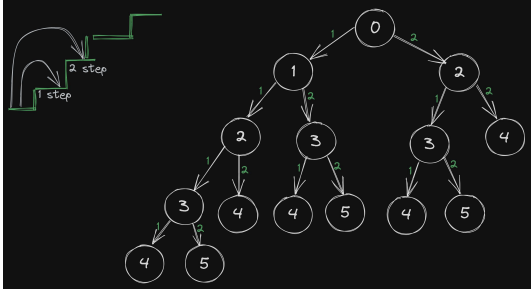
Note That when ever there is choices then we are going to use Recursion.

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

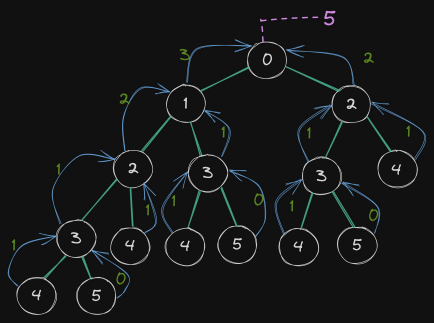
Note:- we have two possible ways either we can take one step or two step at a time.

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.



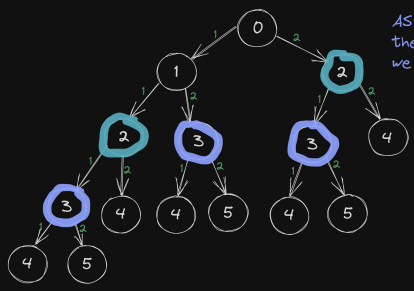
Time Complexity:- O(2^n)
Space Complexity:- O(n)

If we Talk about Auxilary 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){

        if(currStair == targetStair){          // 4 == 4, return 1;
            return 1;
        }                                                          Base Case

        if(currStair > targetStair){           //5 > 4, return 0;
            return 0;
        }

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

        return oneStep + twoStep;       //It will return No. of possible ways.

    }
}
```

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



AS we can see we are repeating the the pattern again and again,while we have solve it earlier.

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

COde USing Memoization :-

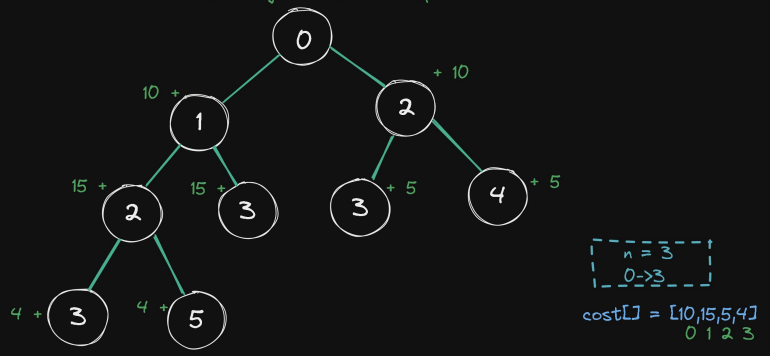
```
class Solution {

    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,n,dp);
    }

    public int stair(int currStair, int targetStair,int[] dp){

        if(currStair == targetStair){
            return 1;
        }

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

        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

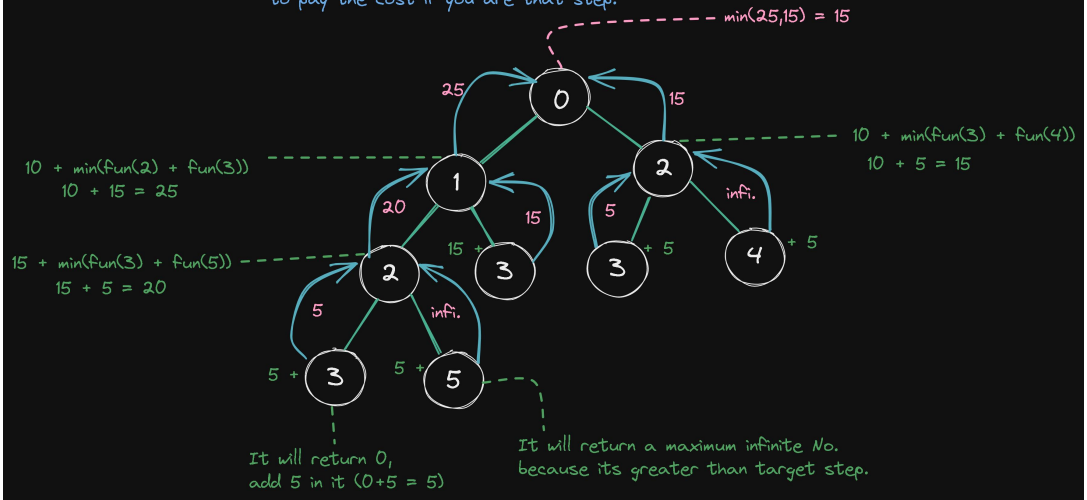
We have given an array of cost, cost[i] is cost of i'th step on stair.

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:-

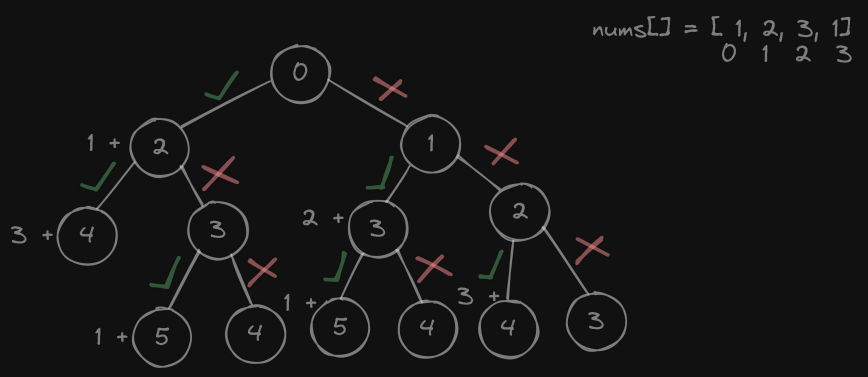
```
class Solution {
    public int minCostClimbingStairs(int[] cost) {

        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.

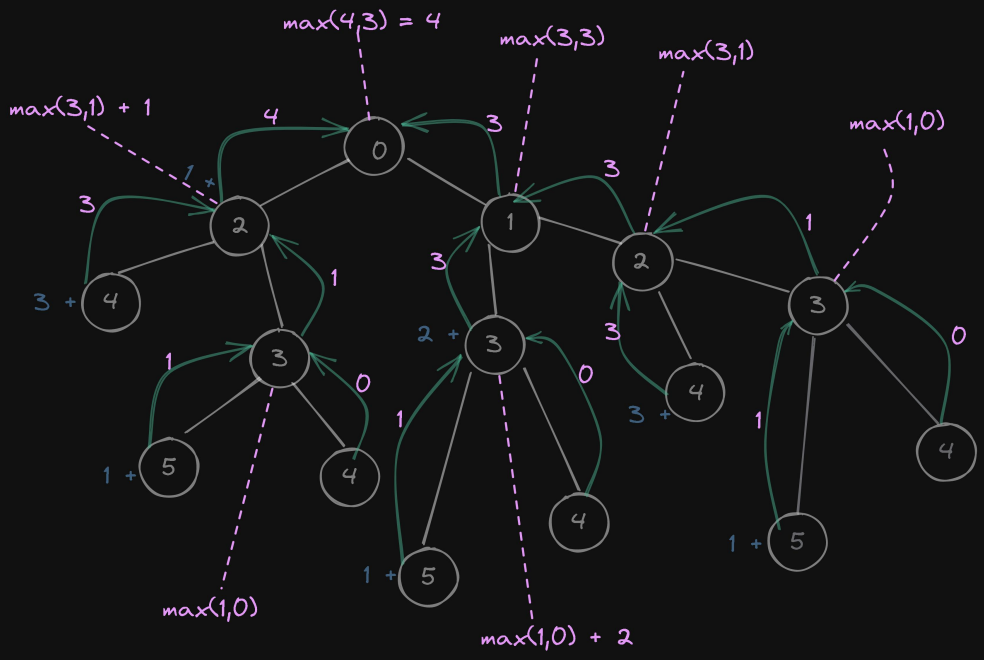
Example 1:

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.



nums[] = [1, 2, 3, 1]
0 1 2 3

If recursion reach 4,5
then, return 0 (becuase 4,5
position doesn't exist in nums[])



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/

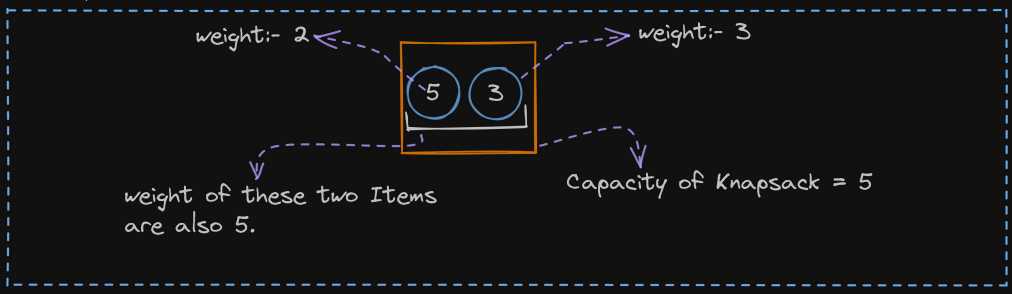
Time Complexity:-
O(n)

Space COmplexity:-
O(n)

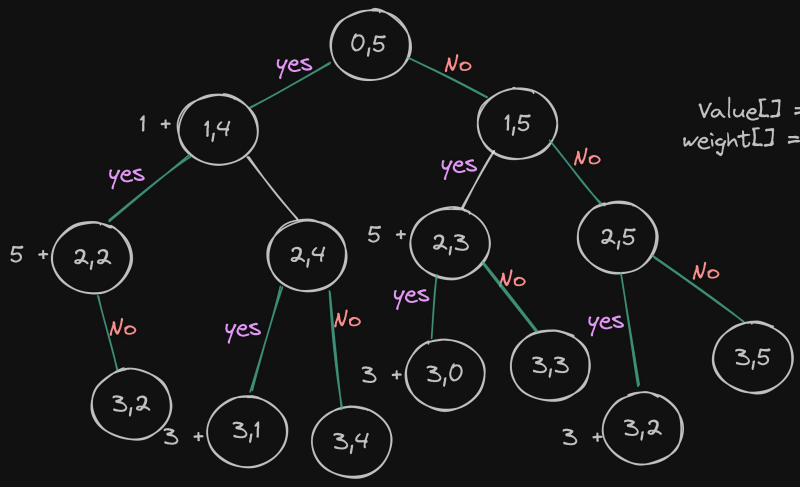
0-1 Knapsack

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?



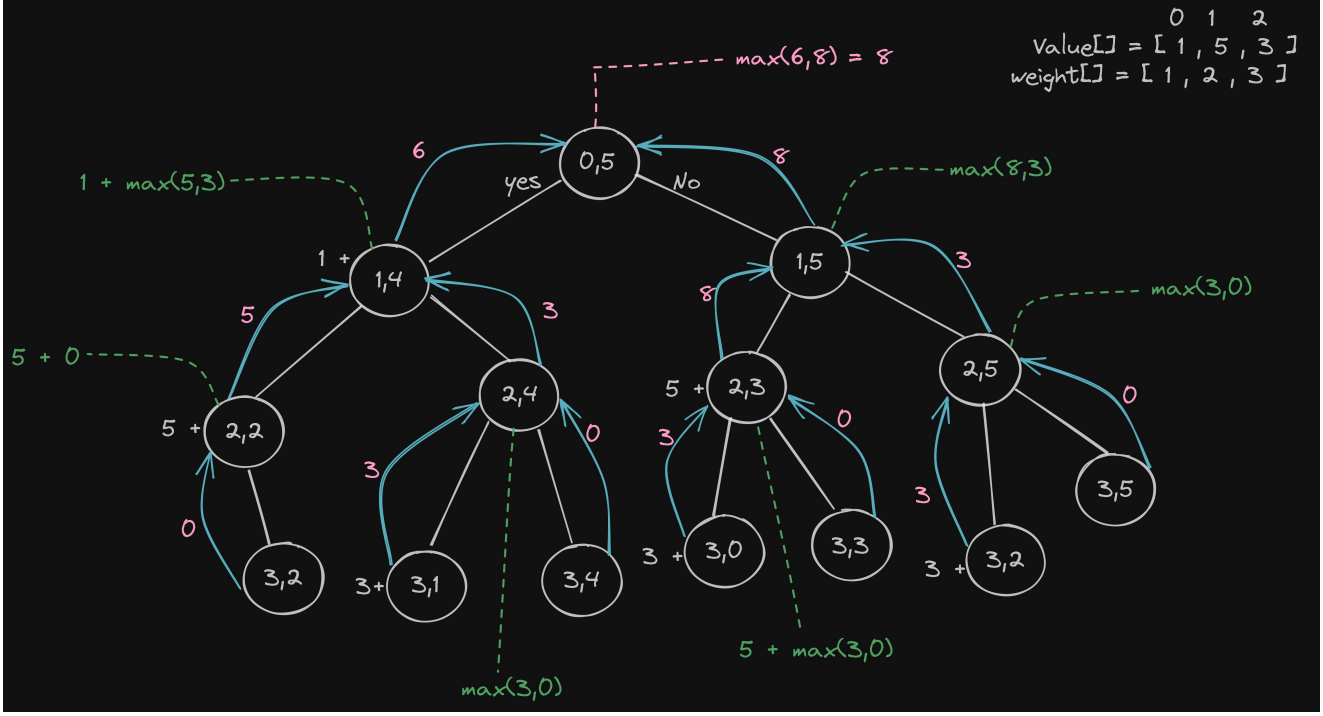
we have two option weather to take element in knapsack or not




