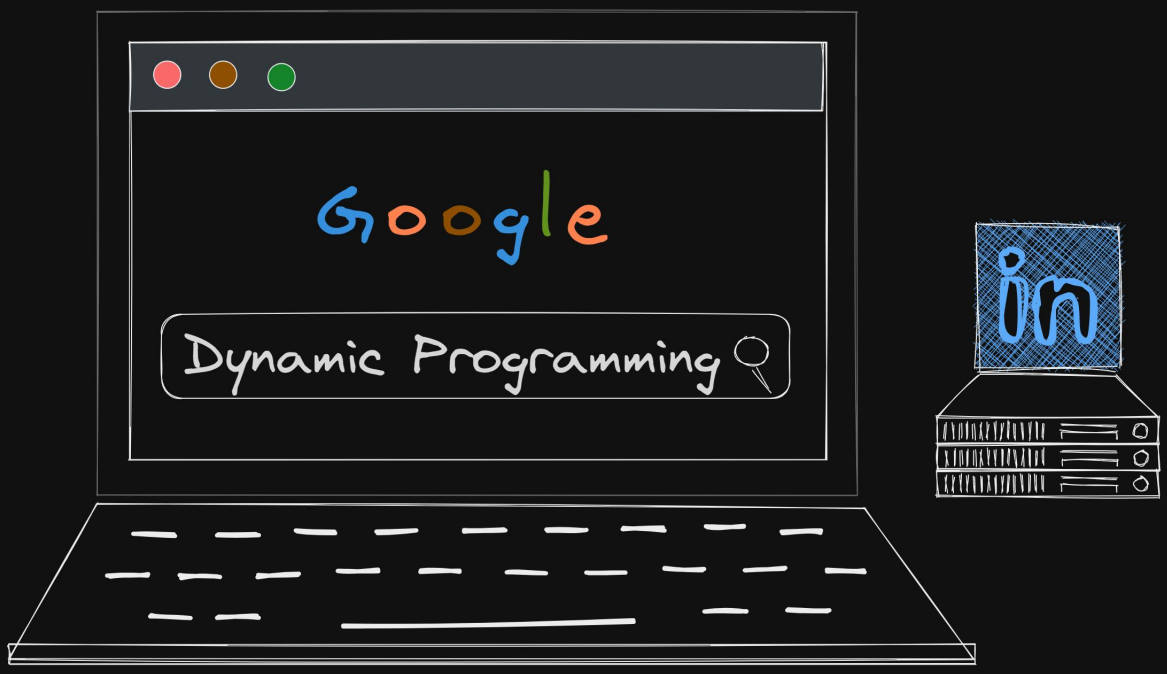
Interview prepration DP Questions



Created By - Vaibhav Chauhan

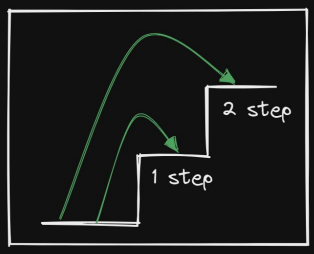
Mentor - Sahil Srivastava

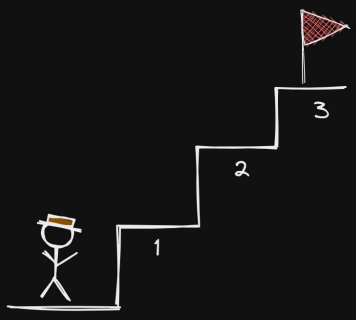
Climbing Stairs

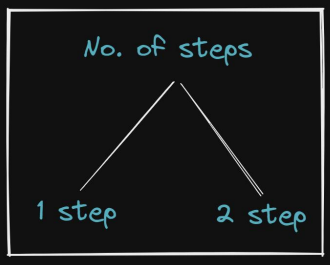
You are climbing a staircase. It takes n steps to reach the top.

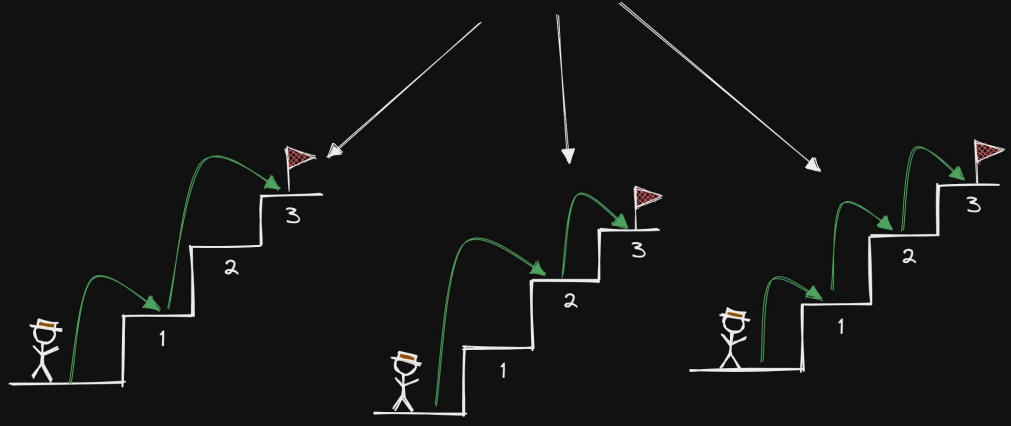
Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

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

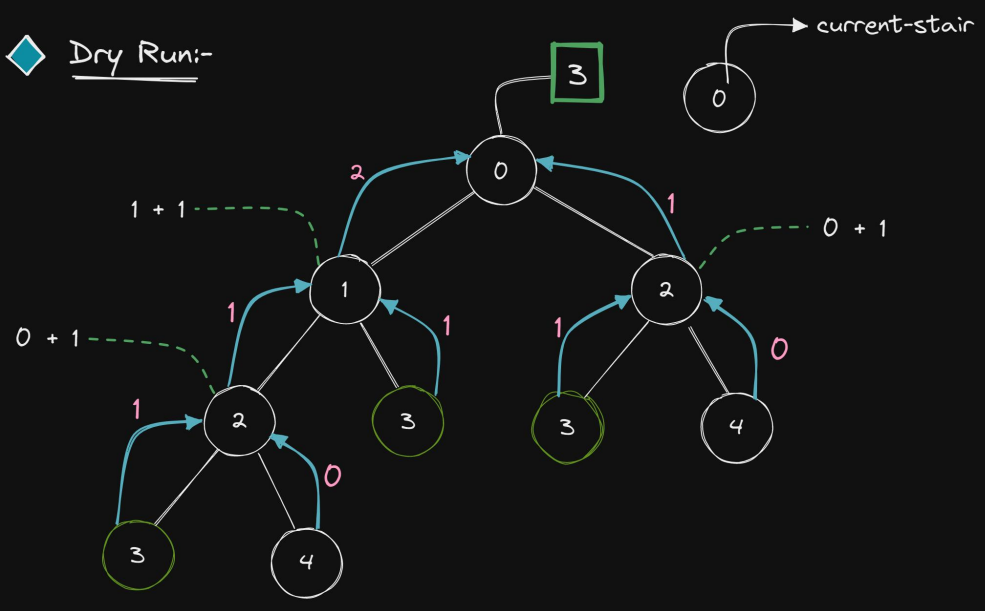








Dry Run:-



consitions:-

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

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

Recursion

Time-complexity → O(2^N)

Space-complexity → O(N)

Dynamic-Programming

Time-complexity → O(N)

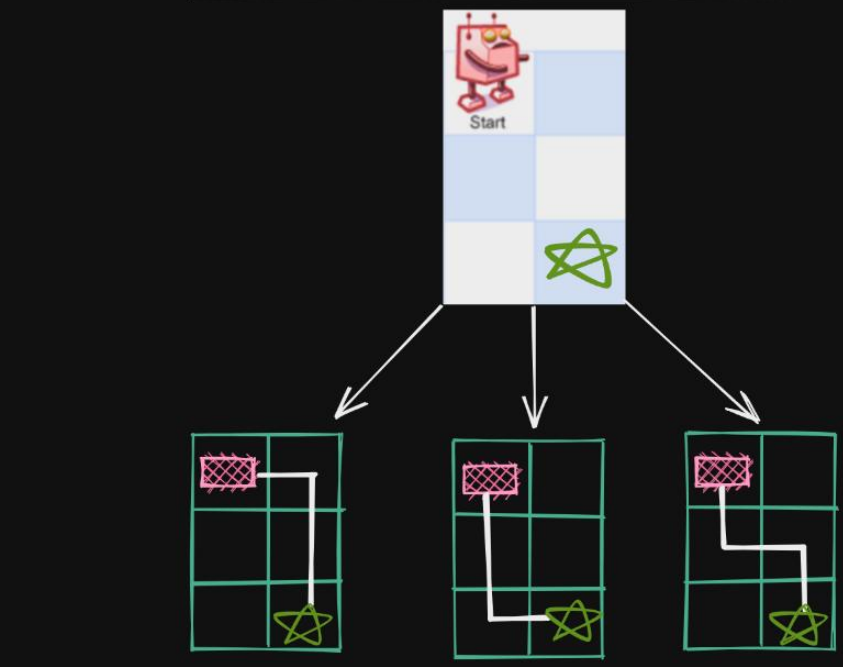
Space-complexity → O(N)

