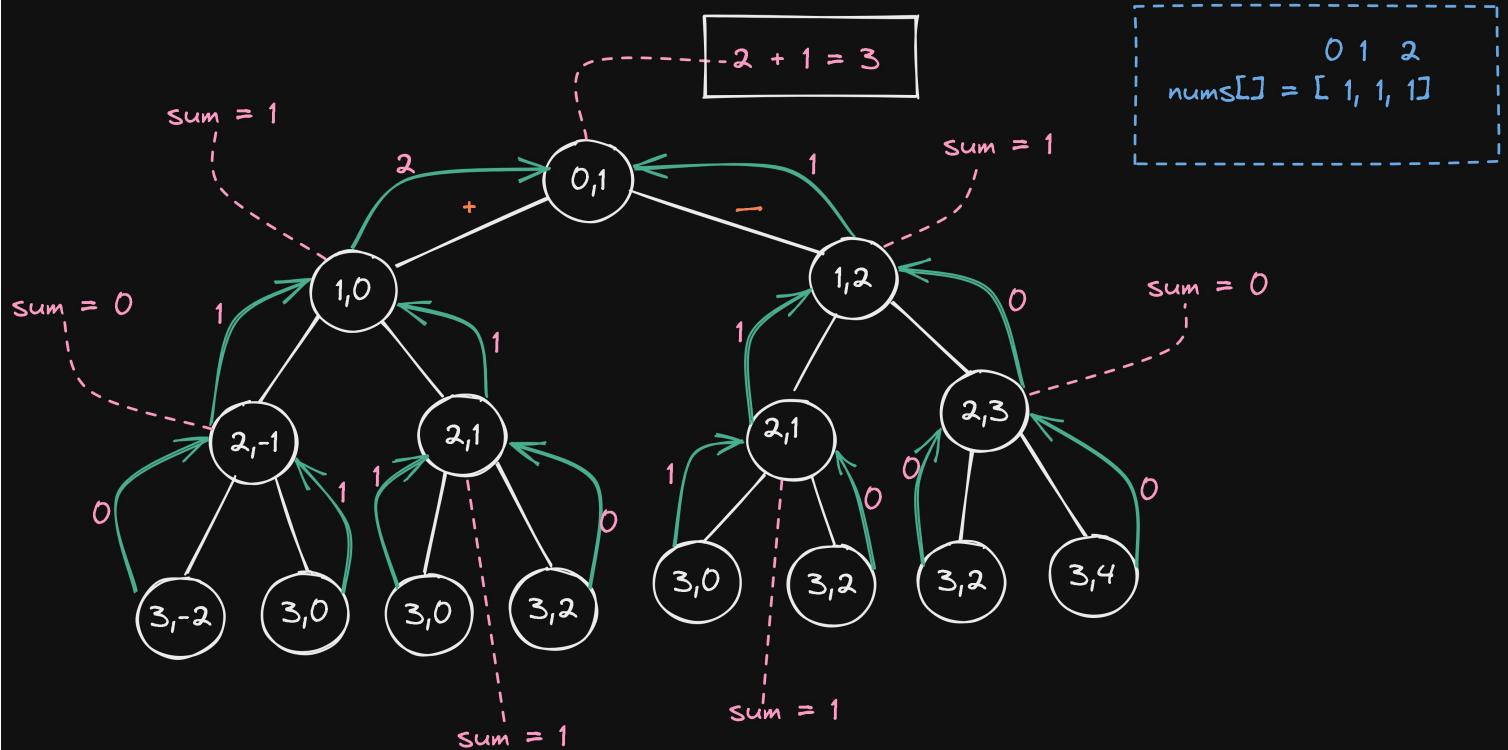
$\text{nums}[] = [1, 1, 1]$

$\text{target} = 1$

$$\begin{aligned}-1 + 1 + 1 &= 1 \\ 1 - 1 + 1 &= 1 \\ 1 + 1 - 1 &= 1\end{aligned}$$

Output: 3

$\text{nums}[] = \begin{matrix} 0 & 1 & 2 \\ 1, 1, 1 \end{matrix}$



Left branch =  $n+1, \text{target} - \text{nums}[n]$

Right branch =  $n+1, \text{target} - (-\text{nums}[n])$

$n+1, \text{target} + \text{nums}[n]$

## Un-Bounded KnapSack

Unbounded Knapsack  $\longrightarrow$  Repetition of items allowed

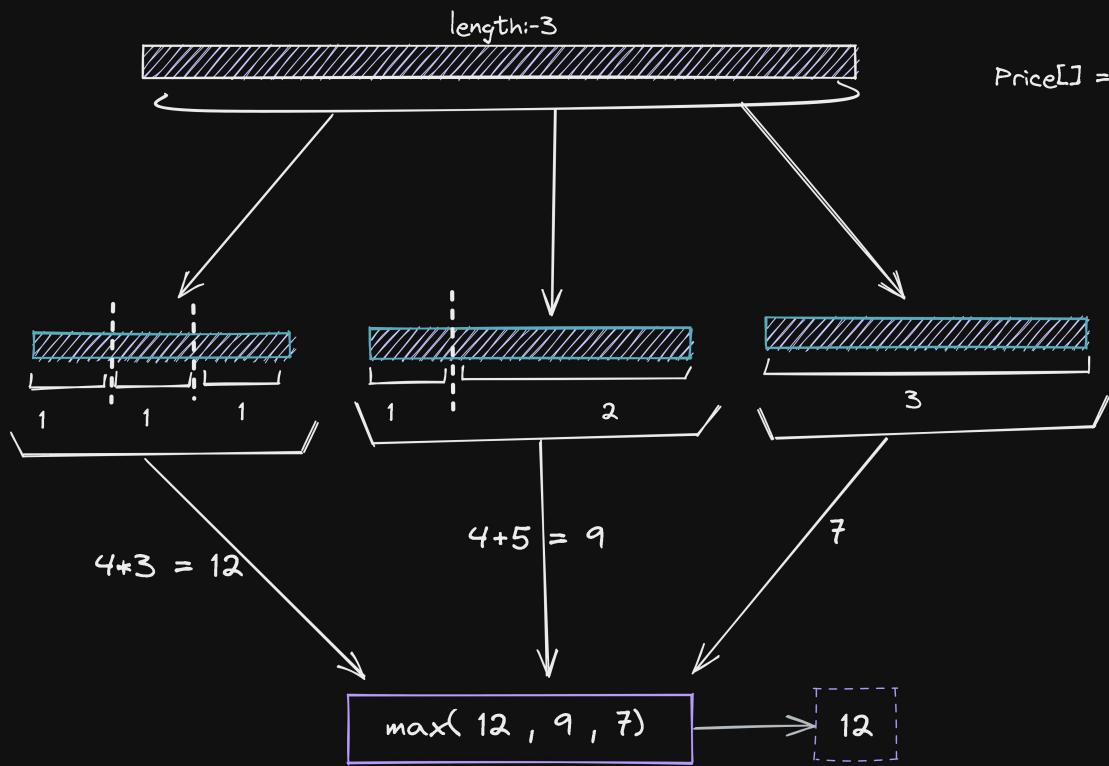
Ques. Suppose a rod of Length 3 , we have to cut that rod to get maximum profit,  
The price is mention below for each cut:-

| cuts | Price | index |
|------|-------|-------|
| 1    | 4     | 0     |
| 2    | 5     | 1     |
| 3    | 7     | 2     |

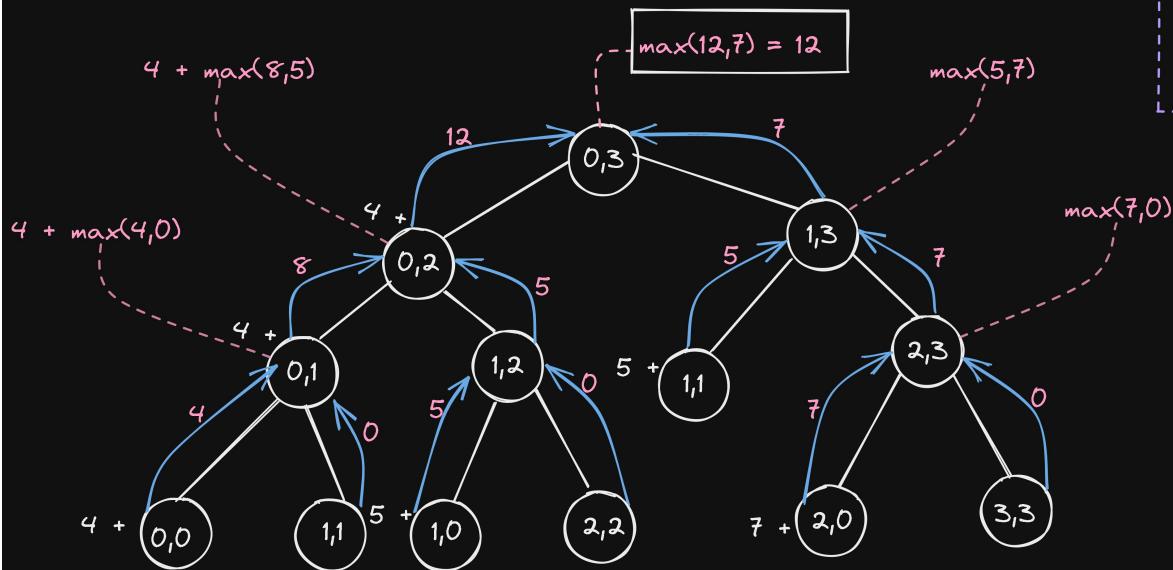
Output → 12

0 1 2 3

Price[] = [ 4, 5, 7, 2 ]