```
          0  1  2
Value[] = [1, 5, 3]
weight[] = [1, 2, 3]
```

* n = 0
* capacity = 5

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



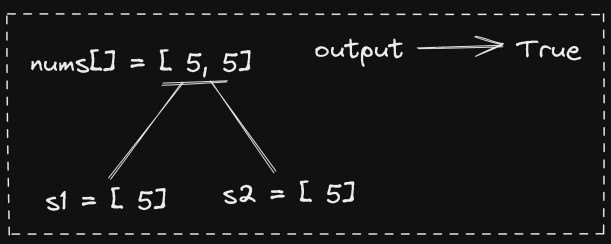

```
          0  1  2
Value[] = [1, 5, 3]
weight[] = [1, 2, 3]
```

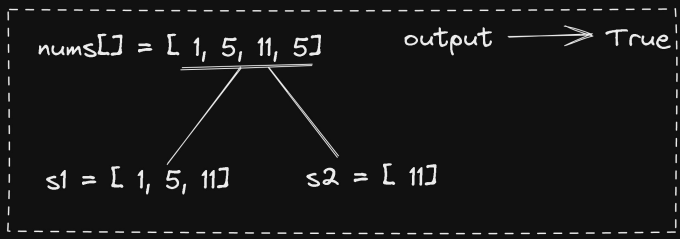
Partition Equal Subset Sum !-

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 = sum(nums);



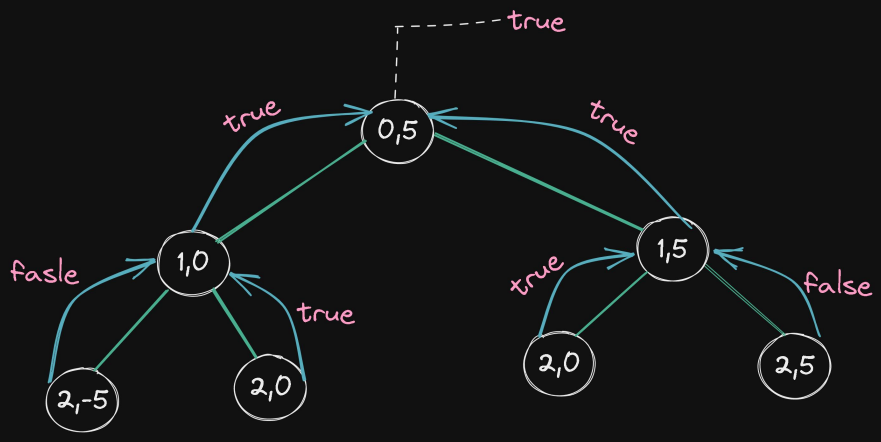


sum(s1) + sum(s2) = sum(nums)

sum(s1) + sum(s1) = sum = (nums)

2sum(s1) = sum(nums)
sum(s1) = sum(nums)/2;

nums[] = [5, 5]
0 1



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 , thats 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

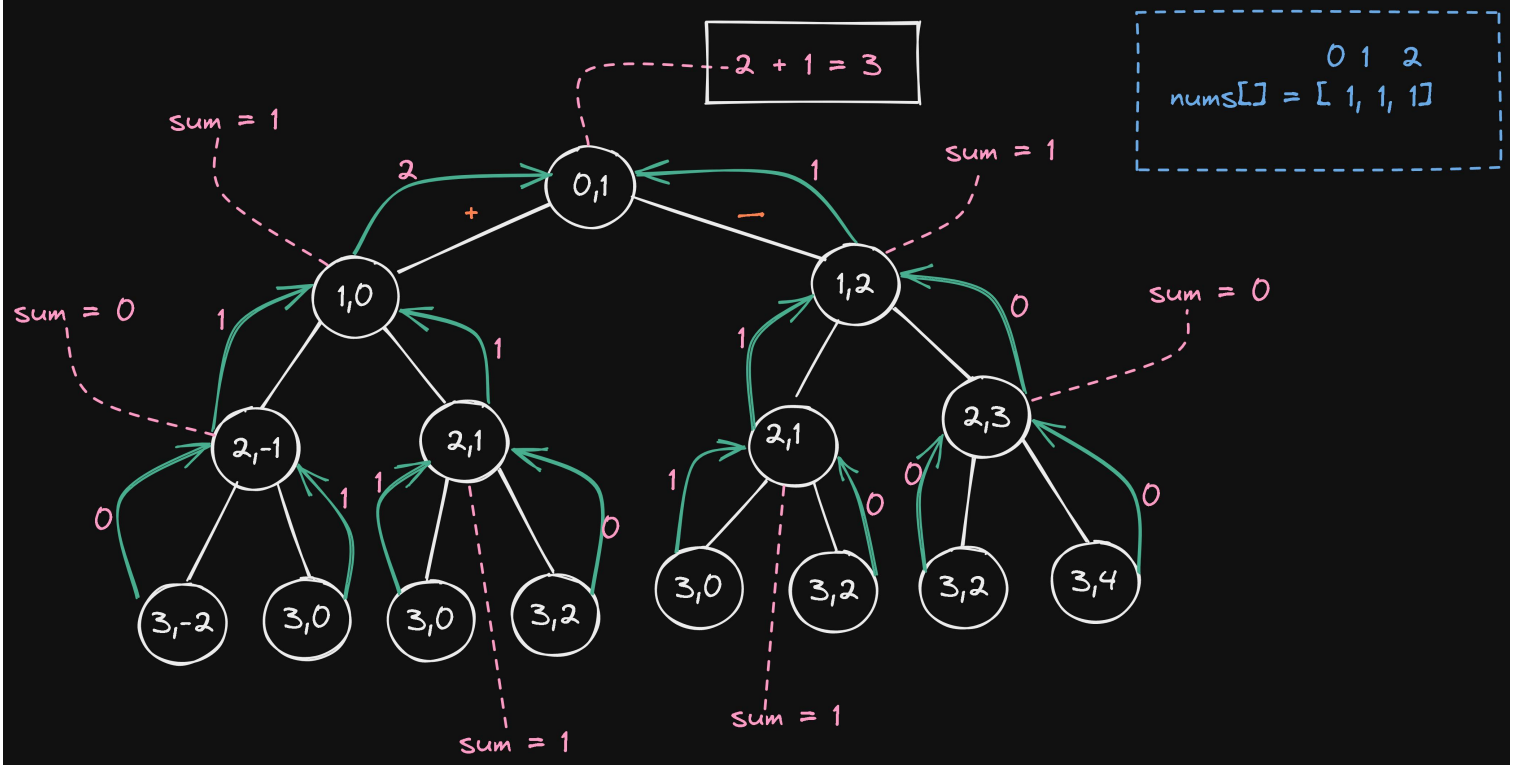
nums[] = [1, 1, 1]

target = 1

-1 + 1 + 1 = 1
1 - 1 + 1 = 1
1 + 1 - 1 = 1

→ Output: 3

0 1 2
nums[] = [1, 1, 1]



Left branch = n+1 , target - nums[n]

Right branch = n+1, target - (-nums[n])

n+1 , target + nums[n]

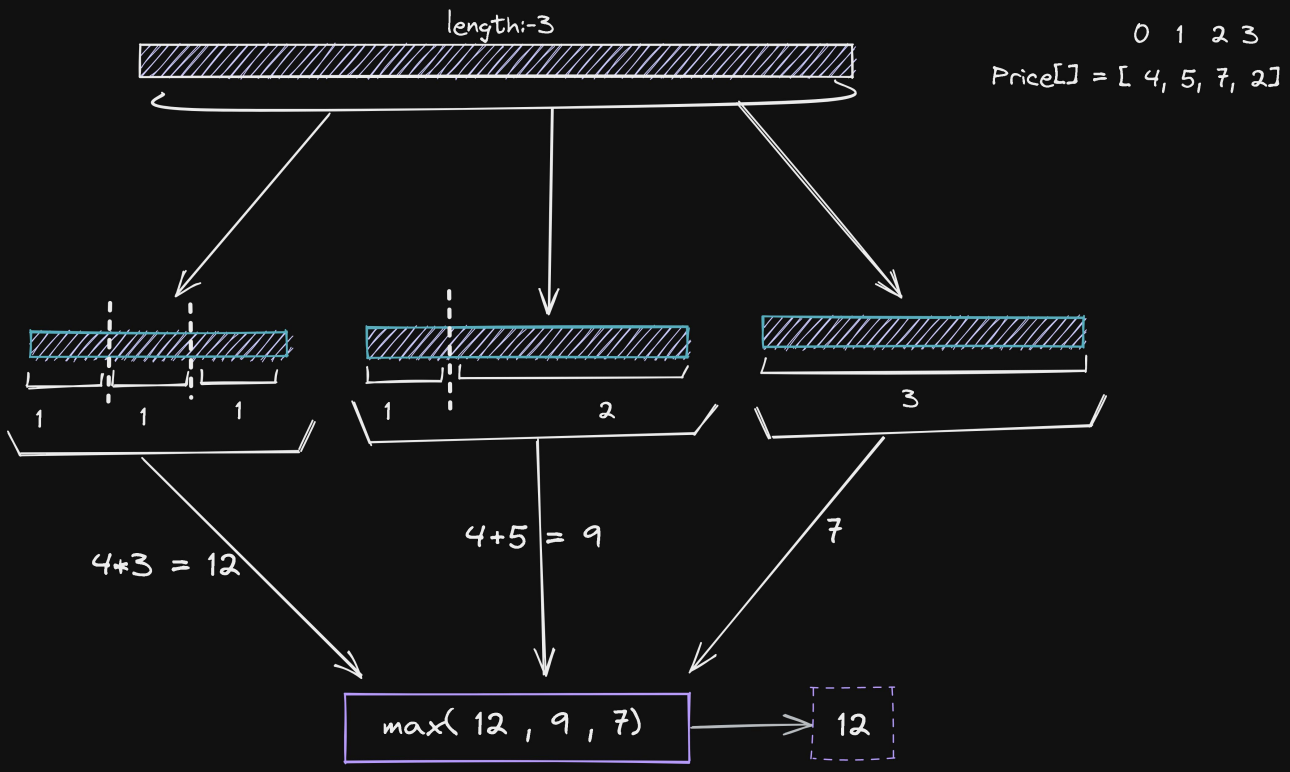
Un-Bounded KnapSack

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:-

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

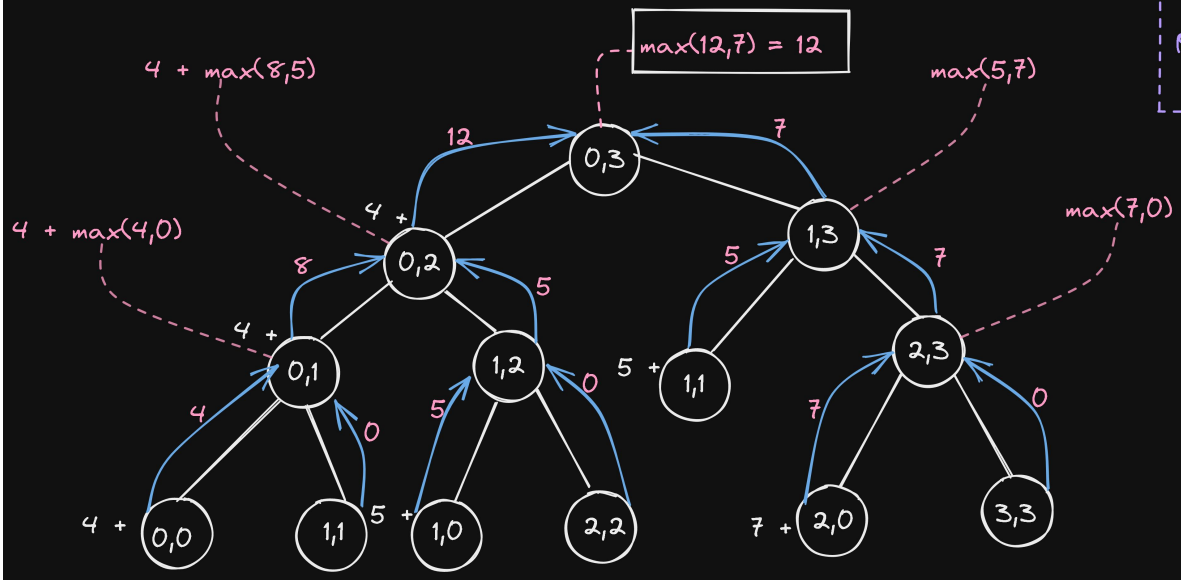
Output ⟶ 12



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

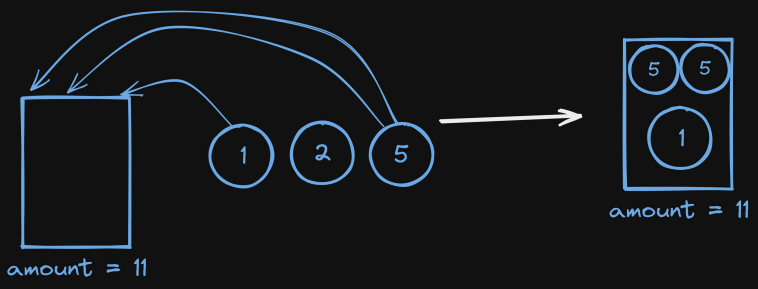


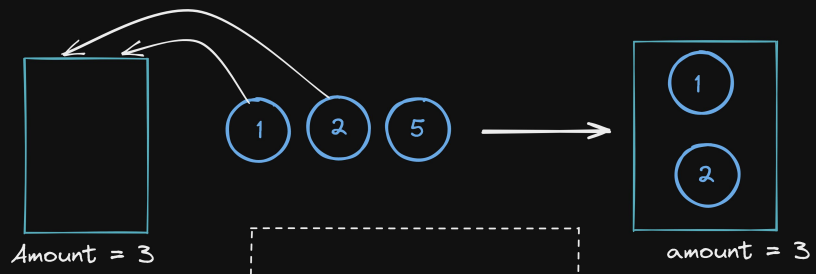
Coin Change:-

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.

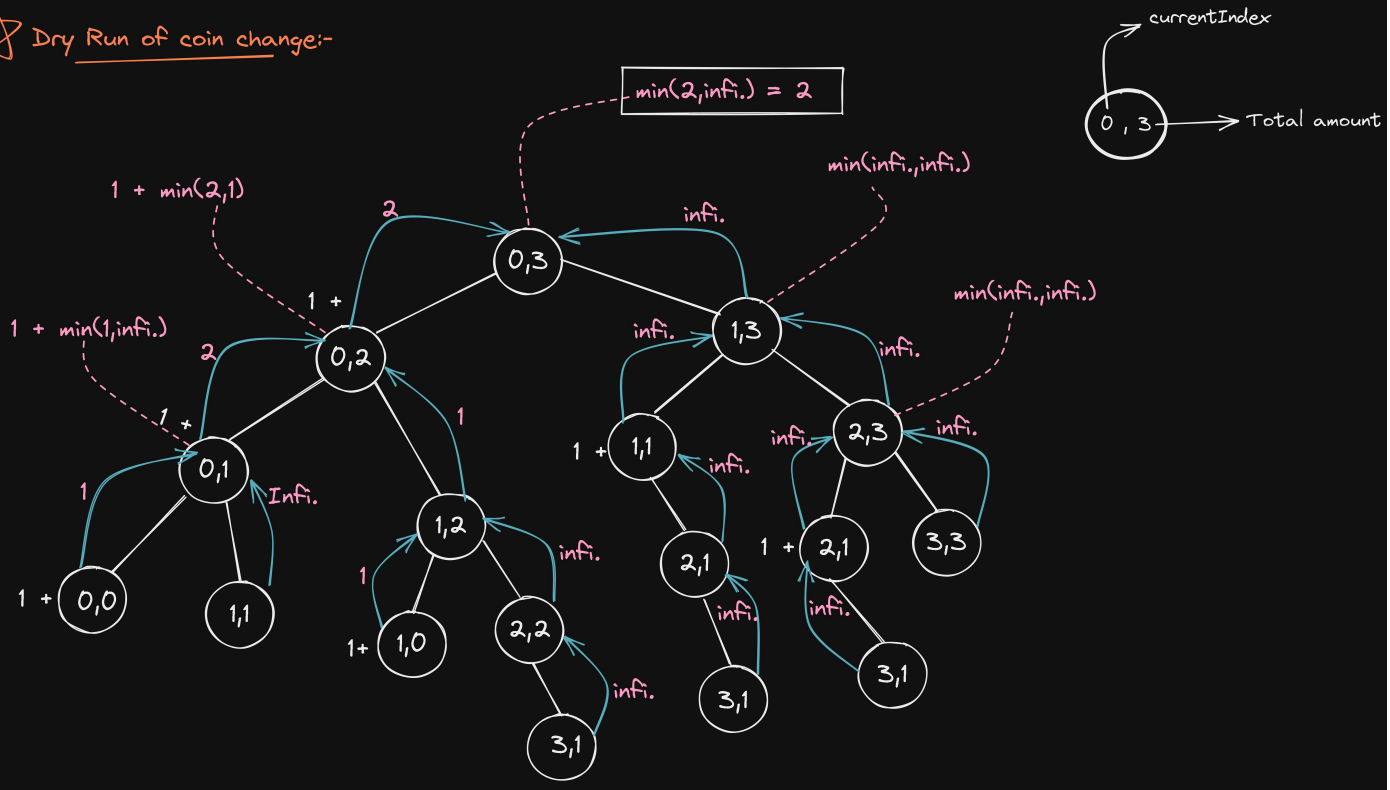
You may assume that you have an infinite number of each kind of coin.





coins = 1 2 5
Index's = 0 1 2

Dry Run of coin change:-