62. Unique Paths

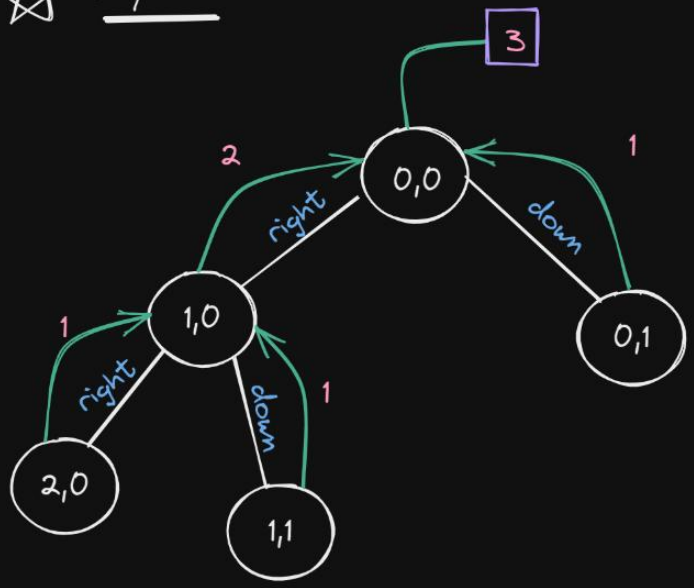
A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

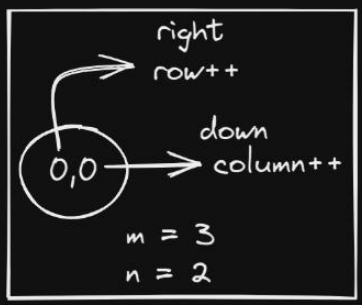
The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid

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))

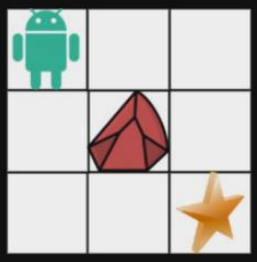
→ using DP

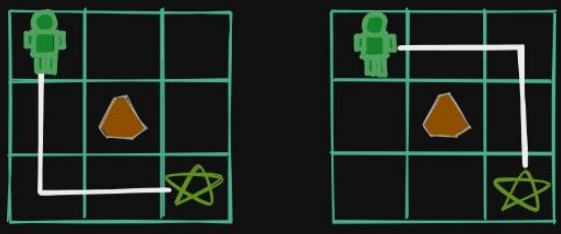
Time-complexity → O(2^(row*col))
Space-complexity → O(max(row,col))

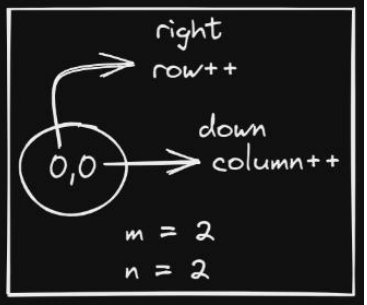
→ using Recursion

62. Unique Paths

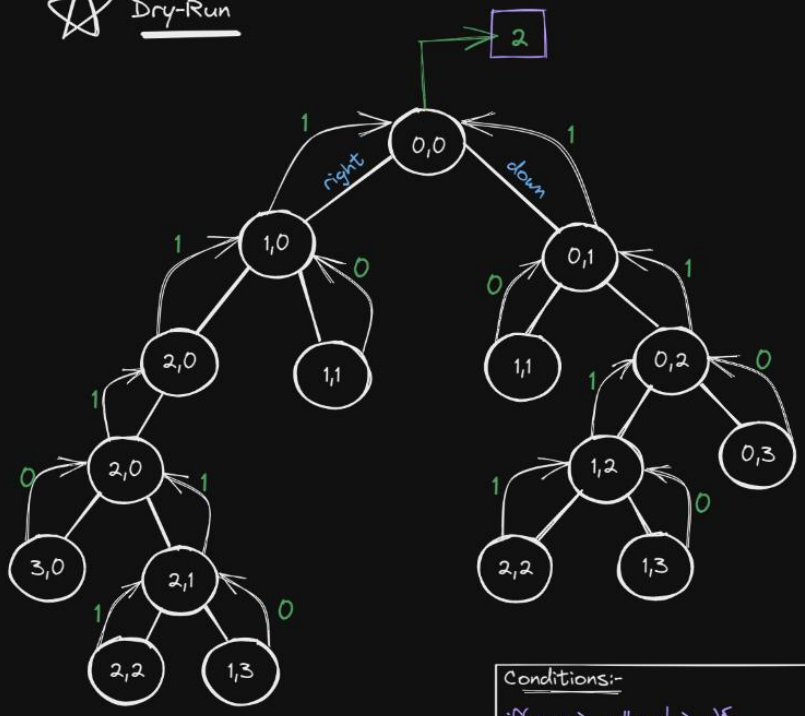
- A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).
- The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid
- we have to Reach to cornor grid at bottom-right, if there is an obstical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

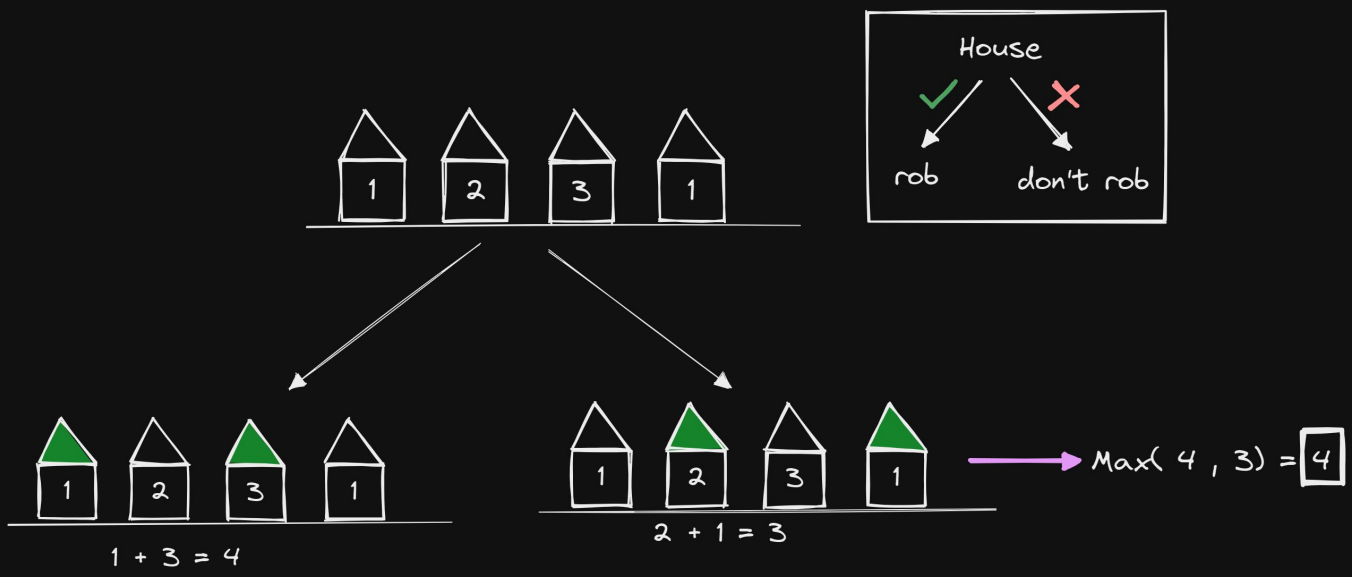
```
if(row > m || col > n){
        return 0;
    }

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

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

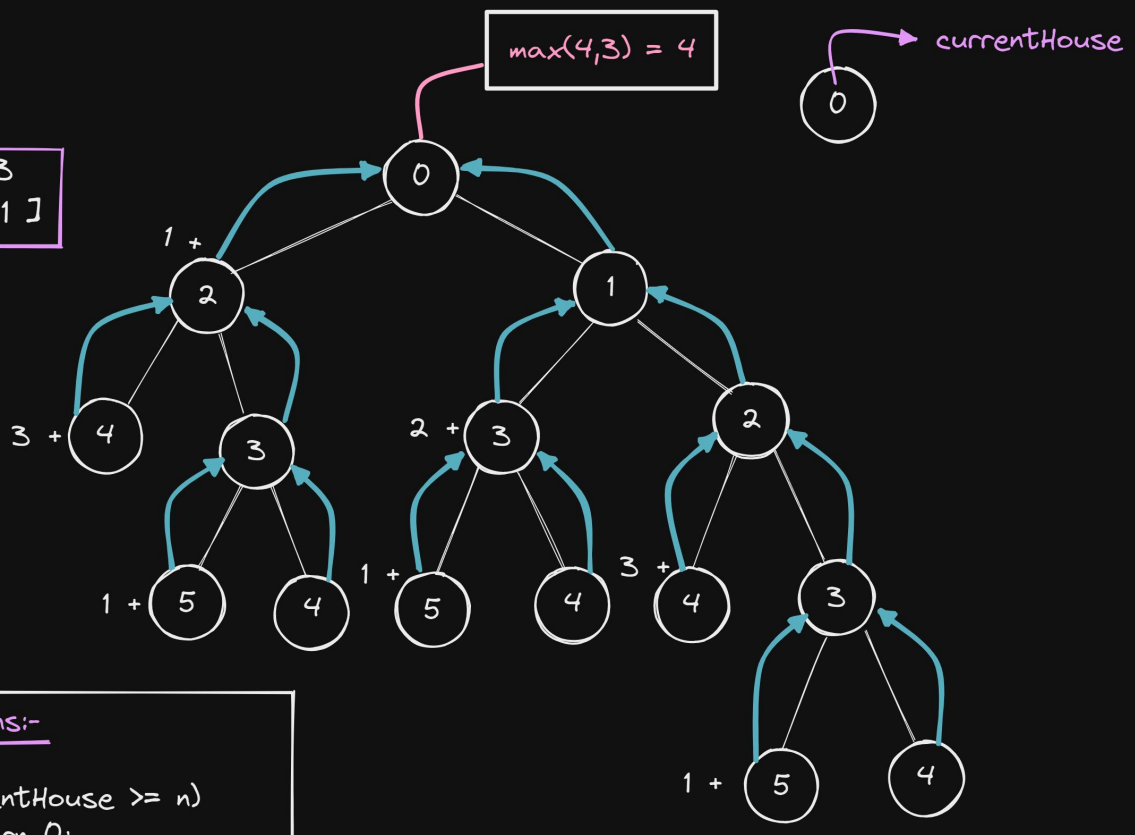
House Robber

You are a professional robber planning to rob houses along a street.
if two adjacent houses were broken into on the same night.



Dry-Run

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



conditions:-