## Rode cutting Problem: -



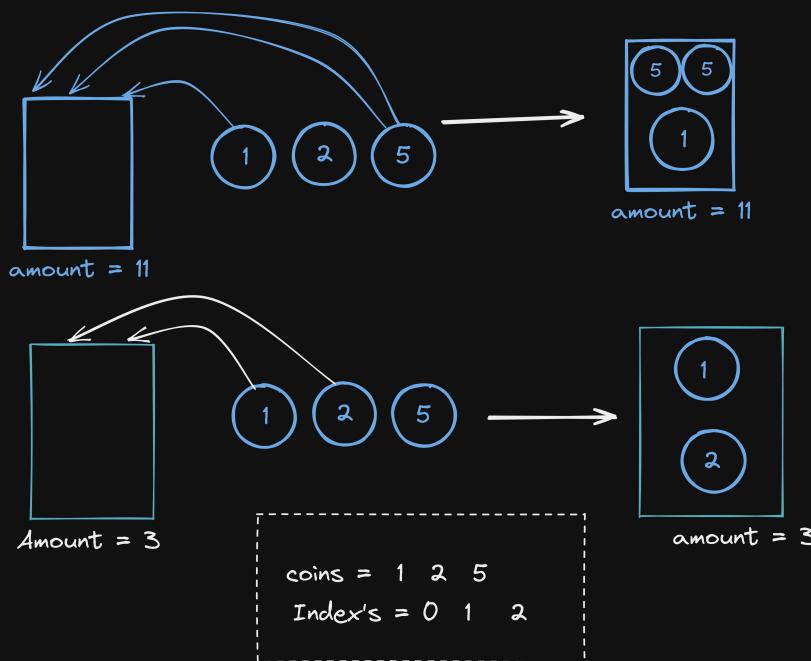
index's: - 0 1 2 3  
price[] = [ 4, 5, 7, 2 ]  
cut's: - 1 2 3 4

## Coin Change:-

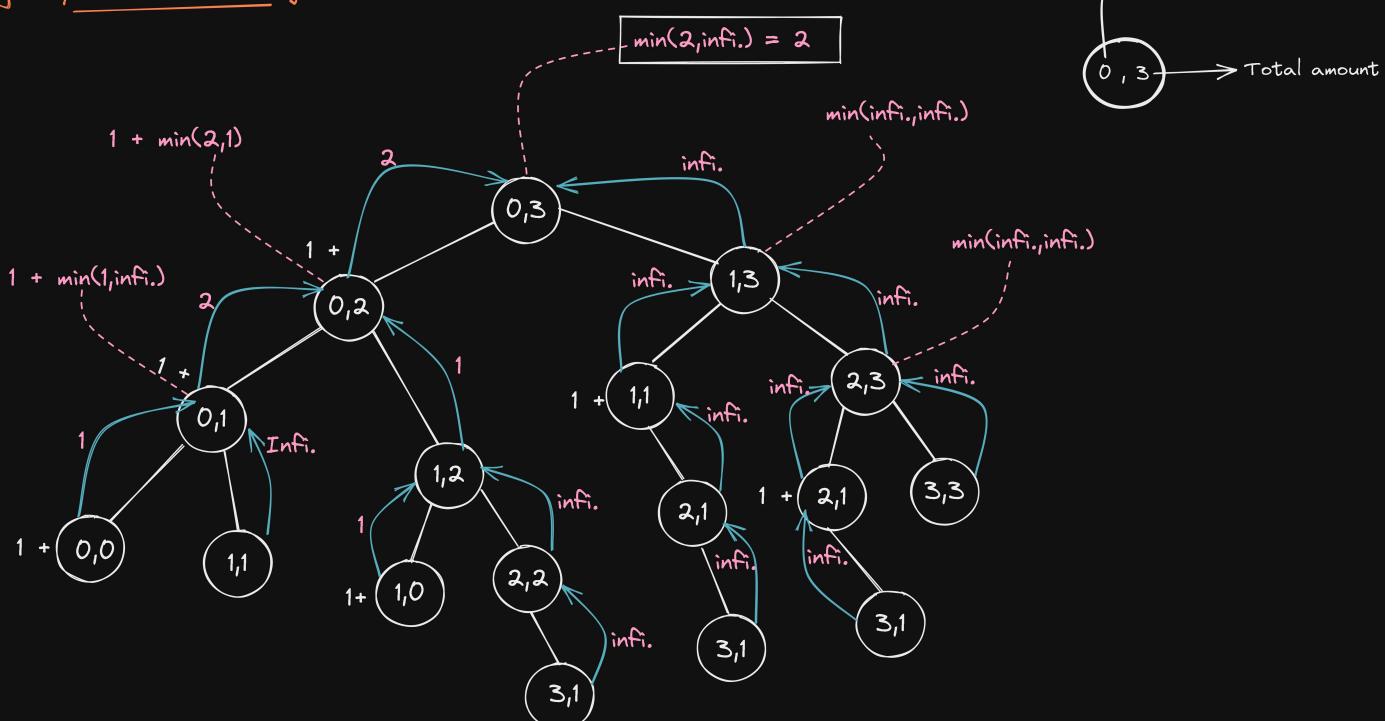
You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount.

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



## Dry Run of coin change:-



### Base Case:-

```

if(amount == 0){
    return 0;
}

if(currentIndex >= amount.length){
    return 100000;
}
  
```

### Using Recursion:-

Time Complexity:-  $O(2^n)$   
Space complexity:-  $O(\text{amount})$

### Using DP:-

Time Complexity:-  $O(n*k)$   
Space complexity:-  $O(n*k)$

It might be possible it will return INFINITY also so , for that use if condition and return -1.

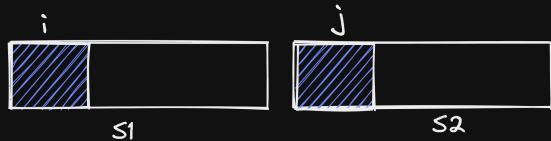
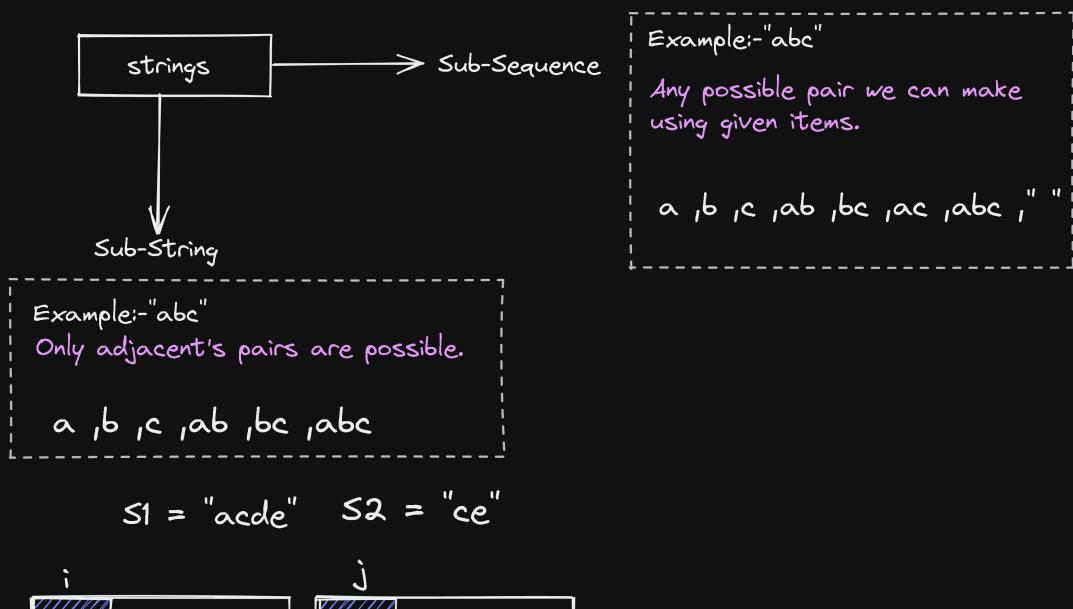
$n = \text{coins.length}$   
 $k = \text{amount}$

## Longest common Subsequence:-

$$S1 = "abcde" \quad S2 = "ce"$$

Output: 2

Explanation: The longest common subsequence is "ace" and its length is 3.