Using Recursion:-

Time Complexity:- O(2^n)
Space complexity:- O(amount)

Base Case:-

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

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

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

Using DP:-

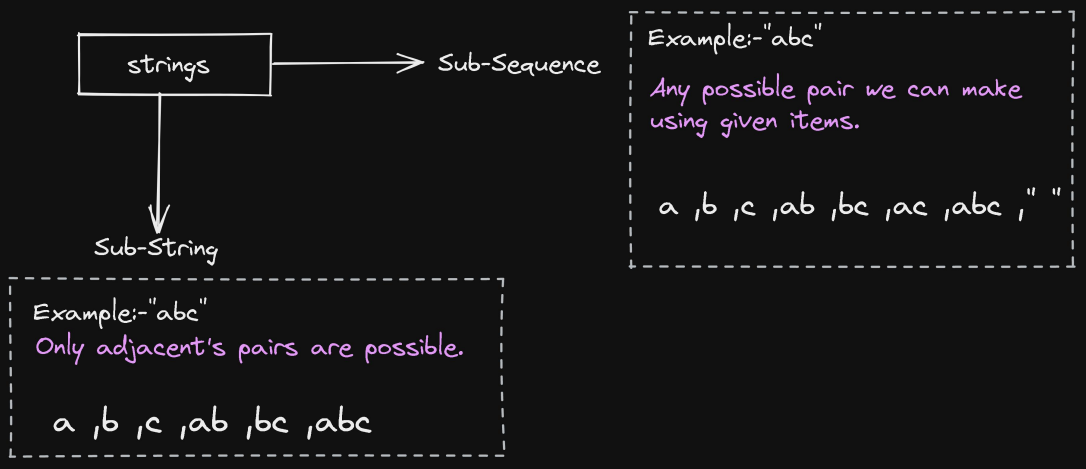
Time Complexity:- O(n*k)
Space complexity:- O(n*k)

n = coins.lenth
k = amount

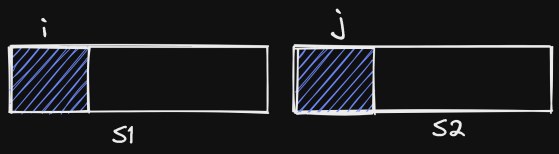
Longest common Subsequence:-

S1 = ""abcde"" S2 = "ce"

Output: 2
Explanation: The longest common subsequence is "ace" and its length is 3.



S1 = "acde" S2 = "ce"



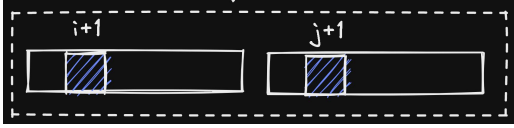
There are only Two possiblity:-

S1[i] = S2[j]

S1[i] != S2[j]

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

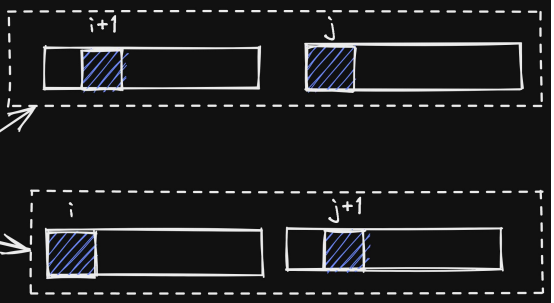
(1 + S1[i+1],S2[j+1])



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

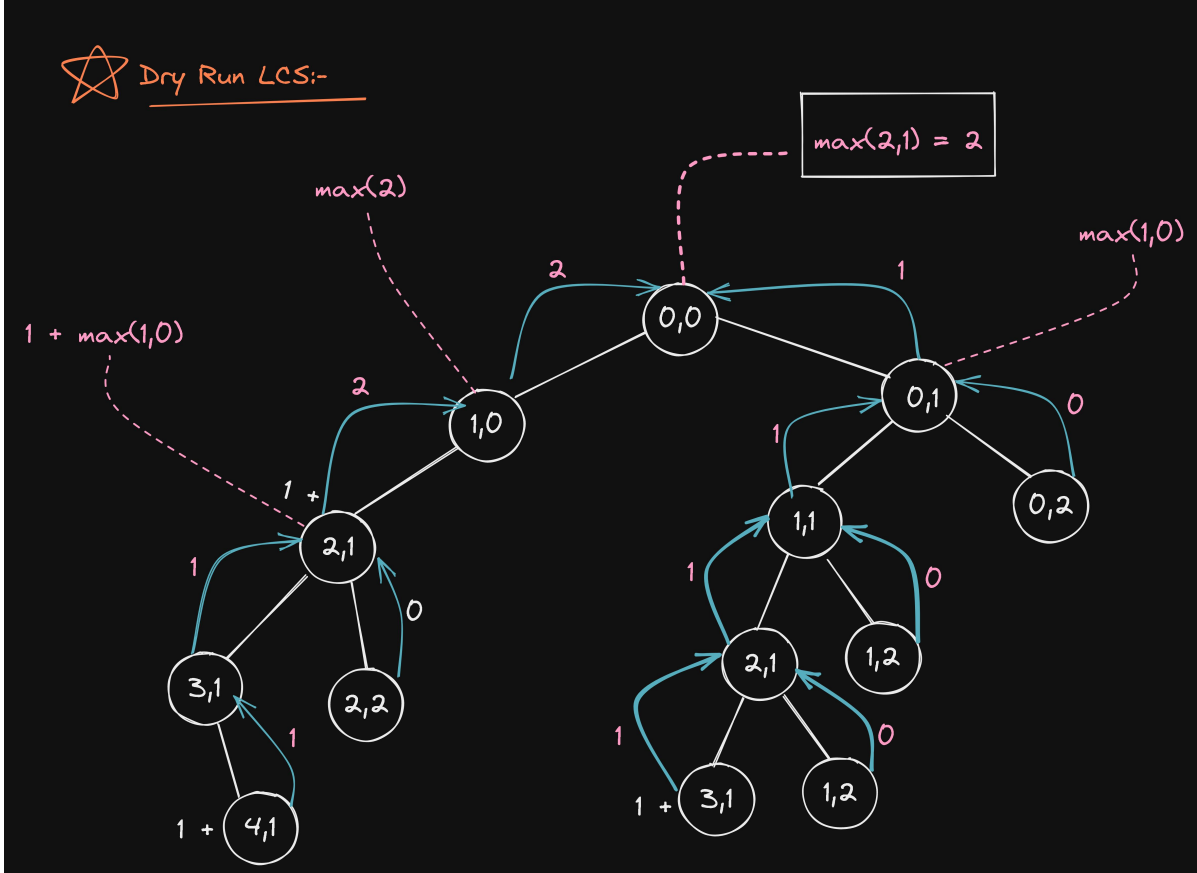
(S1[i+1], S2[j])

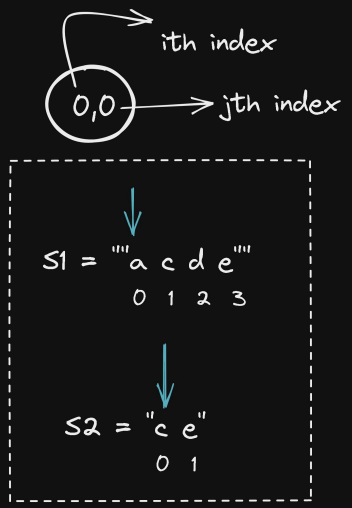
(S1[i], S2[j+1])



Longest common Subsequence:-

Dry Run LCS:-





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;

Time Complexity = O(2^n)

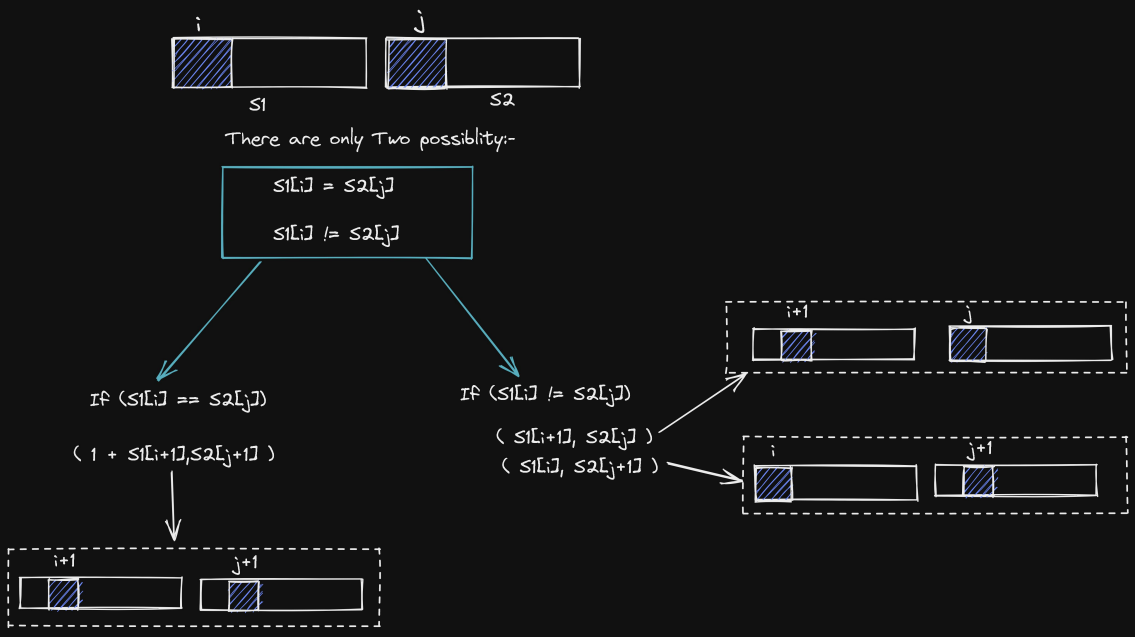
516. Longest Palindromic Subsequence

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.

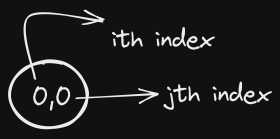
Approach:-
->create another string s2, which is reverse of s string.
->Apply the same approach of LCS(longest common subsequence).

s1 = "cbbd" ⟶ original string.
s2 = "dbbc" ⟶ reverse of s1.



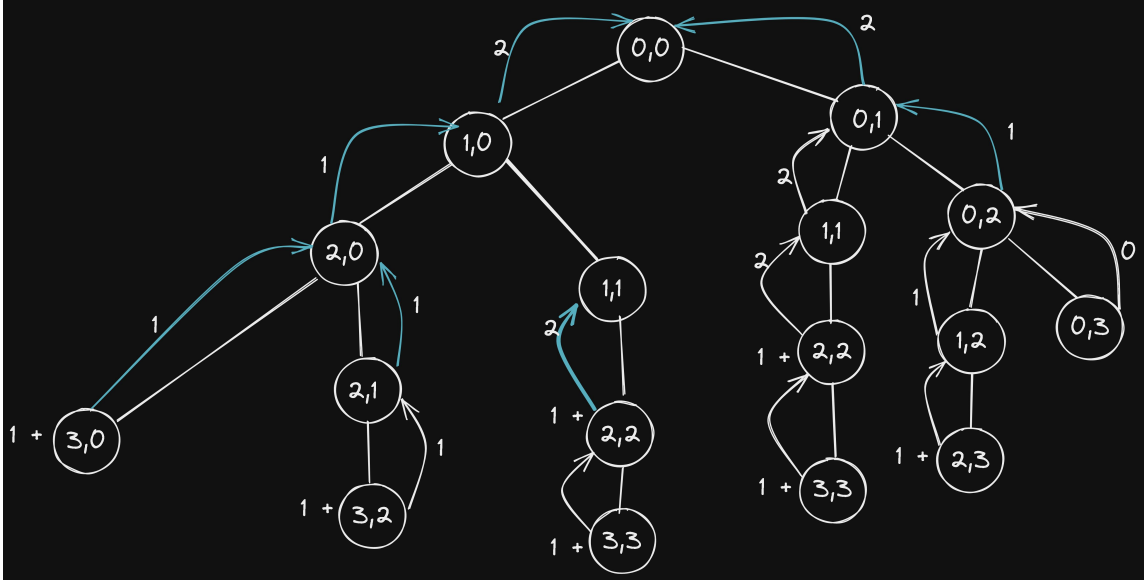
Dry Run LPS:-