```
if(currentHouse >= n)
    return 0;
```

Recursion

Time-complexity	O(2^N)
Space-complexity	O(N)

Dynamic-Programming

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

Jump Game

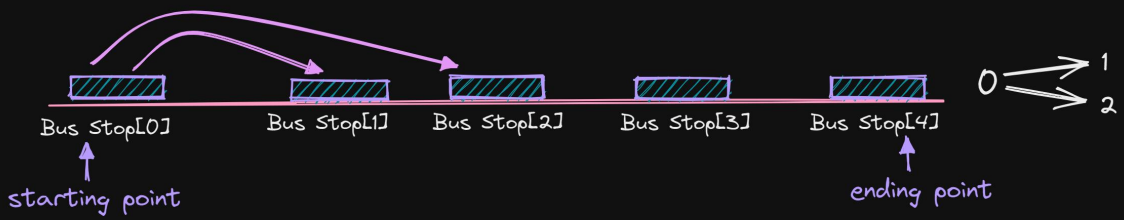
In this Question we have to reach to the last Index.

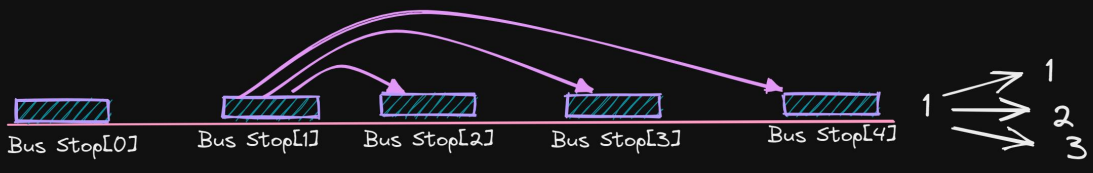
→ Reach at last Index, Return "true".
→ If not reach at last Index, Return "false".

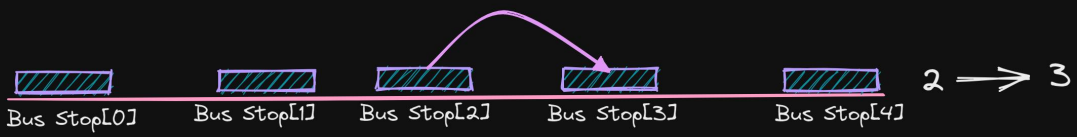
Input: nums = [2 , 3 , 1 , 1 , 4]
Output: True

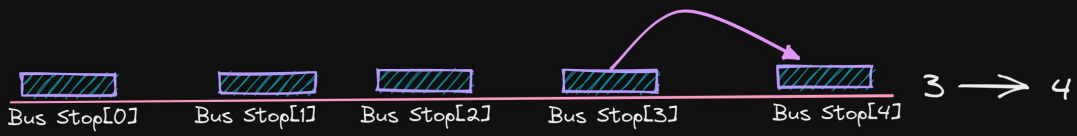
[	2	,	3	,	1	,	1	,	4	]
	0		1		2		3		4	

* Take a Example of Bus :-
Like, from Bus-Stop[0] we can go to bus-stop[1] or bus-stop[2]

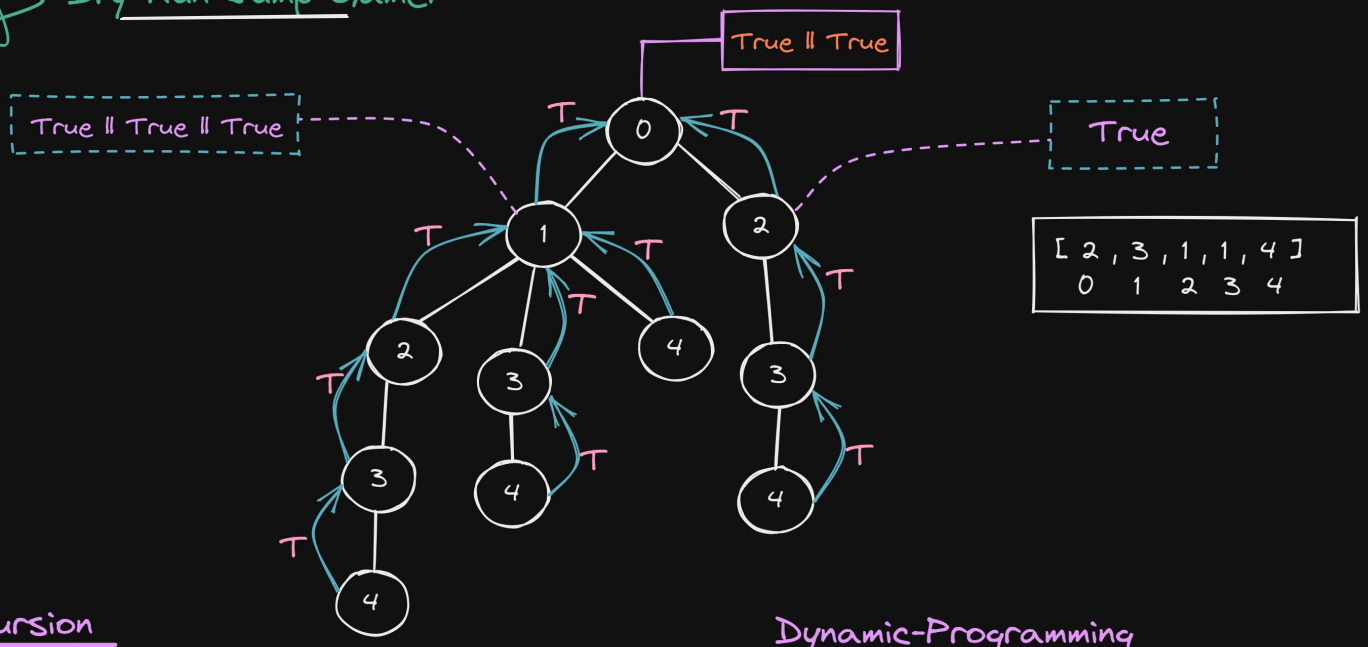








Dry Run Jump Game:-



Recursion

Time-Complexity	→	Exponential(2^n / 3^n ....)
Space-Complexity	→	O(n)

Dynamic-Programming

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

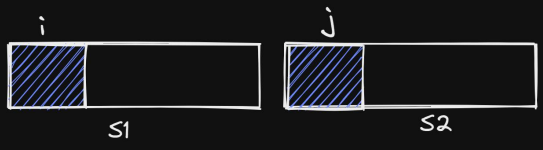
Longest common Subsequence (LCS):-

Sub-Sequence

Example:-"abc"

Any possible pair we can make using given items.

a ,b ,c ,ab ,bc ,ac ,abc ," "



There are only Two possiblity:-

S1[i] = S2[j]

S1[i] != S2[j]

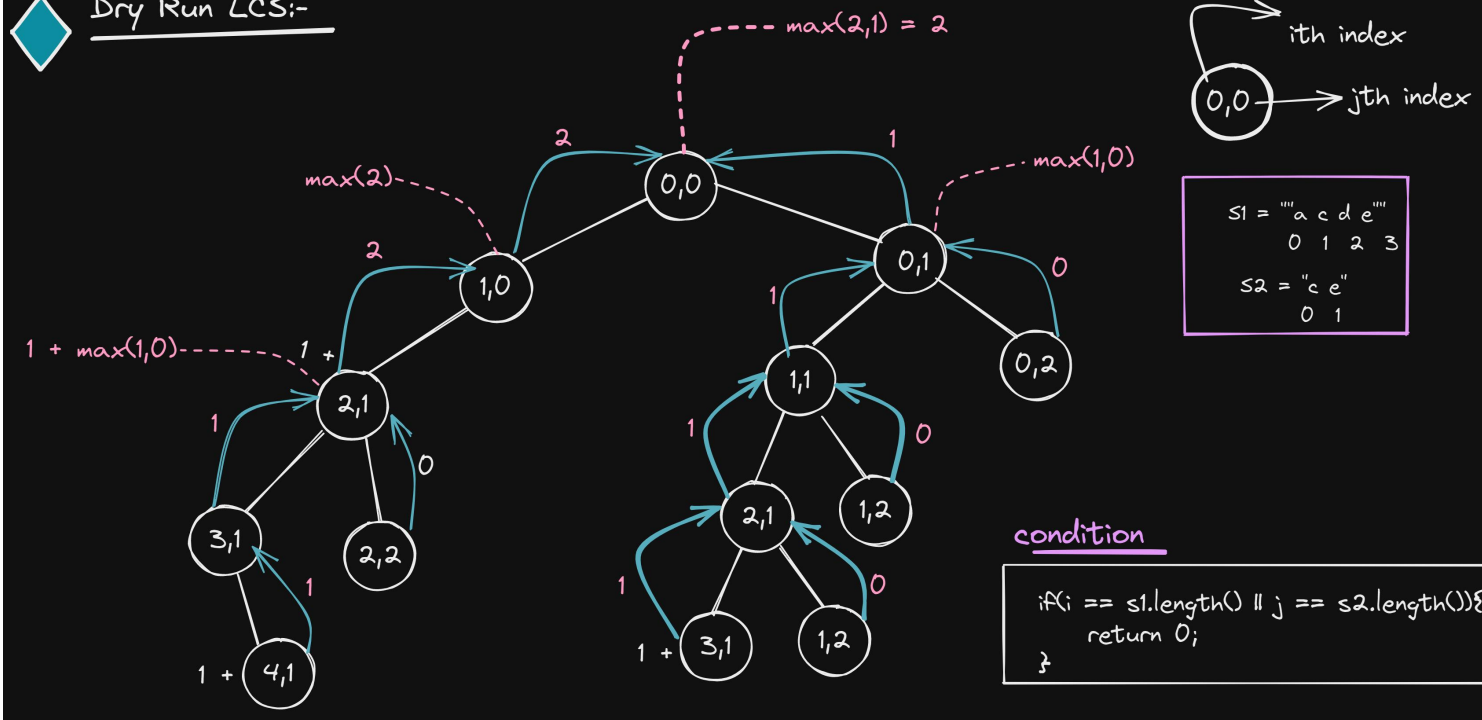
If (S1[i] == S2[j])

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

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

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

Dry Run LCS:-



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

S2 = "c e"
0 1

condition