There are only Two possibility:-

$$S1[i] = S2[j]$$

$$S1[i] \neq S2[j]$$

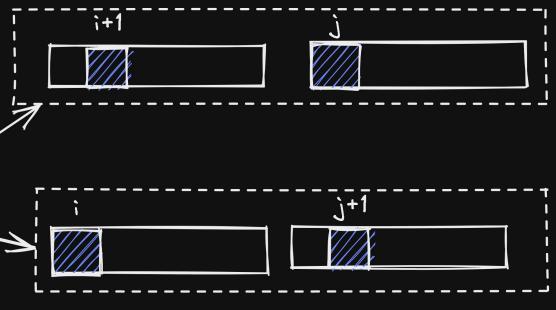
If ( $S1[i] == S2[j]$ )

$$(1 + S1[i+1], S2[j+1])$$

If ( $S1[i] \neq S2[j]$ )

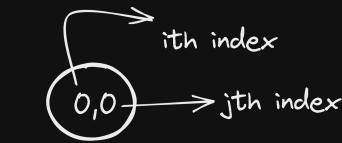
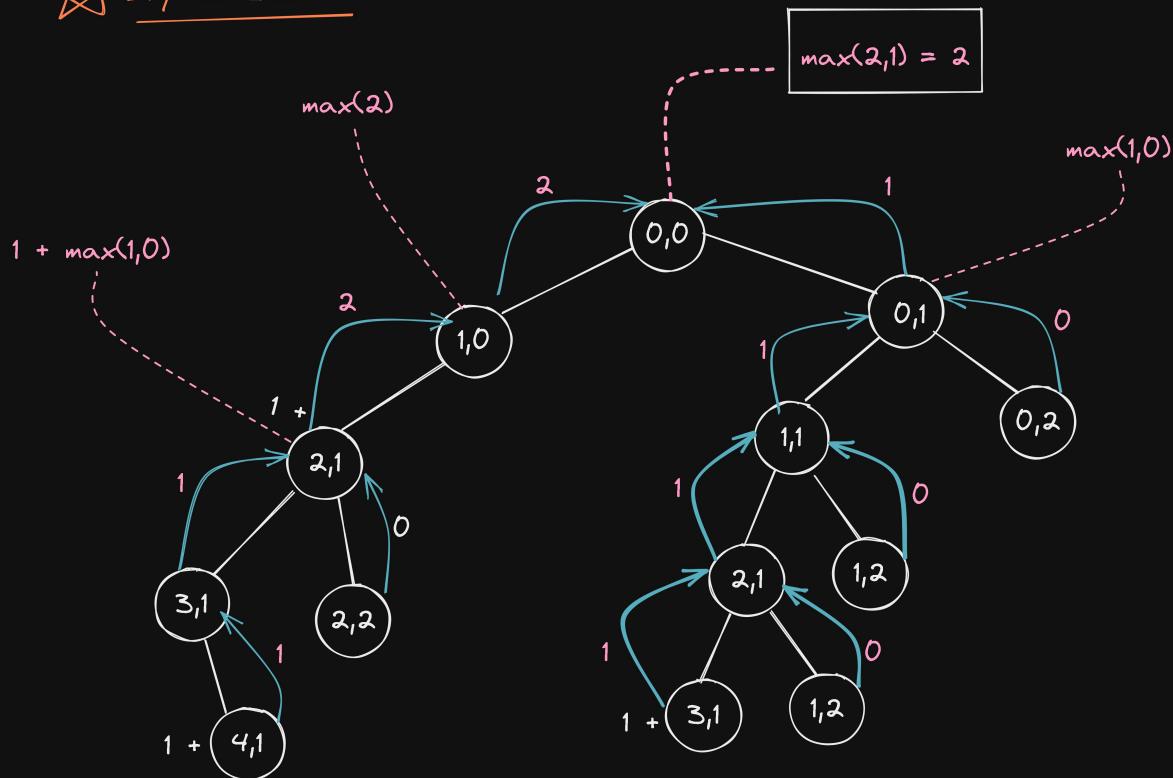
$$(S1[i+1], S2[j])$$

$$(S1[i], S2[j+1])$$



# Longest common Subsequence:-

 Dry Run LCS:-



$S1 = "a c d e"$   
 0 1 2 3  
 $S2 = "c e"$   
 0 1

to checking at any point use this :-

Using DP:-

Time Complexity =  $O(S1.length * S2.length)$   
 Space Complexity =  $O(S1.length * S2.length)$

Using Recursion Only:-

It bit difficult to say time complexity at Recursion Only:- because it vary, at some stage's there only one call, or at some place there are two call.

$$\max(S1.length, S2.length) = S1.length = n;$$

$$\text{Time Complexity} = O(2^n)$$

## 516. Longest Palindromic Subsequence

Input:  $s1 = "cbbd"$

Output: 2

Explanation: One possible longest palindromic subsequence is "cbbd".

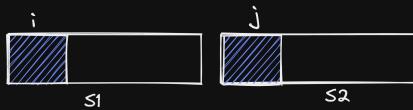
Note:- Twist is we have to return longest palindrome subsequence present in string  $s1$ .

Approach:-

->create another string  $s2$ , which is reverse of  $s$  string.

->Apply the same approach of LCS(longest common subsequence).

$s1 = "cbbd"$   $\longrightarrow$  original string.  
 $s2 = "dbbc"$   $\longrightarrow$  reverse of  $s1$ .



There are only Two possibility:-

$s1[i] = s2[j]$

$s1[i] \neq s2[j]$

If ( $s1[i] == s2[j]$ )

( $i + 1, s1[i+1], s2[j+1]$ )

If ( $s1[i] \neq s2[j]$ )

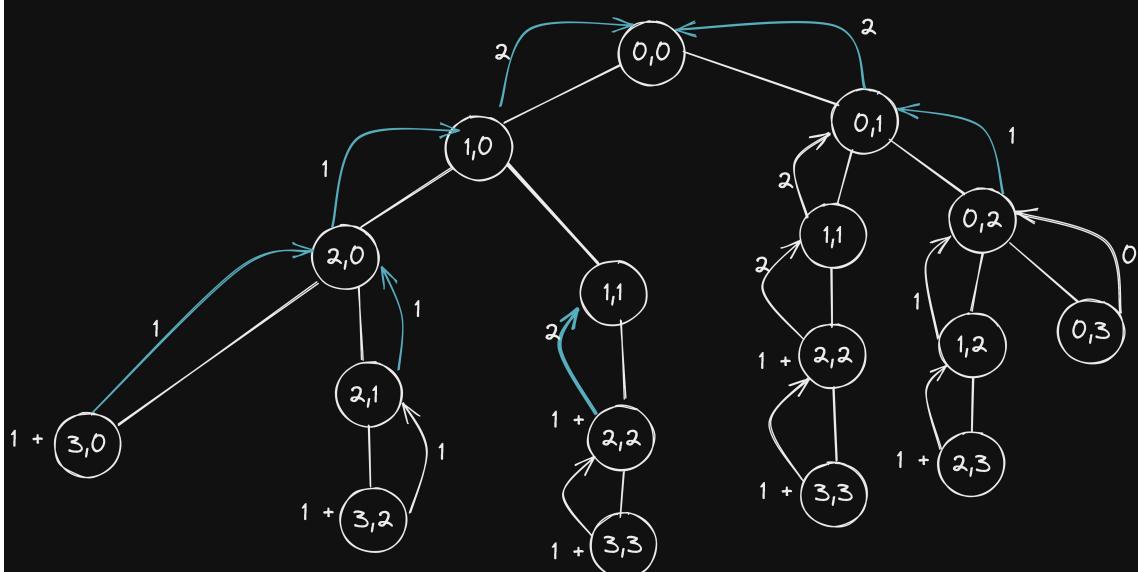
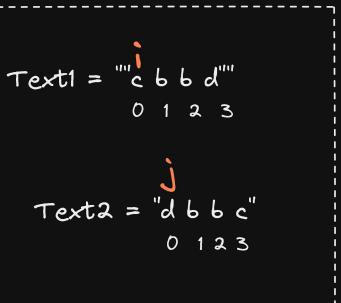
( $s1[i+1], s2[j]$ )

( $s1[i], s2[j+1]$ )



Dry Run LPS:-

if  $i == s.length \text{ or } j == s.length$   
return 0;



## Jump Game

Its Optimal Approach  $\longrightarrow$  Greedy  
Native Approach  $\longrightarrow$  Dynamic Programming

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

->Reach at last Index, Return "true".