```
if i == s.length || j == s.length
    return 0;
```



Text1 = ""c b b d""
i
0 1 2 3

Text2 = "d b b c"
j
0 1 2 3



Jump Game

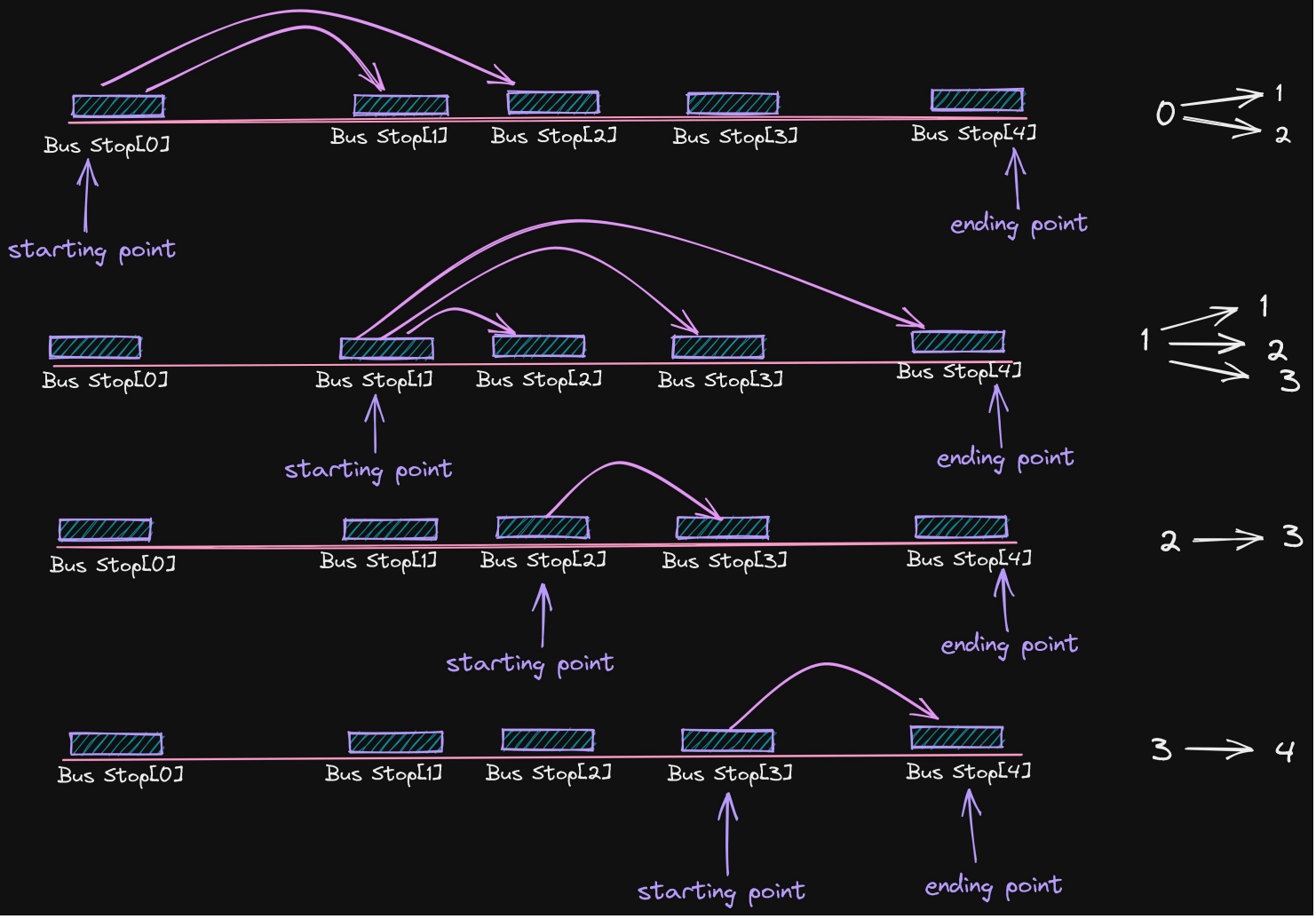
Its Optimal Approach →	Greedy
Native Approach →	Dynamic Programming

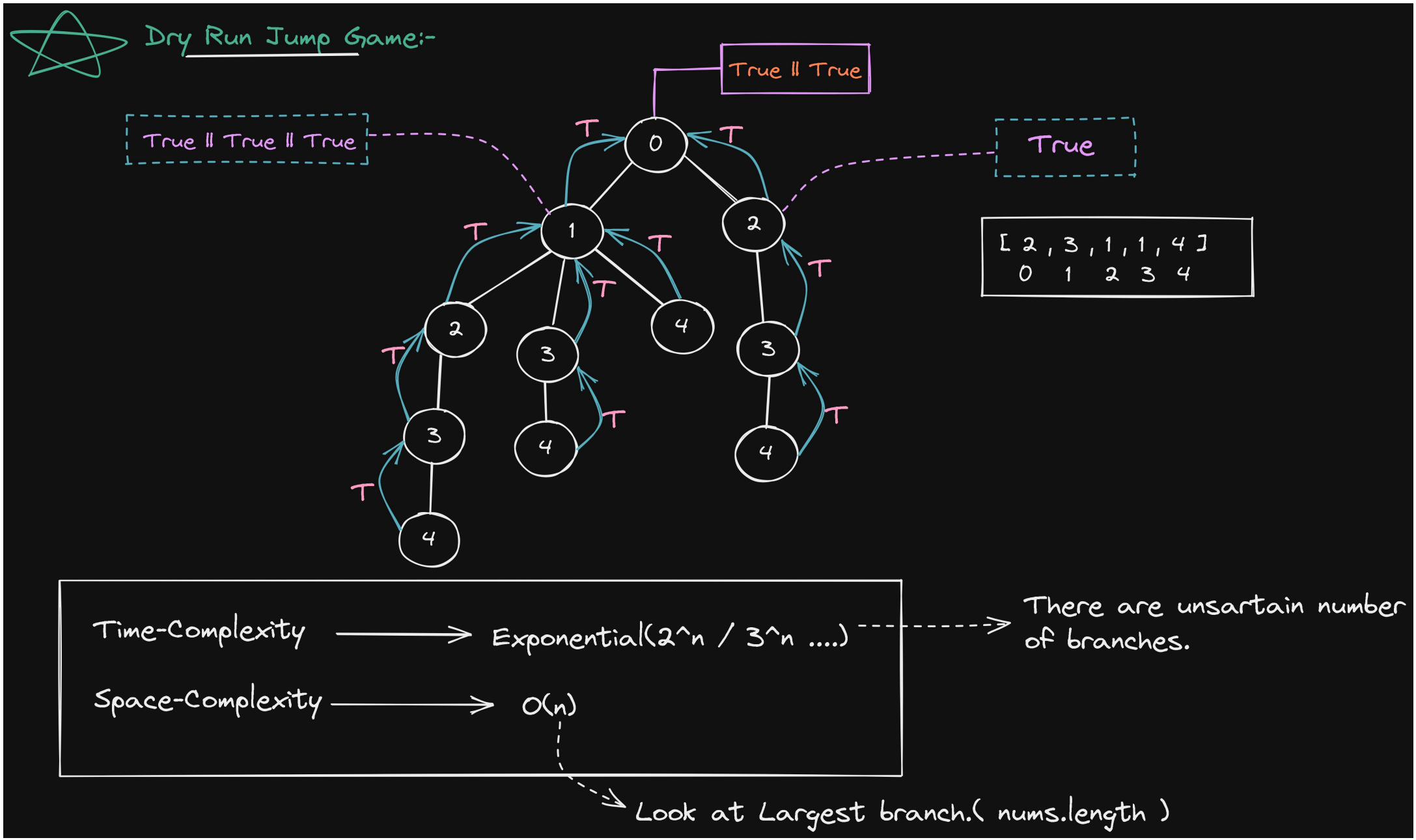
In this Question we have to reach to the last Index.
->Reach at last Index, Return "true".
->If not reach at last Index, Return "false".