```
if(i == s1.length() || j == s2.length()){
    return 0;
}
```

Recursion

Time-complexity	O(2^N)
Space-complexity	O(N)

Dynamic-Programming

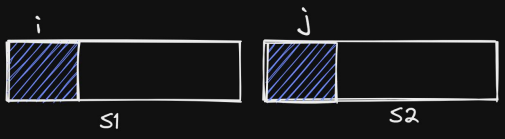
Time-complexity	O(N)
Space-complexity	O(N)

Longest Palindromic Subsequence (LPS)

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.

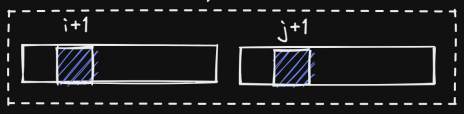
There are only Two possiblity:-

S1[i] = S2[j]

S1[i] != S2[j]

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

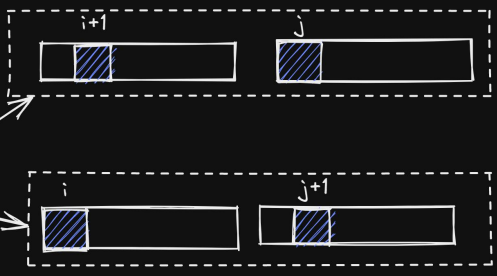
(1 + S1[i+1],S2[j+1])

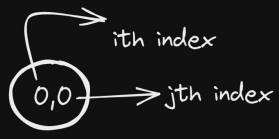


If (S1[i] != S2[j])

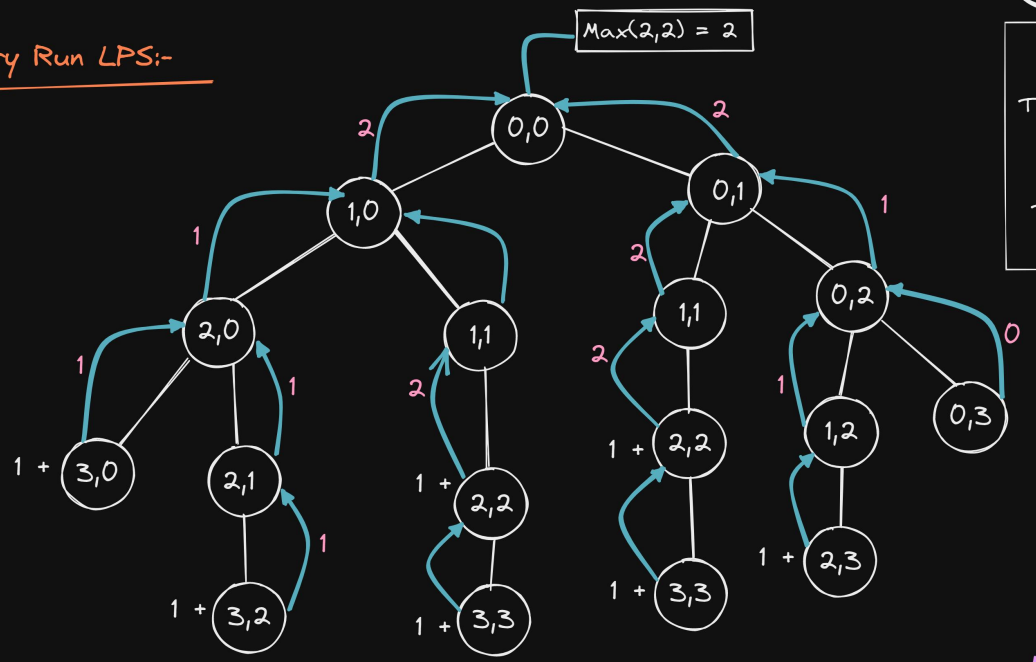
(S1[i+1], S2[j])

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





Dry Run LPS:-



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

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

Recursion

Time-complexity ⟶ O(2^N)

Space-complexity ⟶ O(N)

Dynamic-Programing

Time-complexity ⟶ O(N)

Space-complexity ⟶ O(N)

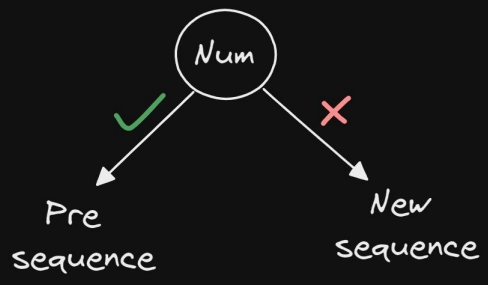
condition

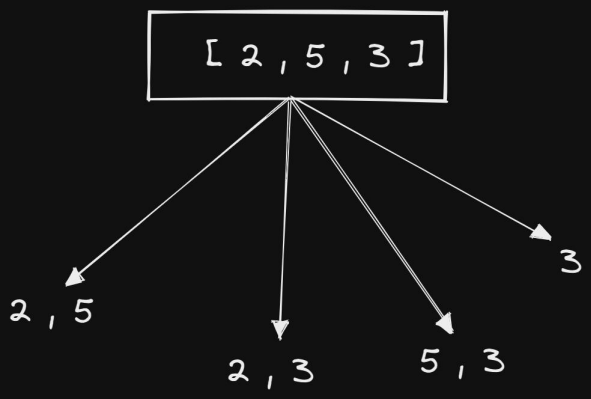
```
if i == s.length || j == s.length
    return 0;
```

Longest increasing subsequnece

[10 , 9 , 2 , 5 , 3 , 7 , 101 , 18]

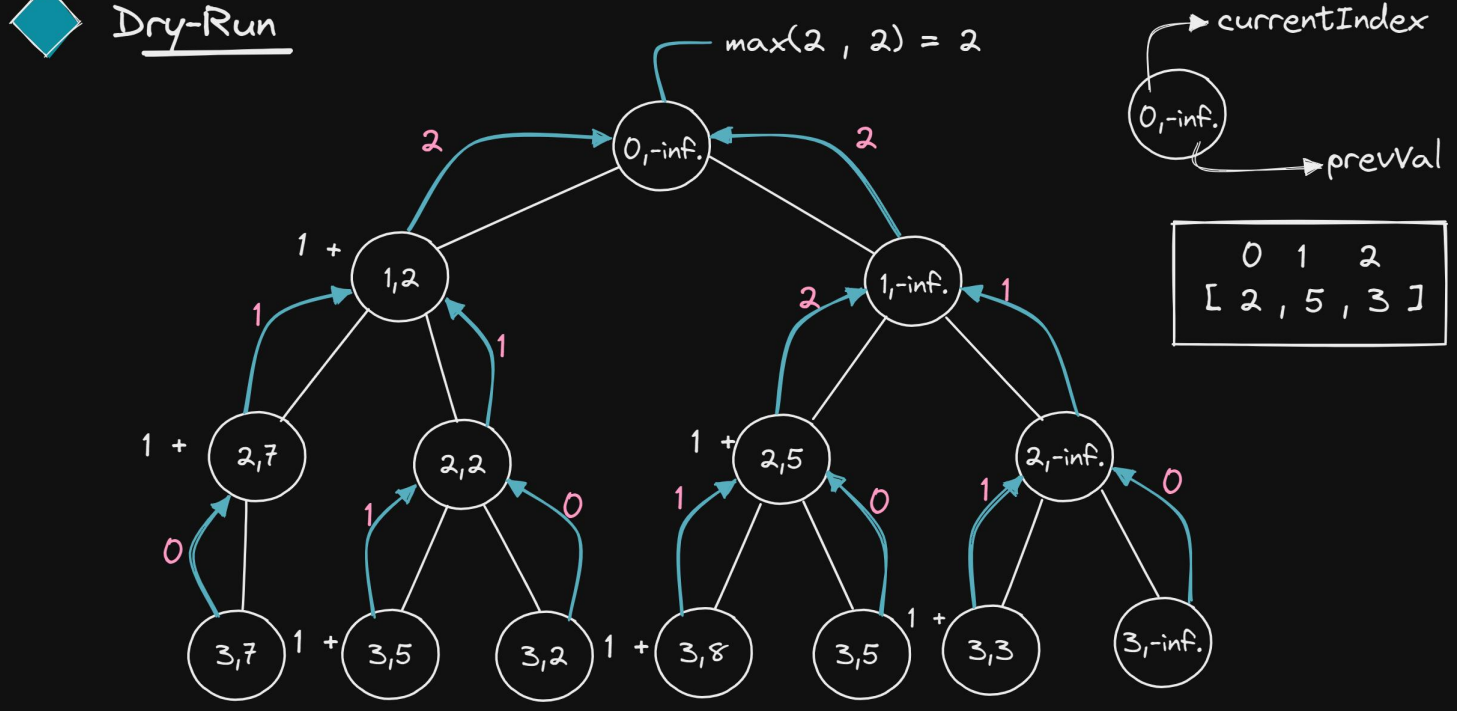
The longest increasing subsequence is [2 , 3 , 7 , 101] , therefore the length is 4.





→ so, from [2 , 5 , 3] LIS is of 2 length.

Dry-Run



Recursion

Time-complexity	O(2^N)
Space-complexity	O(N)

Dynamic-Programming

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

Longest Increasing Path in a Matrix

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

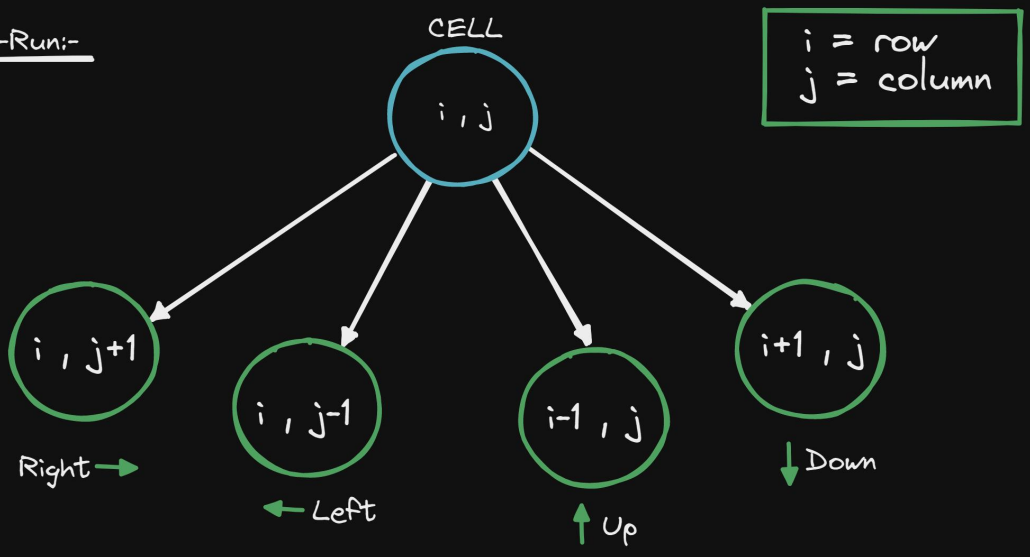
→ From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

Input:-

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;
}
```