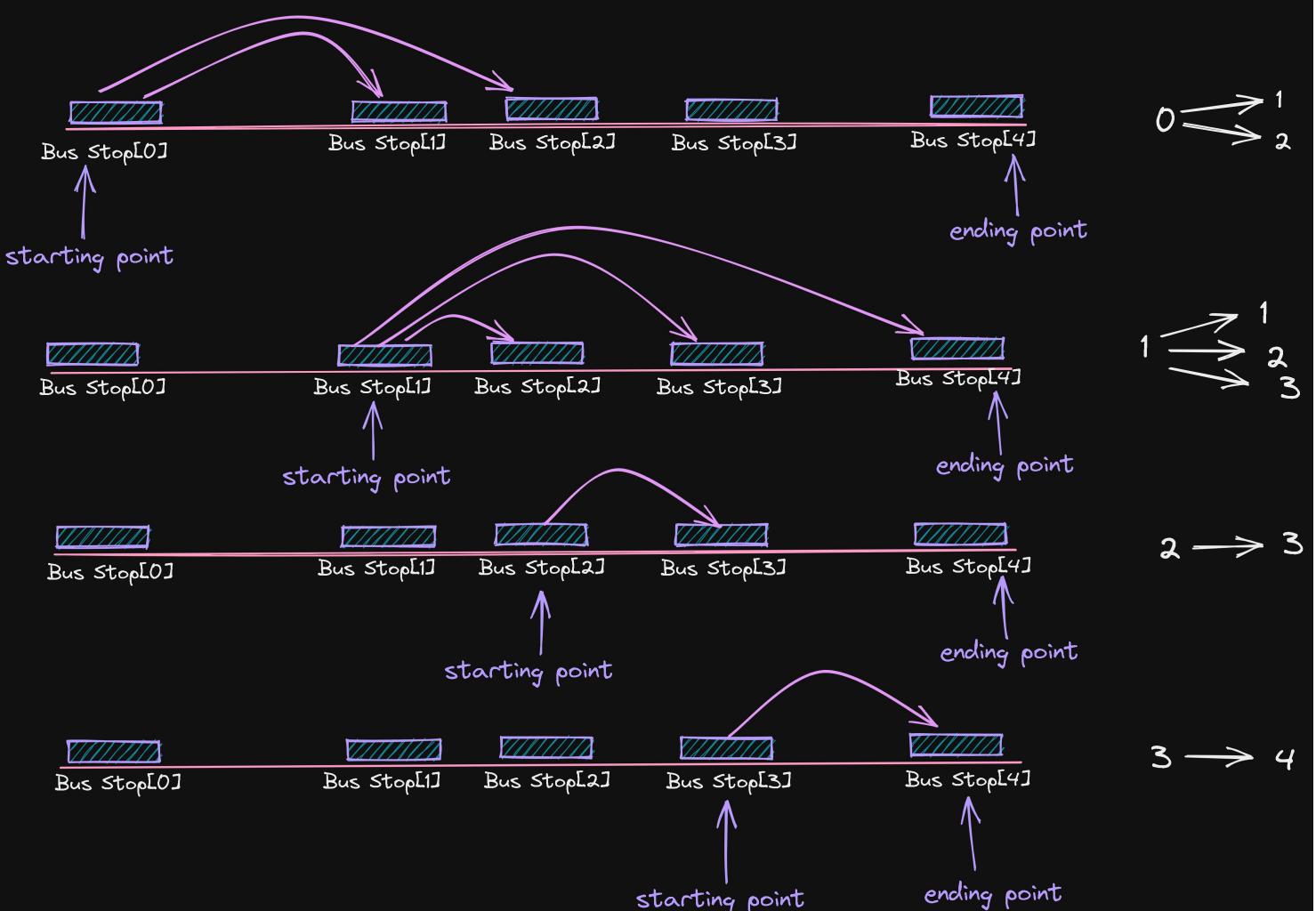
->If not reach at last Index, Return "false".

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

★ Take a Example of Bus :-

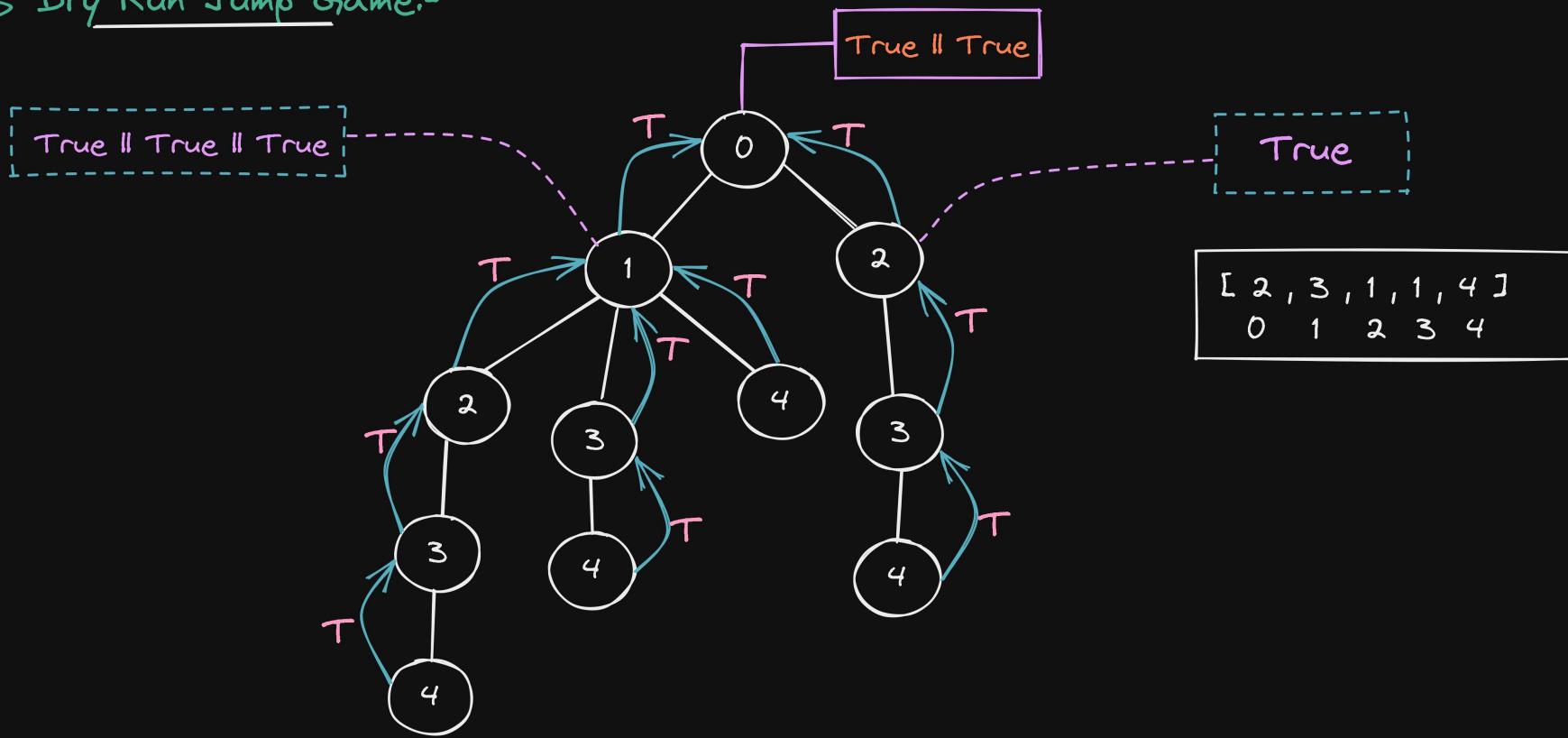
Like, from Bus-Stop[0] we can go to bus-stop[1] or bus-stop[2]

[ 2 , 3 , 1 , 1 , 4 ]  
0 1 2 3 4





## Dry Run Jump Game:-



Time-Complexity  $\longrightarrow$  Exponential( $2^n / 3^n \dots$ )