```
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
```

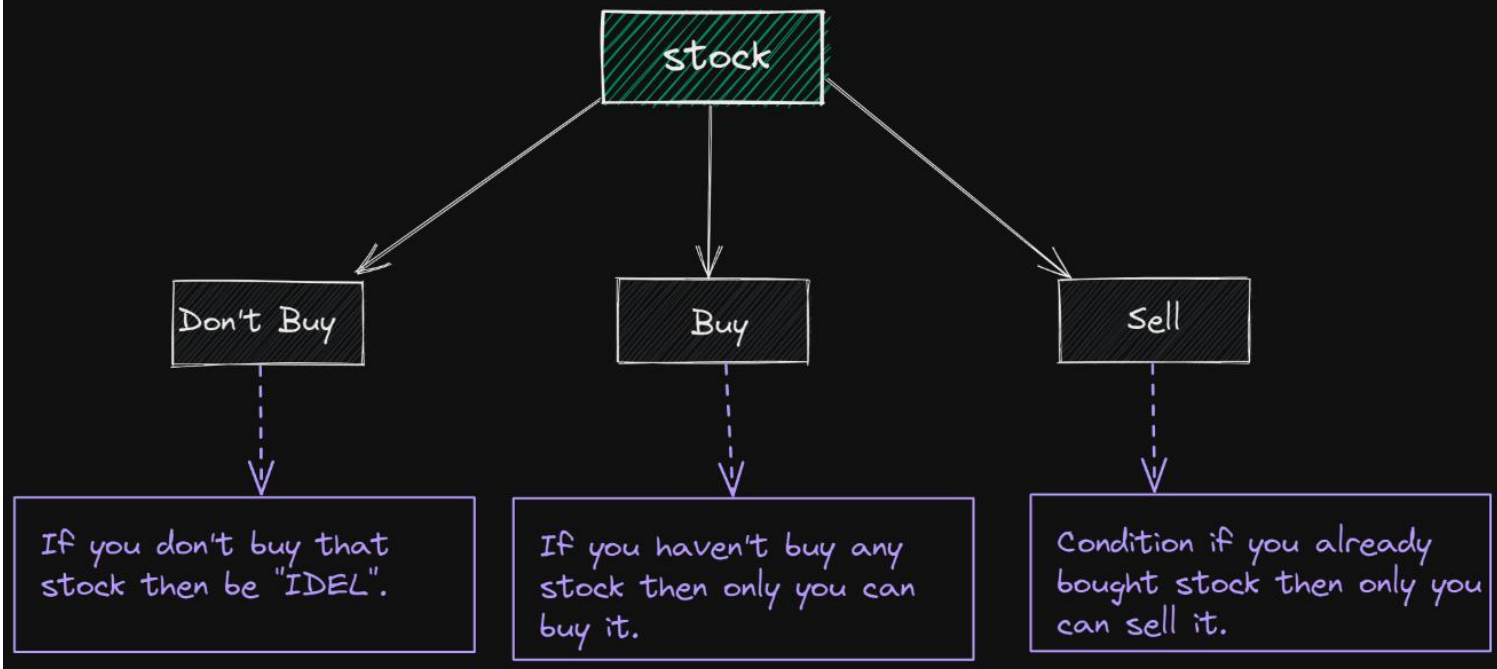



Dry Run Jump Game:-
True || True
True || True || True
True
[2 , 3 , 1 , 1 , 4]
0 1 2 3 4
Time-Complexity
Exponential(2^n / 3^n ....)
There are unsartain number of branches.
Space-Complexity
O(n)
Look at Largest branch.(nums.length)

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 price[i], where ith is the day.



Note:- Only one transaction is allowed.
Buy & Sell ⟶ 1 transaction

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

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

Output(profit) : 5

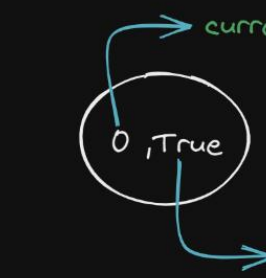
[7 , 1 , 5 , 3 , 6 , 4] ⟶ -(Buy) + (Sell) = Profit
- 1 + 6 = 5

Buy Sell

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

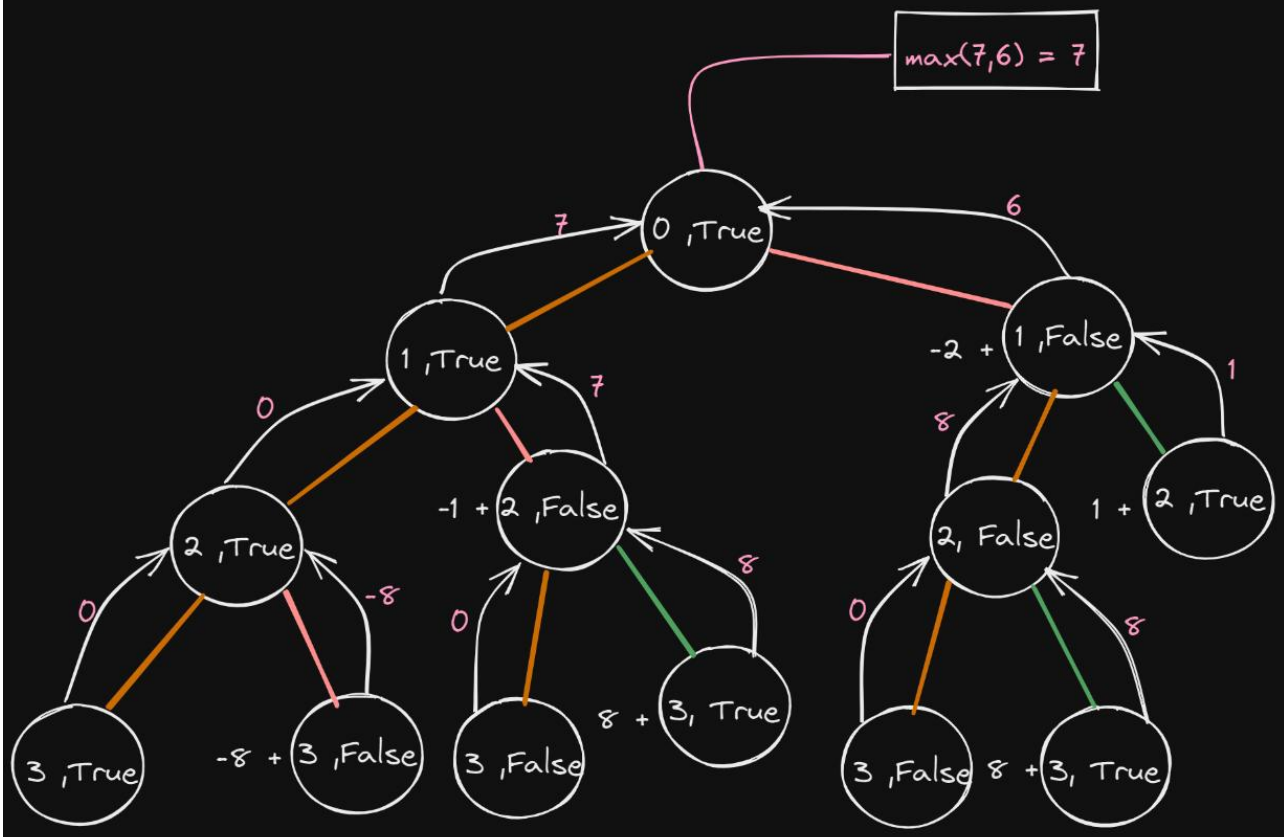
Best Time to Buy and Sell Stock 2

We can do infinite number of transaction in this question.



Ideal
Buy
Sell