Recursion

Time-complexity → O(4^N)

Space-complexity → O(N)

Dynamic-Programming

Time-complexity → O(N)

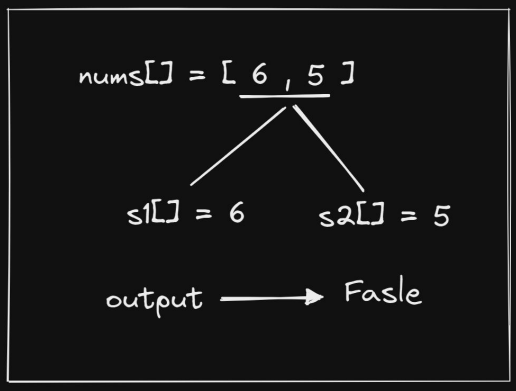
Space-complexity → O(N)

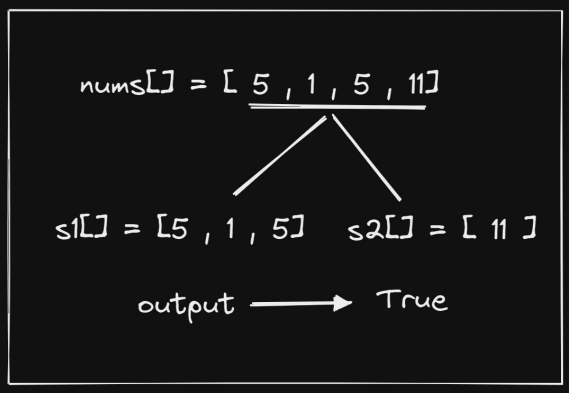
Partition Equal Subset Sum

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

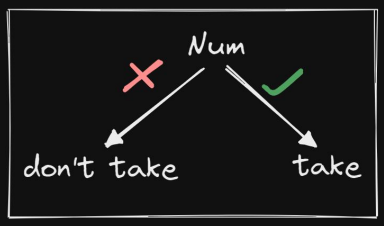
1st check - if its sum is odd, then return false