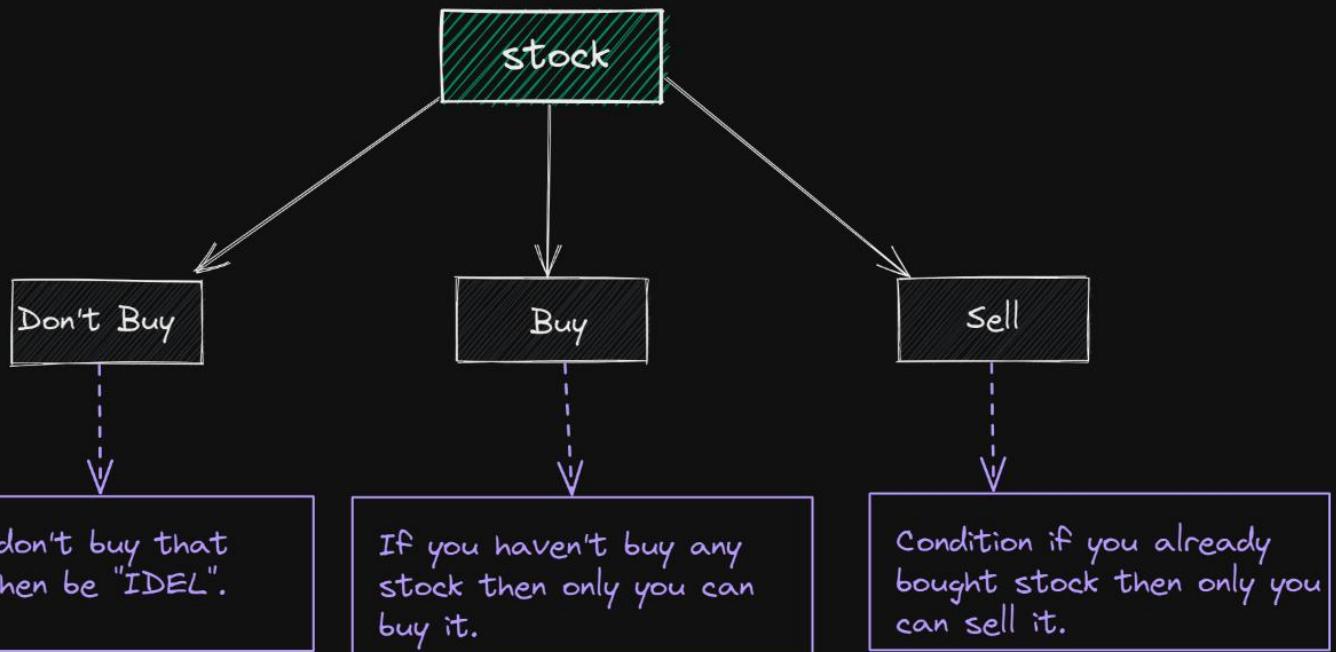
Space-Complexity  $\longrightarrow$   $O(n)$

There are unsartain number  
of branches.

Look at Largest branch.( `nums.length` )

## Best Time to Buy and Sell Stock I

- we have to buy & sell stock to get maximum amount of profit.  
In just one transaction.
- we have given an price array  $\text{price}[i]$ , where  $i$ th is the day.



Note:- Only one transaction is allowed.  
Buy & Sell  $\longrightarrow$  1 transaction

Index's(Day) : 0 1 2 3 4 5

prices(amount) : [ 7 , 1 , 5 , 3 , 6 , 4 ]

Output(profit) : 5

$$[ 7 , 1 , 5 , 3 , 6 , 4 ] \longrightarrow -(Buy) + (Sell) = \text{Profit}$$

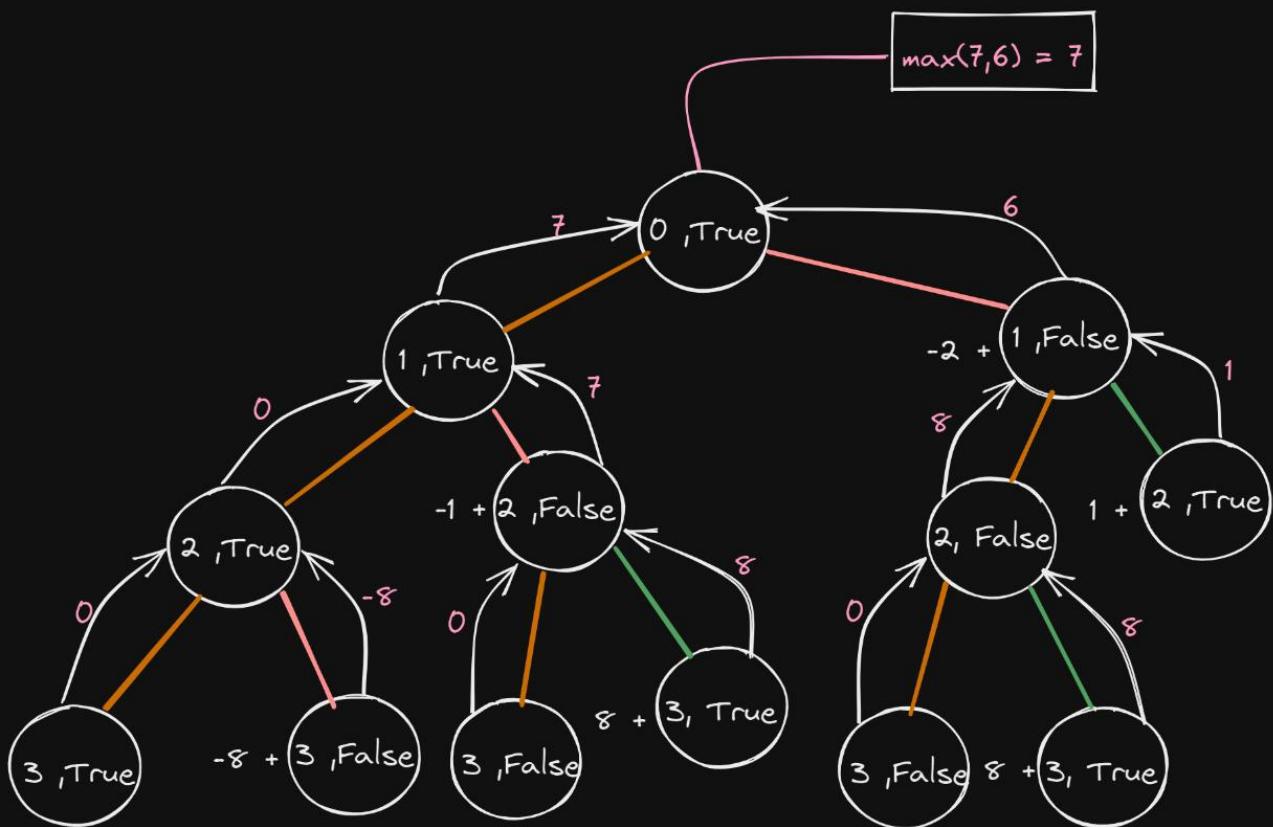
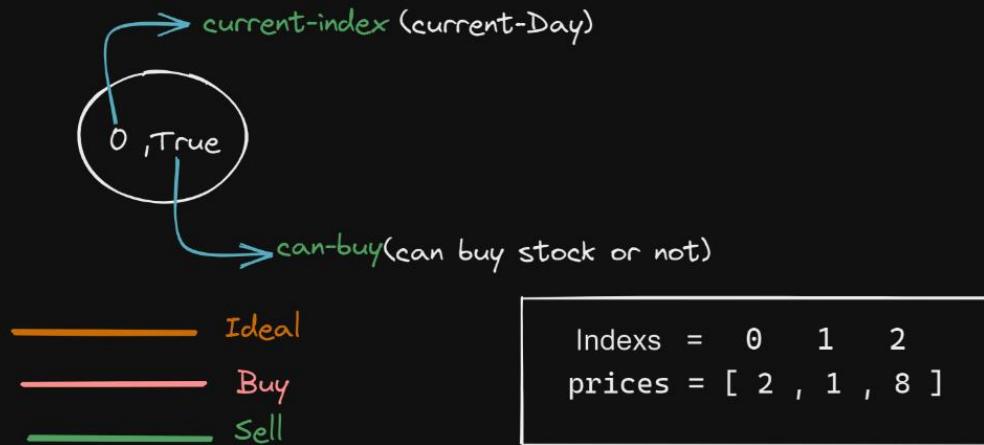
$\uparrow \qquad \uparrow$   
Buy      Sell

$$-1 + 6 = 5$$

Note:- we have to check multiple ways (Recursion) to get maximum amount of profit.

## Best Time to Buy and Sell Stock 2

We can do infinite number of transaction in this question.