```
Indexs =   0   1   2
prices = [ 2 , 1 , 8 ]
```



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

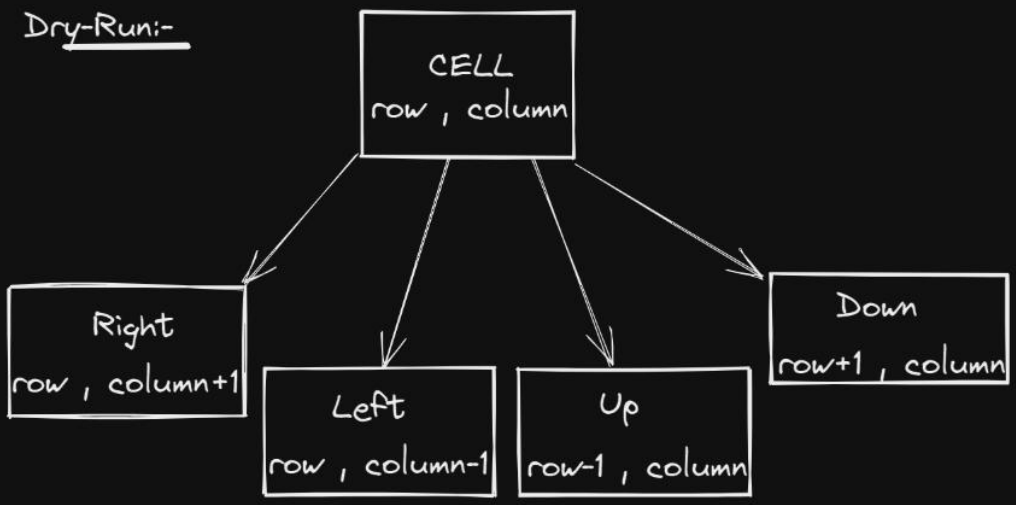
→ Given an m x n integers matrix, return the length of the longest increasing path in matrix.

→ 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:-

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:-

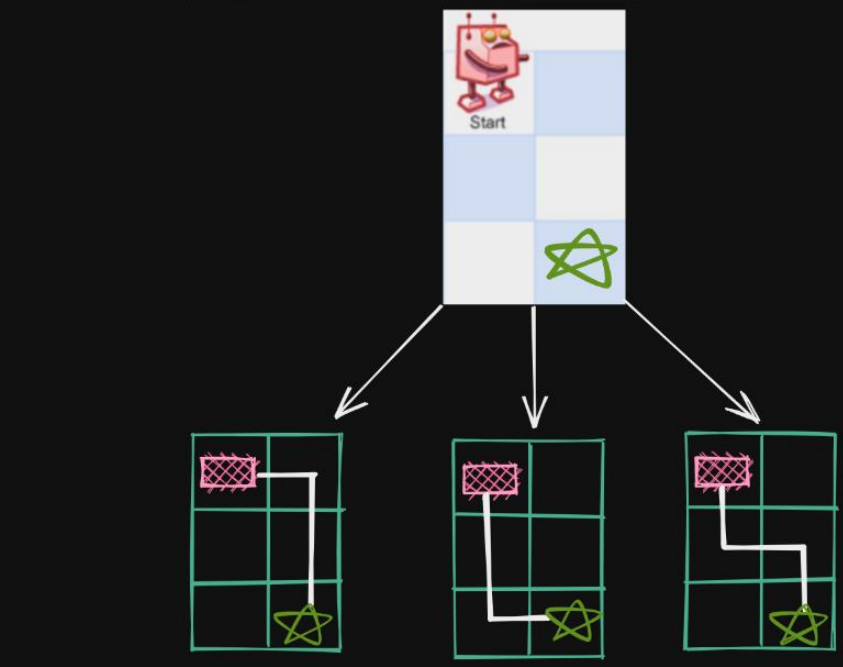
```
if(currentRow < 0 || currentCol < 0 || currentRow >= m || currentCol >= n){
    return 0;
}