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

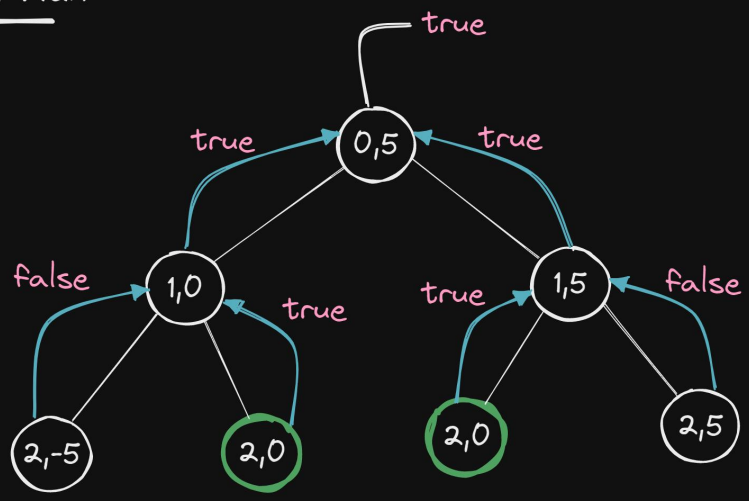


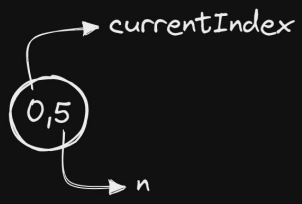


sum(s1) + sum(s2) = sum(nums)
sum(s1) + sum(s1) = sum(nums)
2sum(s1) = sum(nums)
sum(s1) = sum(nums)/2



Dry-Run





conditions:-

```
if(currentIndex >= nums.length && n != 0)
    return False;
if(currentIndex >= nums.length && n == 0)
    return True;
```

Recursion

Time-complexity	O(2^N)
Space-complexity	O(N)

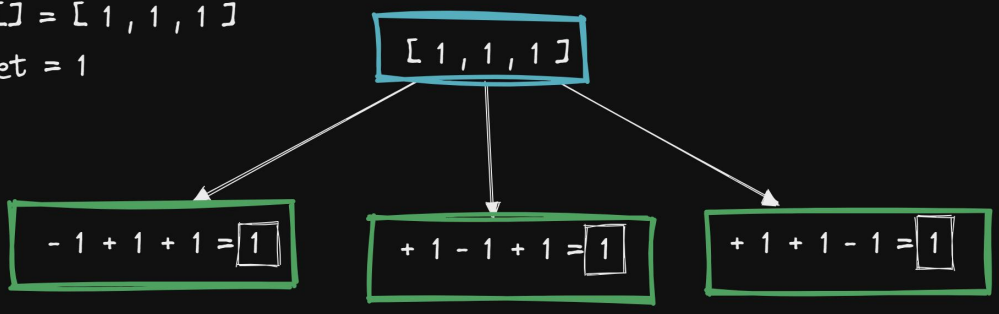
Dynamic-Programing

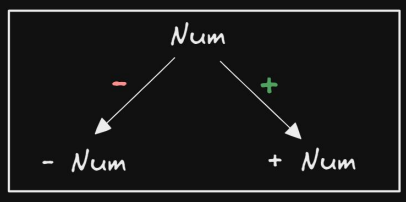
Time-complexity	O(N)
Space-complexity	O(N)

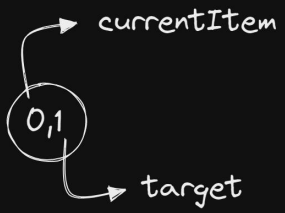
Target Sum

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.

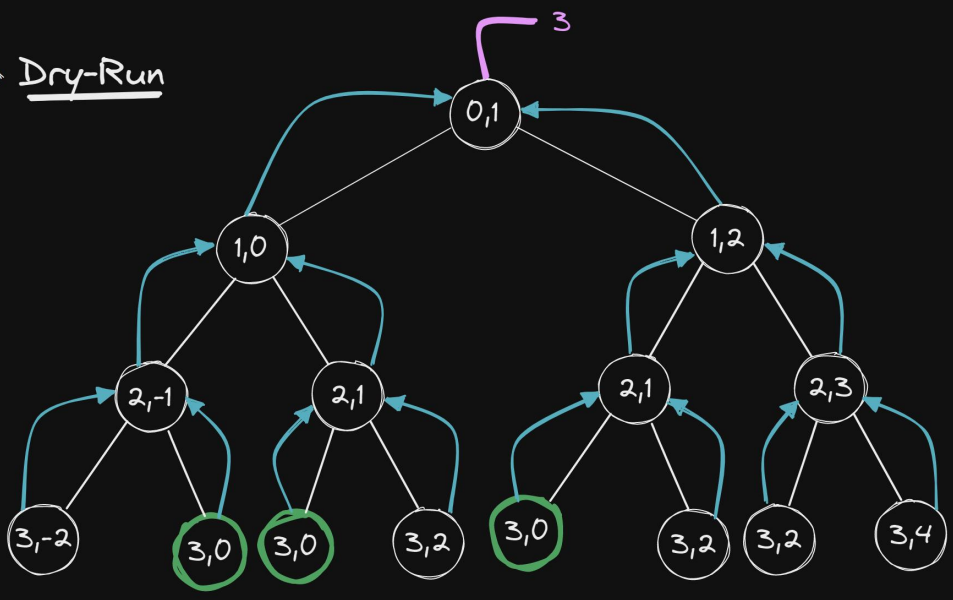
nums[] = [1 , 1 , 1]
target = 1







Dry-Run



conditions

```
if((currentItem >= nums.length) && (target != 0)){
        return 0;
    }
if((currentItem >= nums.length) && (target == 0)){
        return 1;
    }
```

```
Leftcall = n+1,target-num[n]
Rightcall = n+1,target-(-num[n])
```

Recursion

Time-complexity ⟶ O(2^N)
Space-complexity ⟶ O(N)

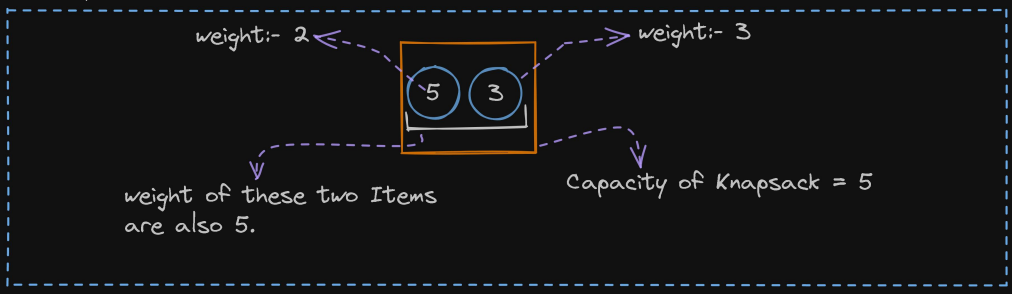
Dynamic-Programming

Time-complexity ⟶ O(N)
Space-complexity ⟶ O(N)

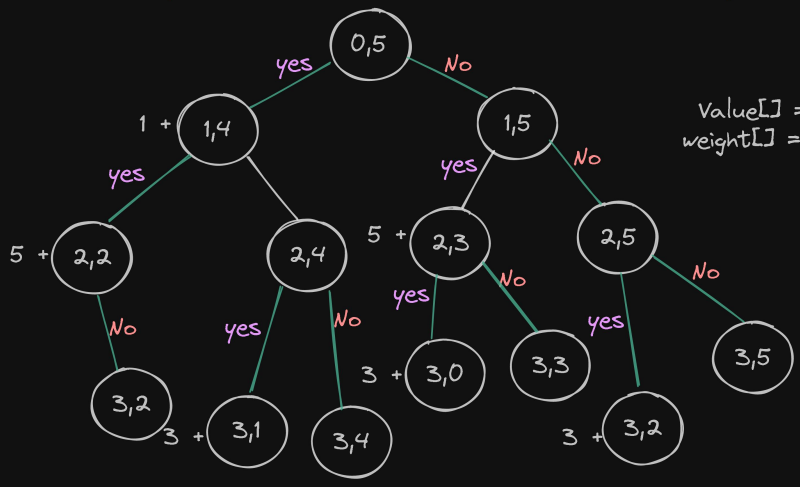
0-1 Knapsack

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?