Conditions:-

```

if(currentIndex >= prices.length){
    return 0;
}

if(canBuy == true){
    Ideal
    Buy the Stock;
}
else{
    Ideal
    Sell the Stock;
}

```

Time-complexity :-  $O(2^n)$   
Space-complexity :-  $O(n)$

→ using Recursion

Time-complexity :-  $O(n^2)$   
Space-complexity :-  $O(n)$

→ using DP

### 329. Longest Increasing Path in a Matrix

→ Given an  $m \times n$  integers matrix, return the length of the longest increasing path in matrix.

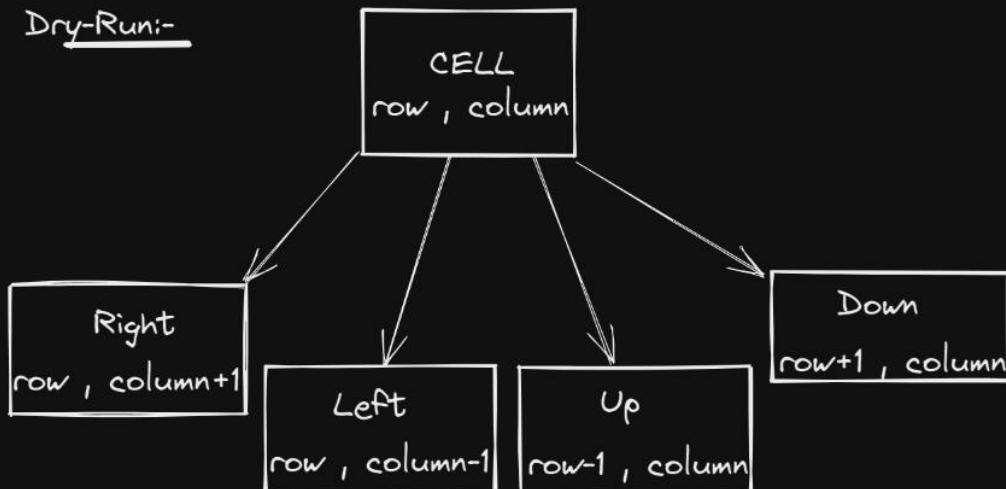
→ From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

Input:-

|   |   |   |
|---|---|---|
| 9 | 9 | 4 |
| 6 | 6 | 8 |
| 2 | 1 | 1 |

Output:- 4 (The longest increasing path is [1, 2, 6, 9].)

Dry-Run:-



NOTE:-

Use PrevVal which is going to keep track of previous Number with current Number.

→ We have to create 4 Recursive call for each cell.

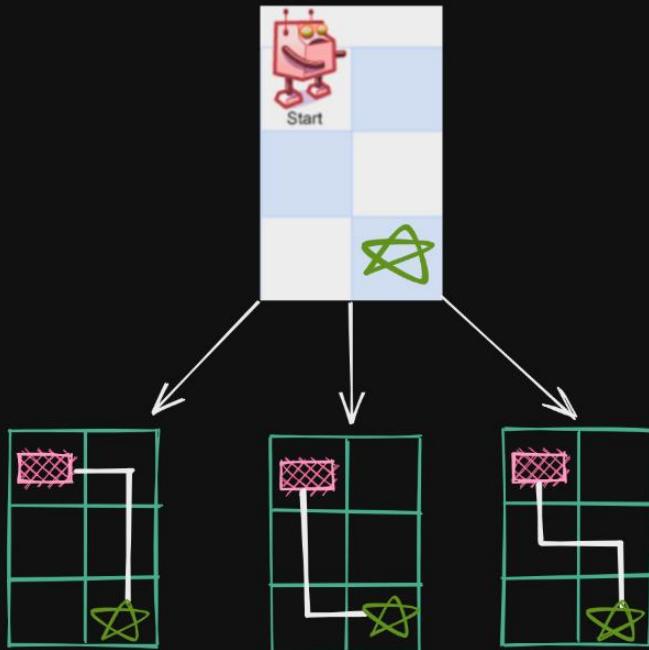
Conditions:-

```
if(currentRow < 0 || currentCol < 0 || currentRow >= m || currentCol >= n){  
    return 0;  
}  
  
if(matrix[currentRow][currentCol] <= prevVal){  
    return 0;  
}
```

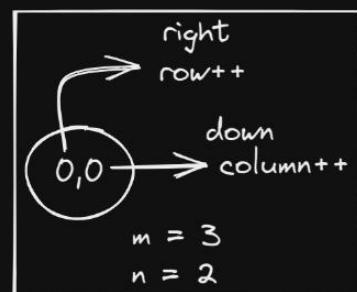
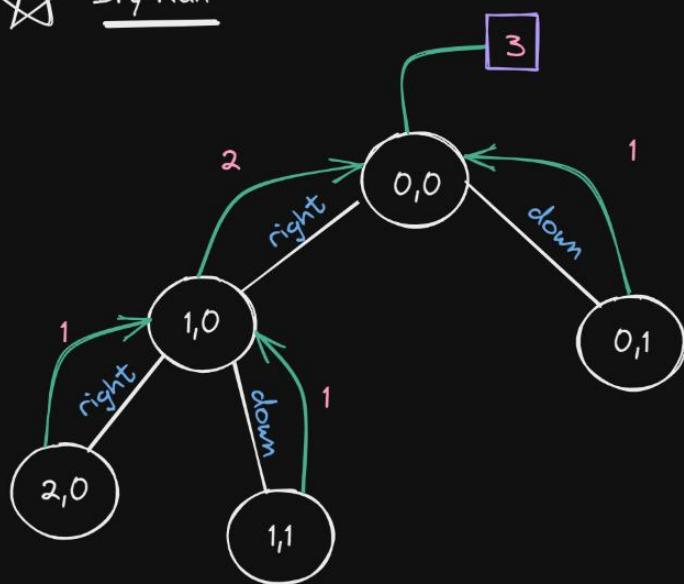
## 62. Unique Paths

- A robot is located at the top-left corner of a  $m \times n$  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 can only move on to right or down only.



Dry-Run



```
conditions:-  
if(row == m-1 || column == n-1){  
    return 1;  
}
```

Time-complexity →  $O(\text{row} * \text{col})$   
Space-complexity →  $O(\max(\text{row}, \text{col}))$

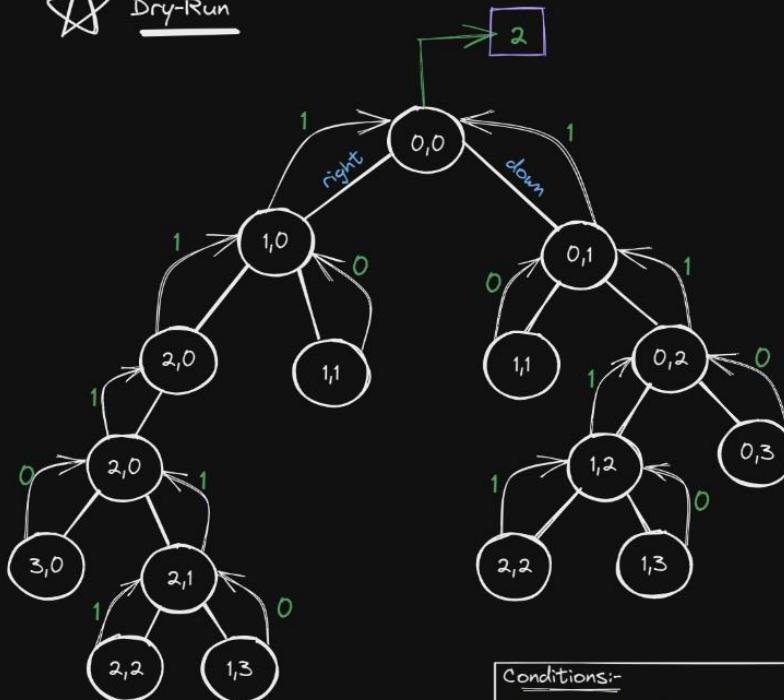
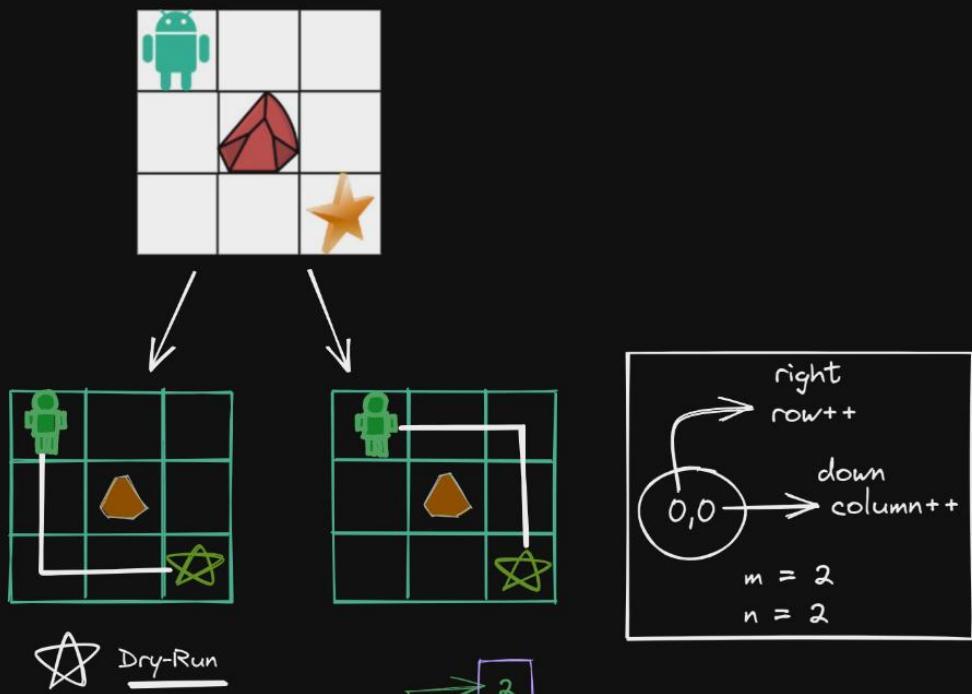
Time-complexity →  $O(2^{(\text{row} * \text{col})})$   
Space-complexity →  $O(\max(\text{row}, \text{col}))$