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

62. Unique Paths

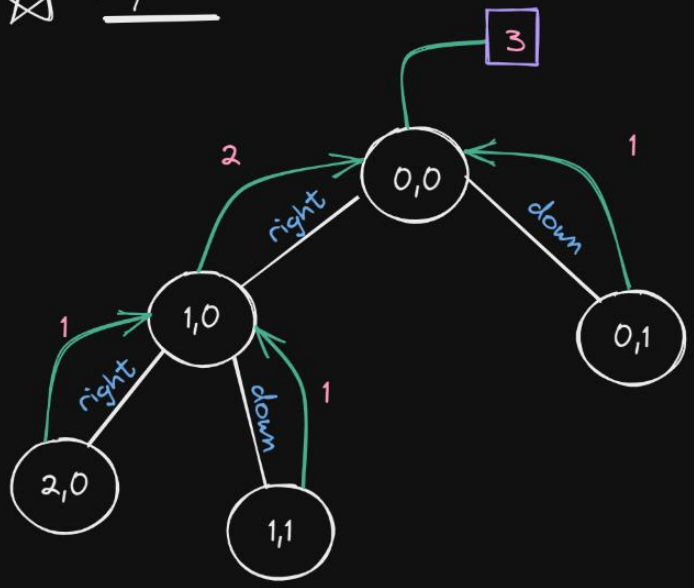
A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

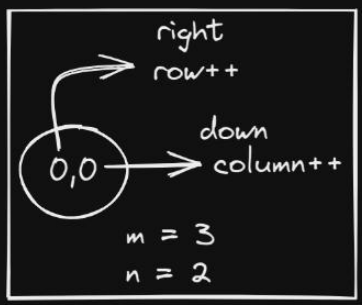
The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid

We can only move on to right or down only.



Dry-Run






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

Time-complexity → O(row*col)
Space-complexity → O(max(row,col))

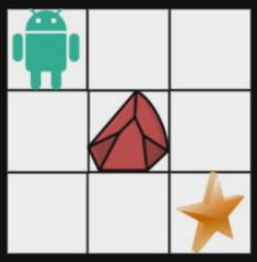
→ using DP

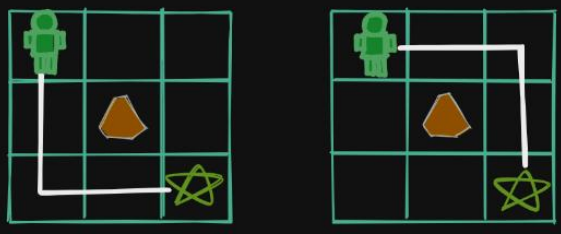
Time-complexity → O(2^(row*col))
Space-complexity → O(max(row,col))

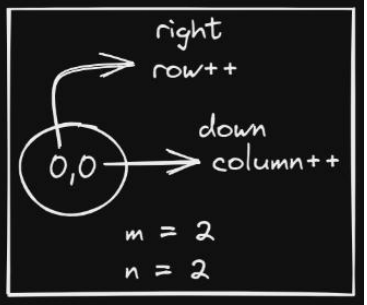
→ using Recursion

62. Unique Paths

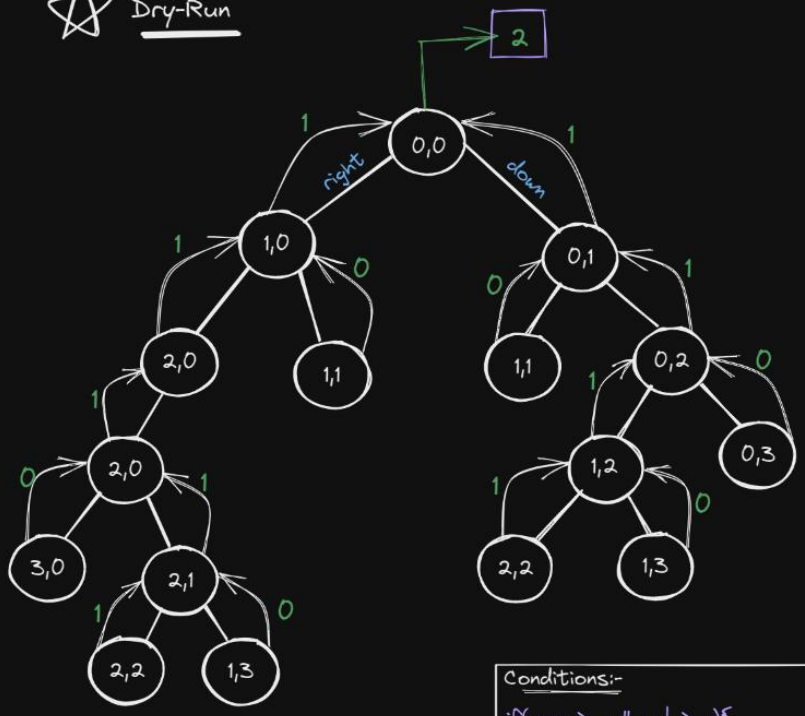
- A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).
- The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid
- we have to Reach to cornor grid at bottom-right, if there is an obstical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







Dry-Run



Using DP

Time-complexity → O(row*col)
Space-complexity → O(max(row,col))

Using Recursion

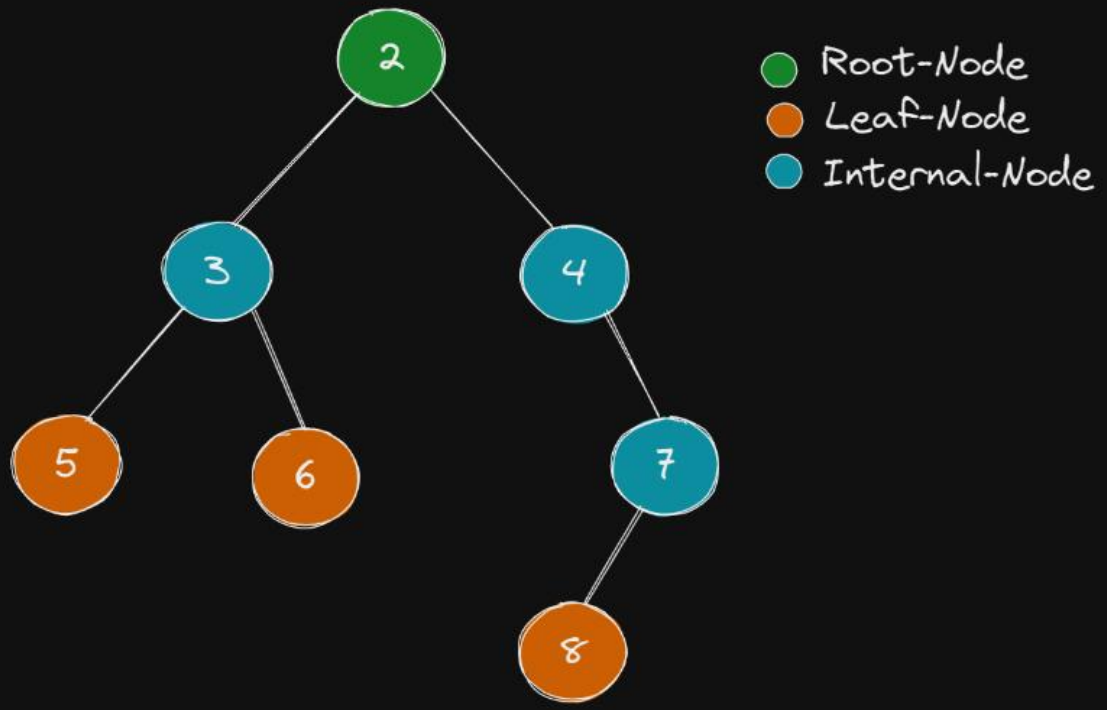
Time-complexity → O(2^(row*col))
Space-complexity → O(max(row,col))

Conditions:-

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

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

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

Binary Tree

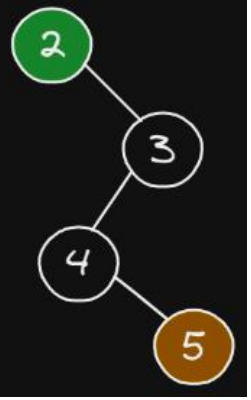
Binary tree is a data structure in which we can store data structure in herarical manner. Every Node present in Binary-Tree can Have atmost 2 childrean only.