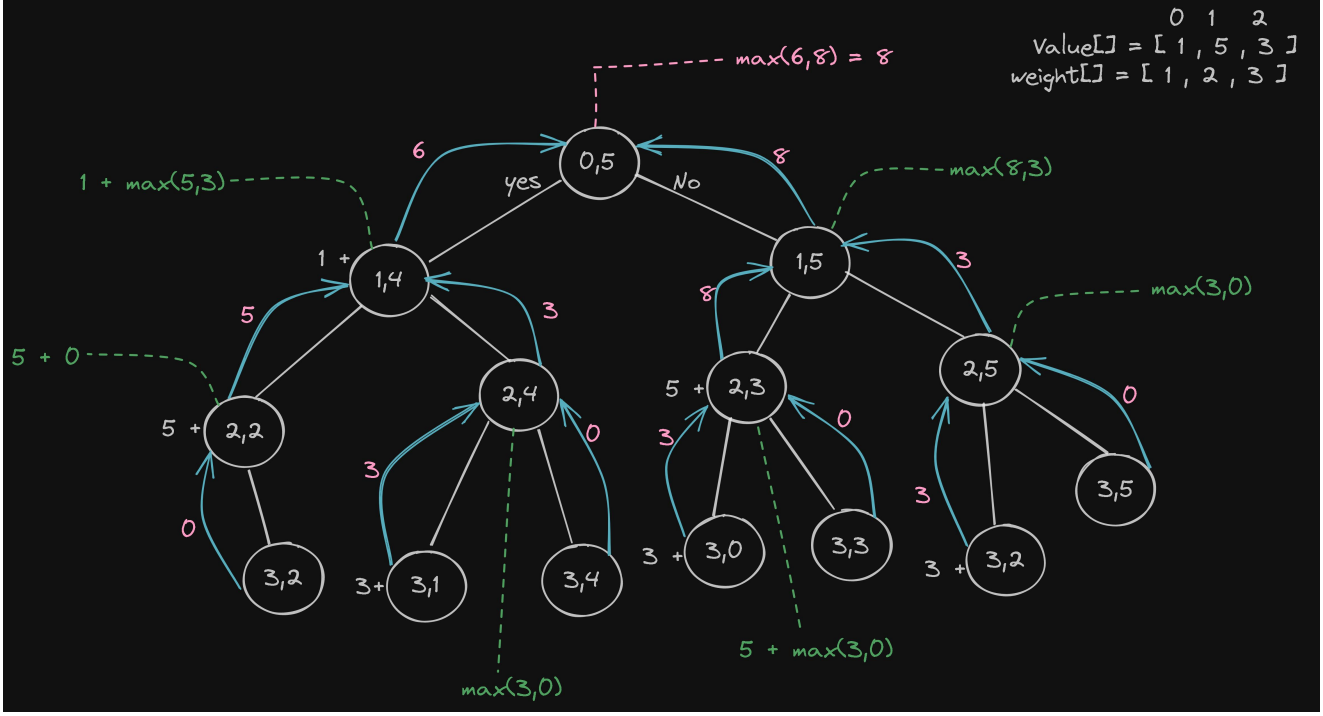
we have two option weather to take element in knapsack or not




```
          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]
```

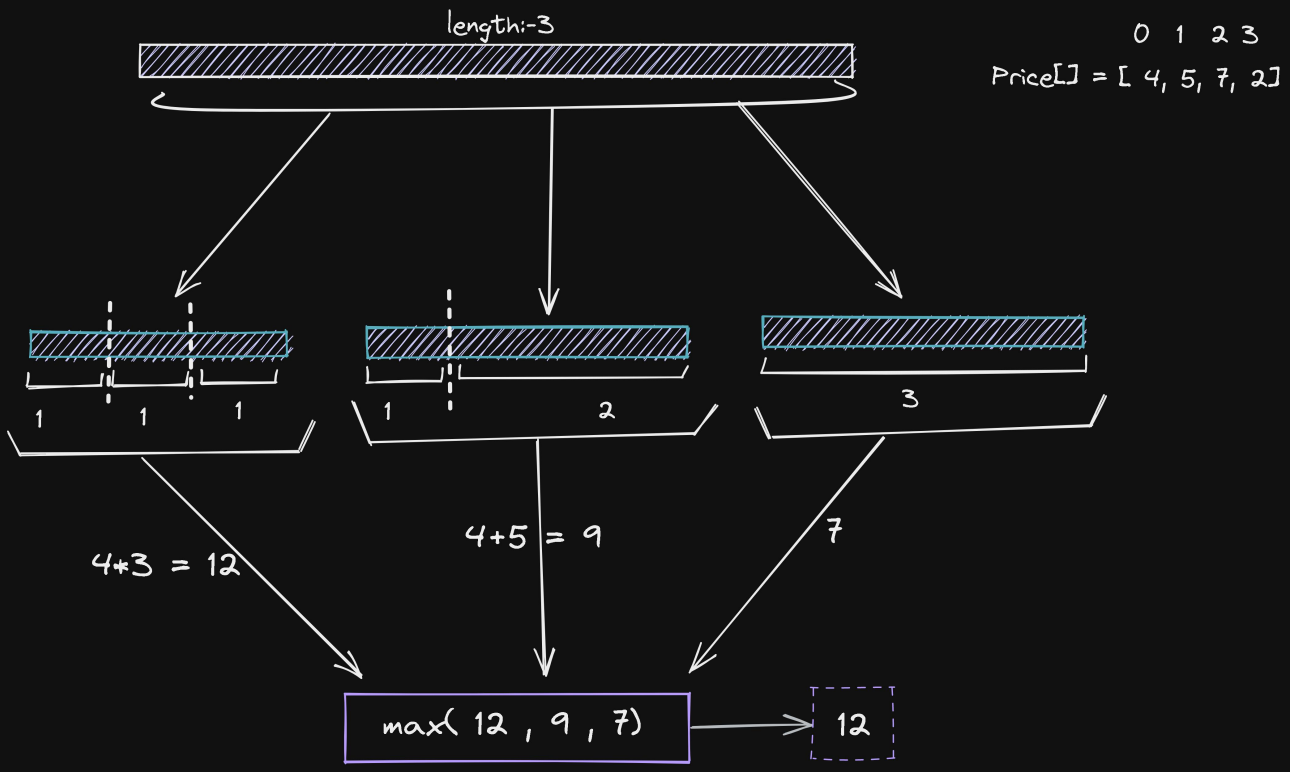
Un-Bounded KnapSack

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

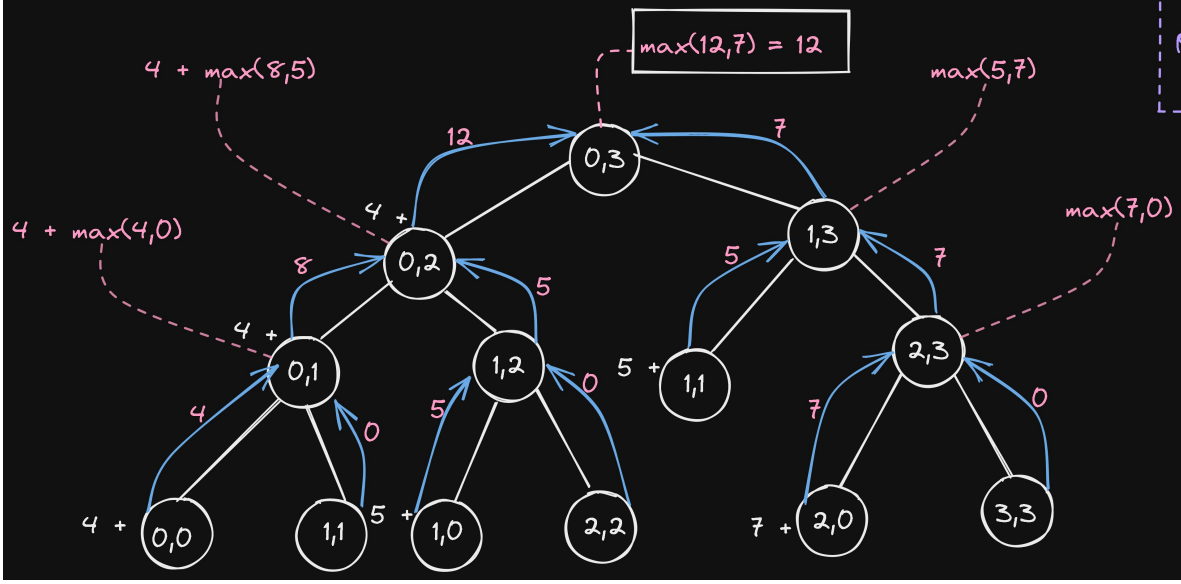
Output ⟶ 12



0 1 2 3
Price[] = [4, 5, 7, 2]