using DP

using Recursion

## 62. Unique Paths

- A robot is located at the top-left corner of a  $m \times n$  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 corner grid at bottom-right, if there is an obstacle so avoid that path.



Conditions:-

```

if(row > m || col > n){
    return 0;
}

if(arr[row][col] == 1){
    return 0;
}

if(row == m && col == n){
    return 1;
}

```

Using DP

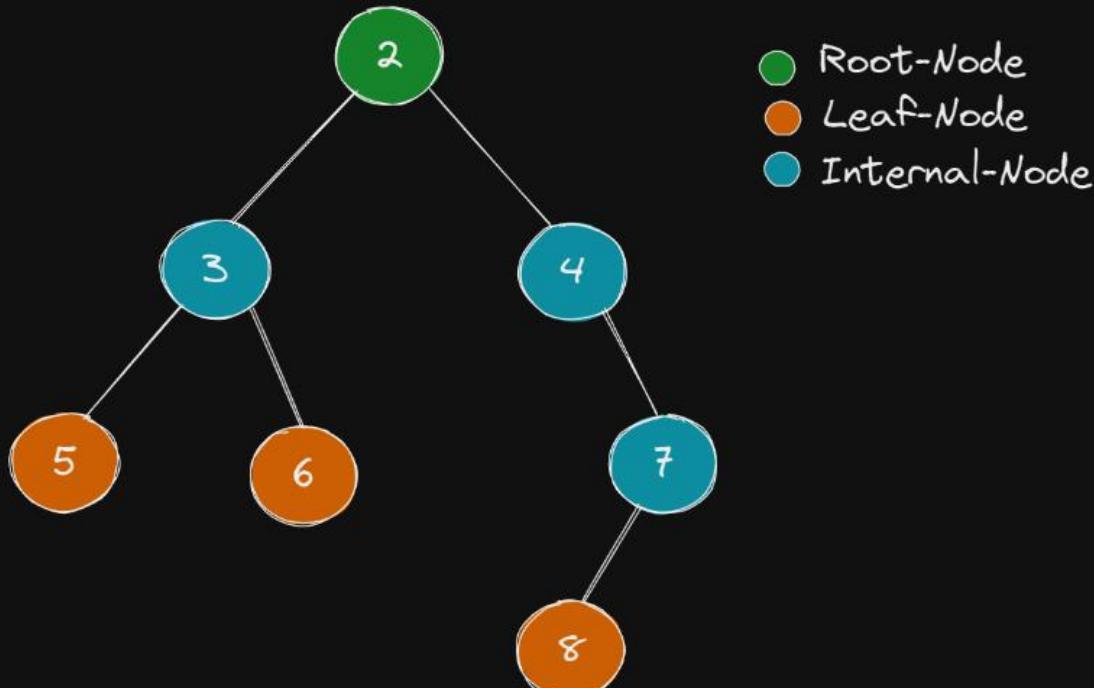
|  |
|--|
| Time-complexity $\rightarrow O(m \times n)$  |
| Space-complexity $\rightarrow O(\max(m, n))$ |

Using Recursion

|  |
|--|
| Time-complexity $\rightarrow O(2^{m+n})$     |
| Space-complexity $\rightarrow O(\max(m, n))$ |

## Binary Tree

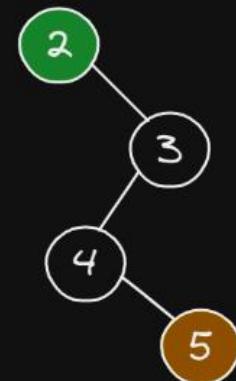
Binary tree is a data structure in which we can store data structure in herarchical manner. Every Node present in Binary-Tree can Have atmost 2 children only.



### Types of Binary-Tree:-

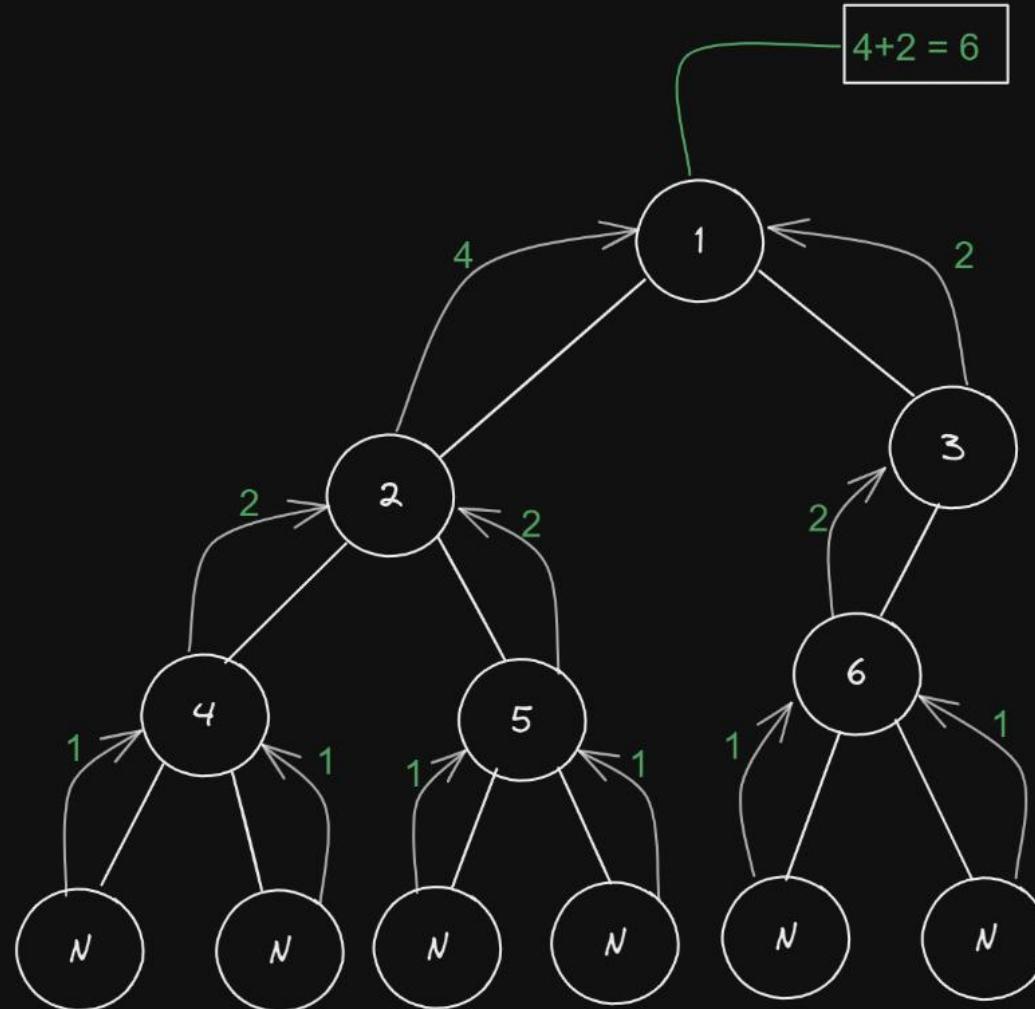
#### 1. Pathological Tree or degenerate

Pathological Tree is a Tree where every parent node has only one child either left or right