Types of Binary-Tree:-

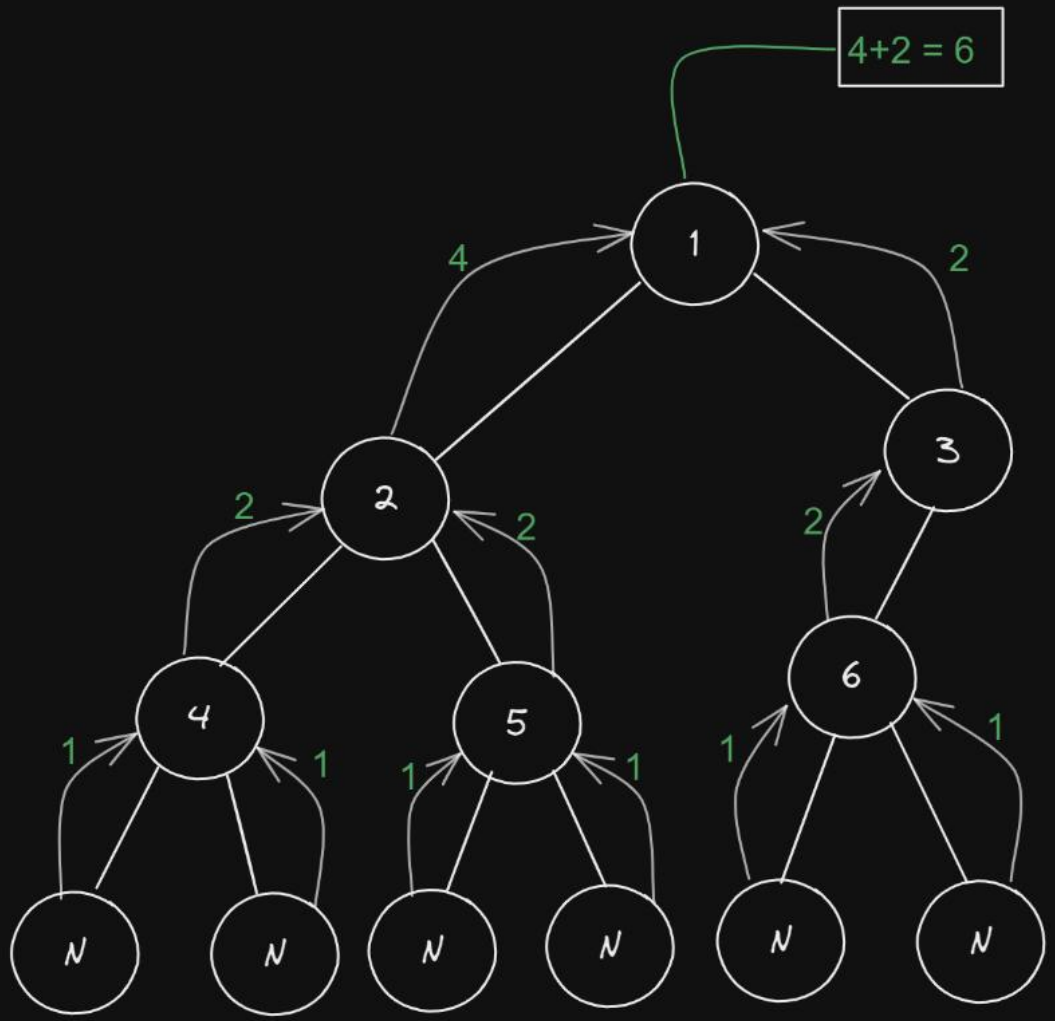
1. Pathological Tree or degenerate

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

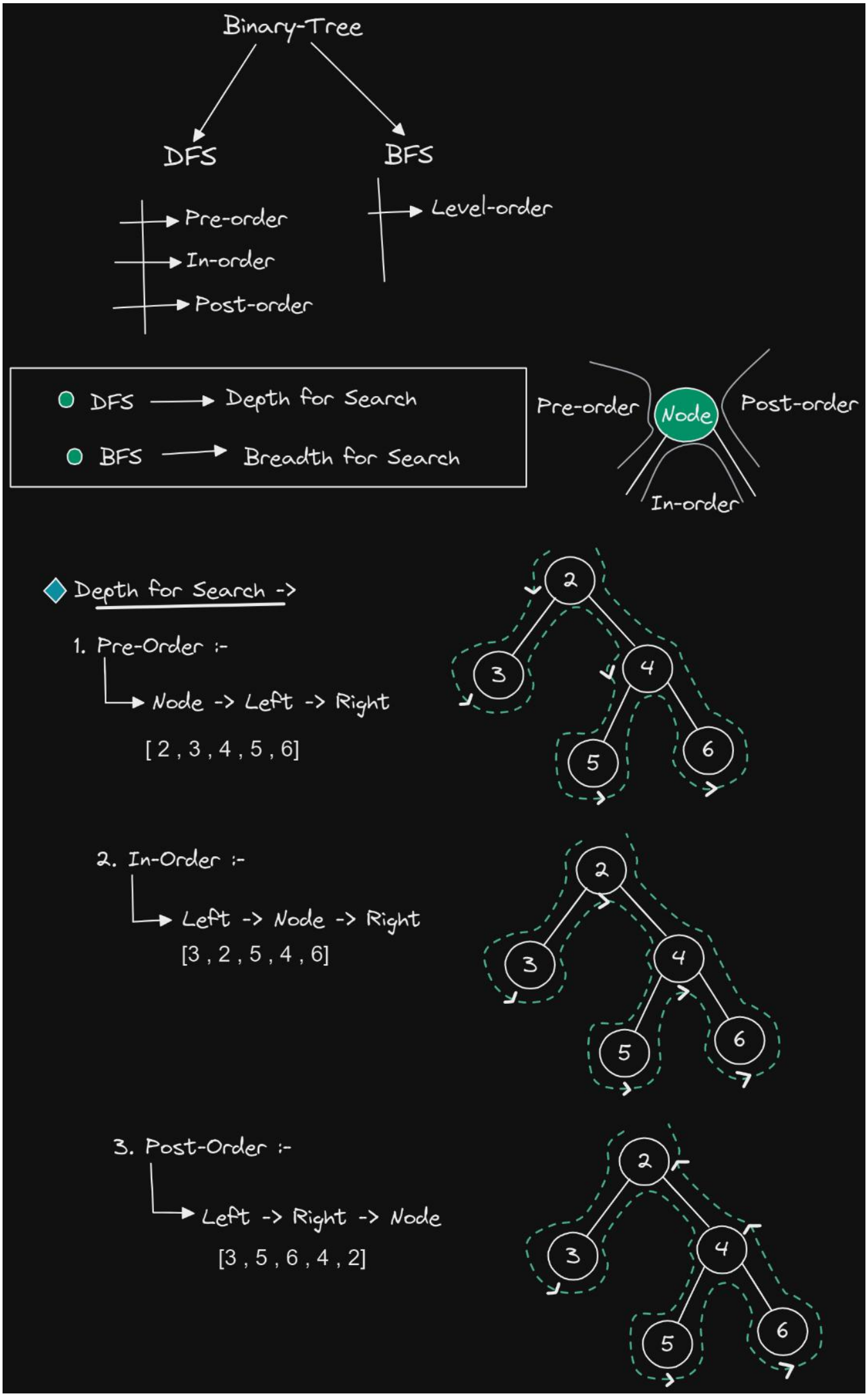


Count Complete Tree Nodes

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

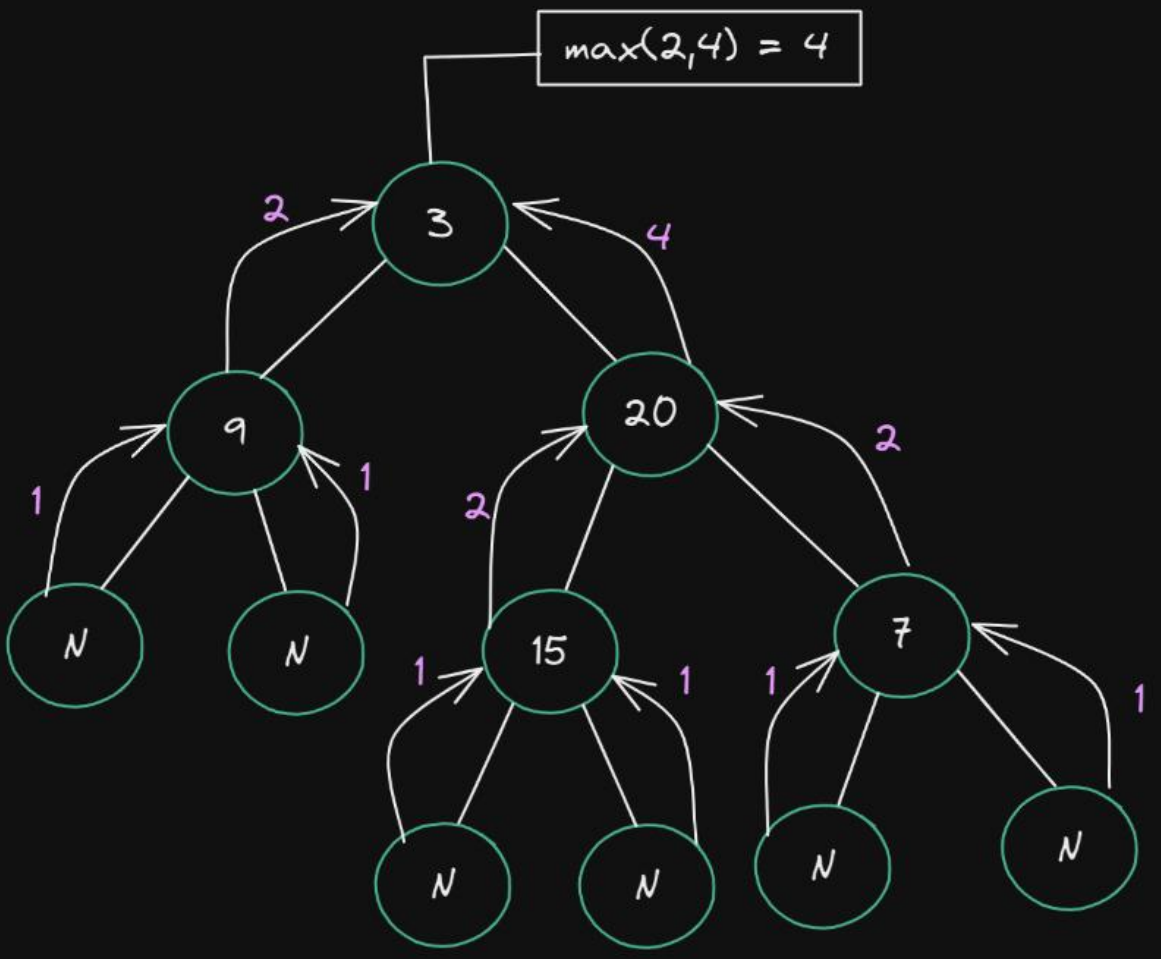


Time-complexity -> O(n)
Space-complexity ->O(max-depth)


Binary-Tree
DFS
BFS
Pre-order
In-order
Post-order
Level-order
DFS → Depth for Search
BFS → Breadth for Search
Pre-order
Node
Post-order
In-order
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 -> Number of Nodes

Time-complexity - O(N)
Space-complexity - O(Max-Depth)

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