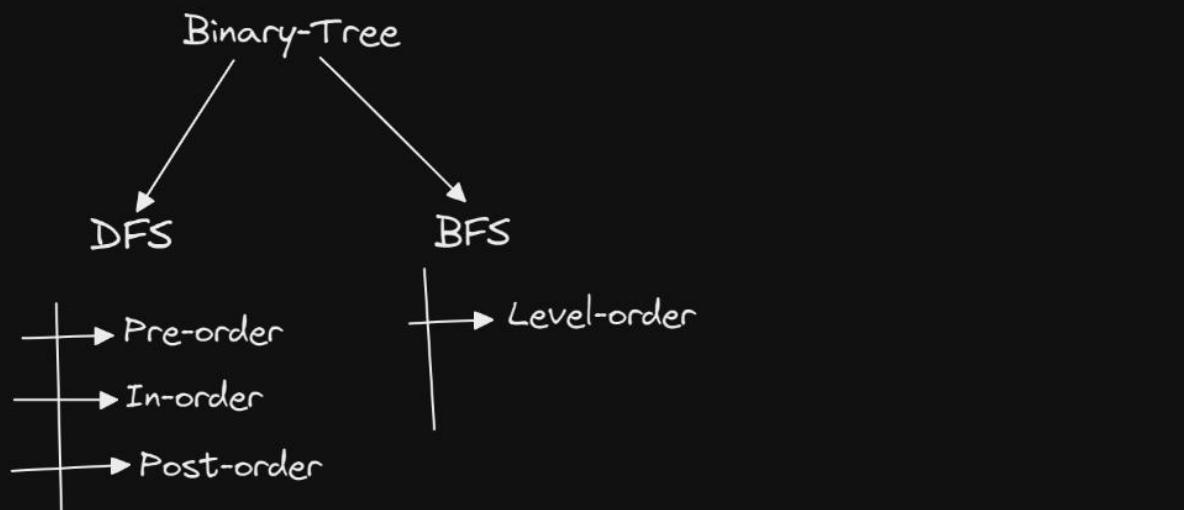


## Count Complete Tree Nodes

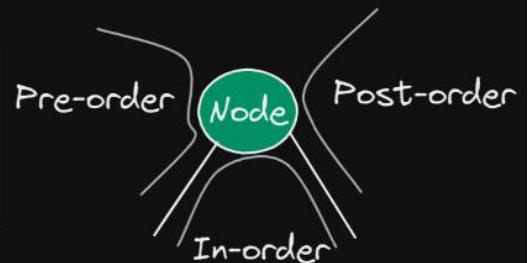
Given the root of a complete binary tree, return the number of the nodes in the tree.



Time-complexity ->  $O(n)$   
Space-complexity -> $O(\text{max-depth})$



- DFS → Depth for Search
- BFS → Breadth for Search

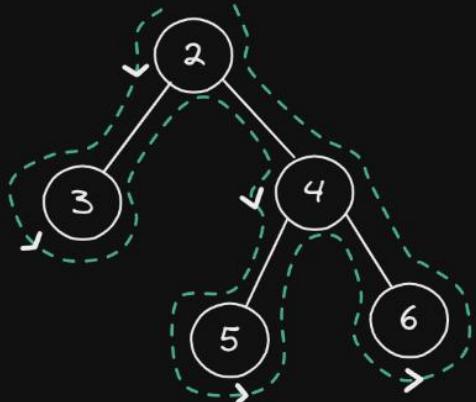


#### ◆ Depth for Search ->

1. Pre-Order :-

→ Node → Left → Right

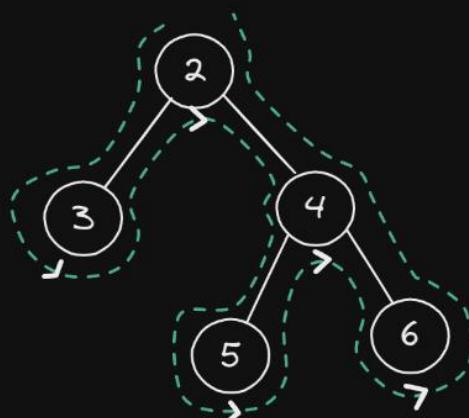
[ 2 , 3 , 4 , 5 , 6 ]



2. In-Order :-

→ Left → Node → Right

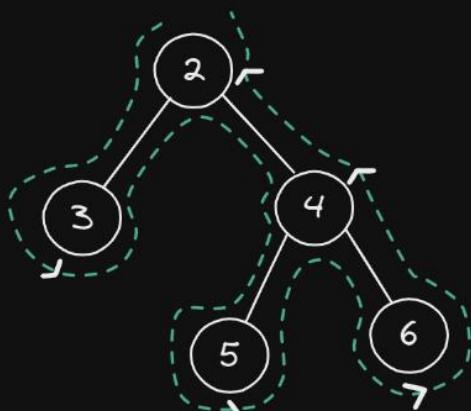
[ 3 , 2 , 5 , 4 , 6 ]



3. Post-Order :-

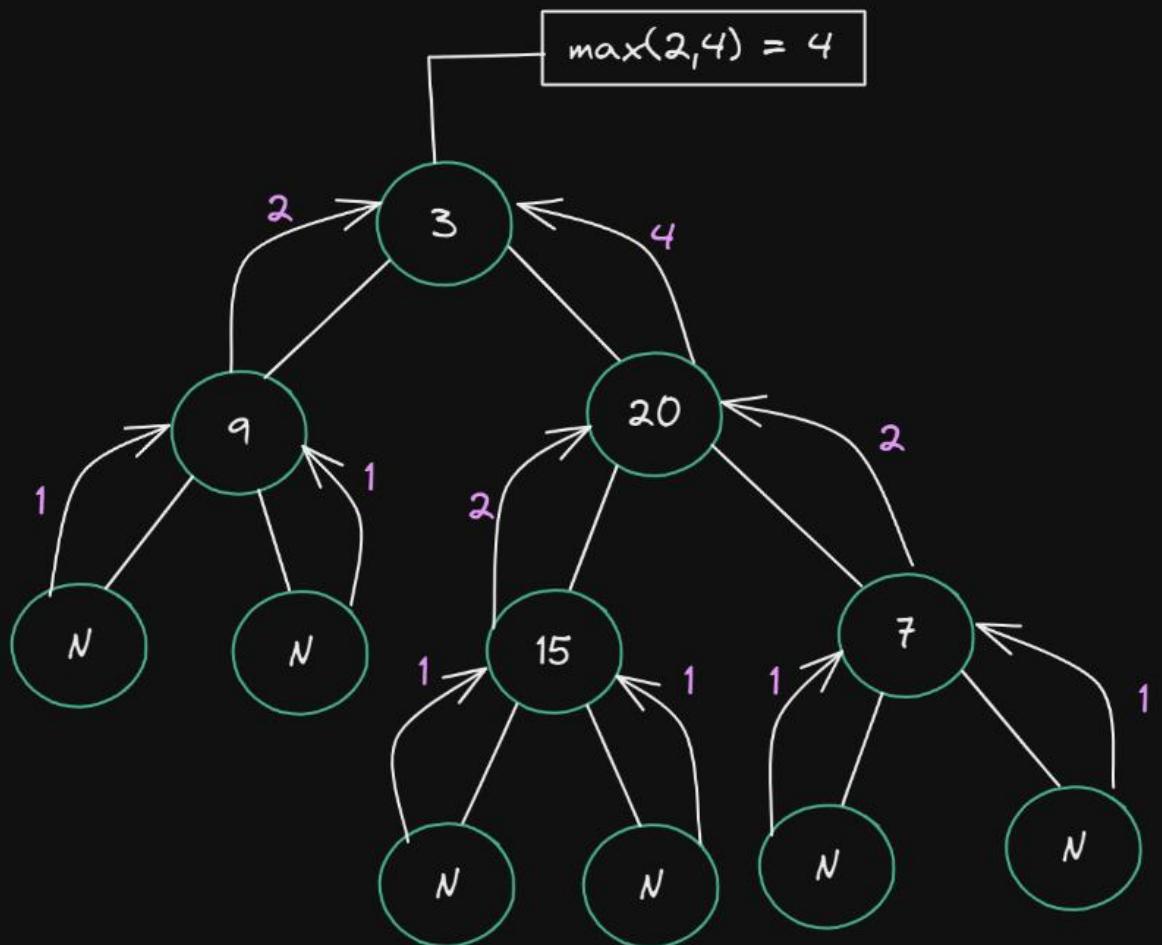
→ Left → Right → Node

[ 3 , 5 , 6 , 4 , 2 ]



## Max-Depth Binary Tree

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.



$N \rightarrow$  Number of Nodes

$1 + \max(L.C, R.C)$

Time-complexity -  $O(N)$

Space-complexity -  $O(\text{Max-Depth})$