Rode cutting Problem: -

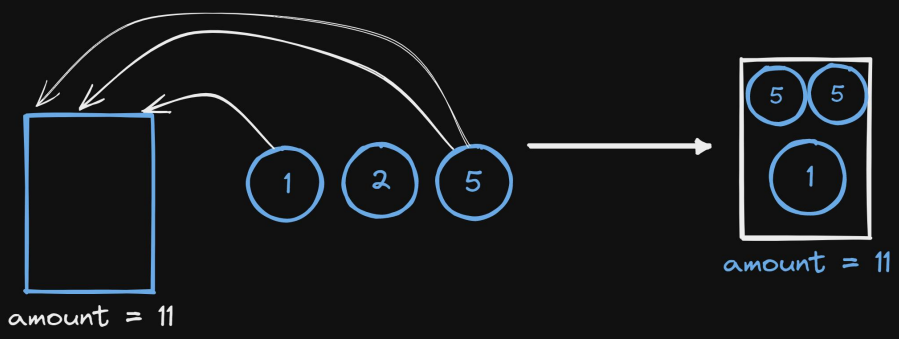
index's: - 0 1 2 3
price[] = [4, 5, 7, 2]
cut's: - 1 2 3 4



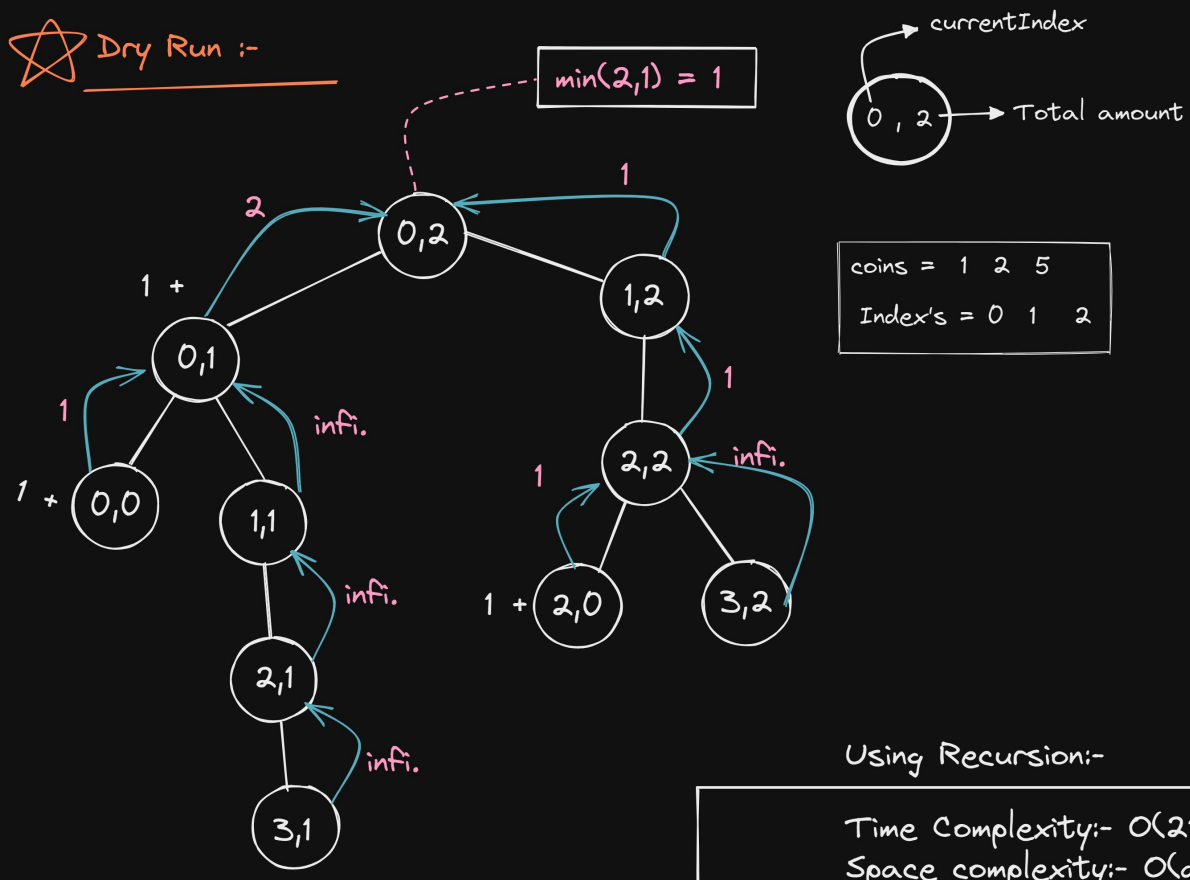
Coin Change

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 :-



Base Case:-

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

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

Using Recursion:-

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

Using DP:-

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

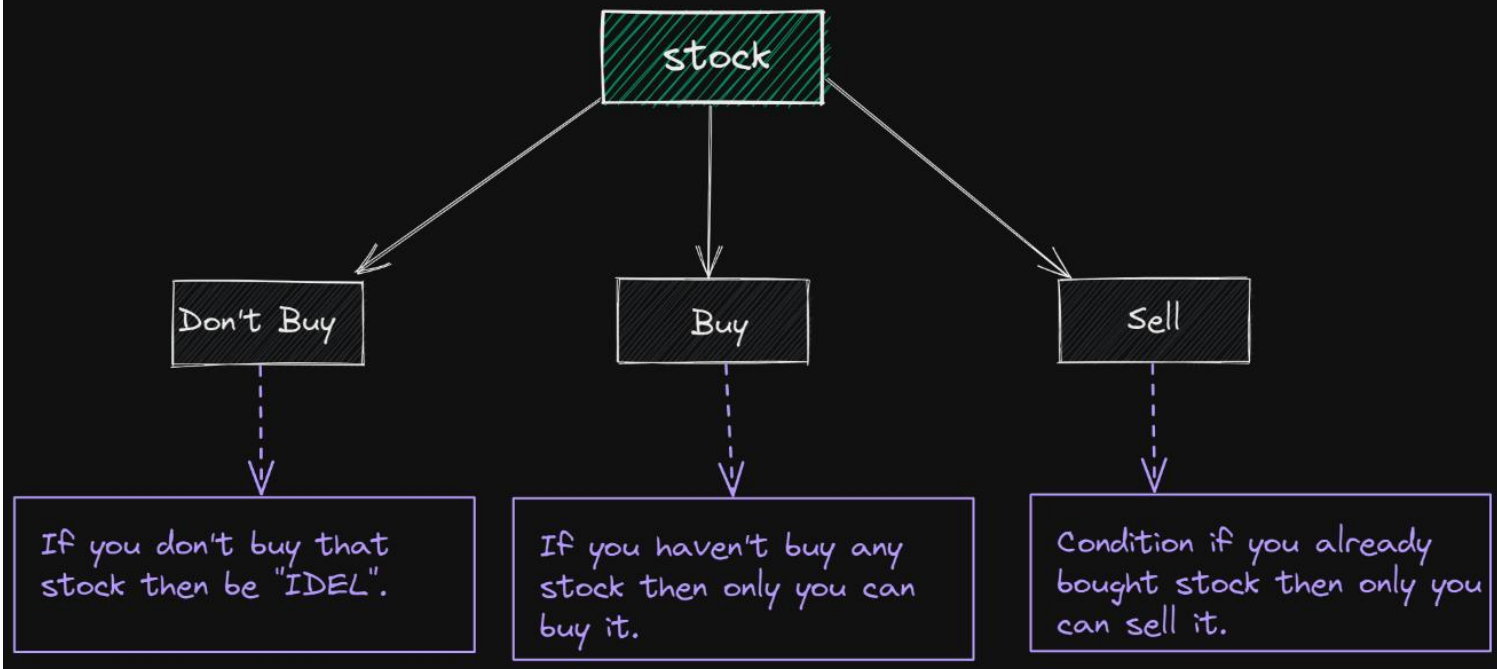
n = coins.lenth
k = amount

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

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]

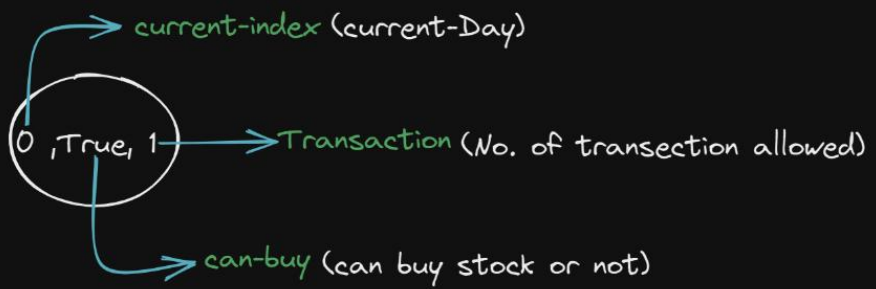
Output(profit) : 5

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

Buy Sell

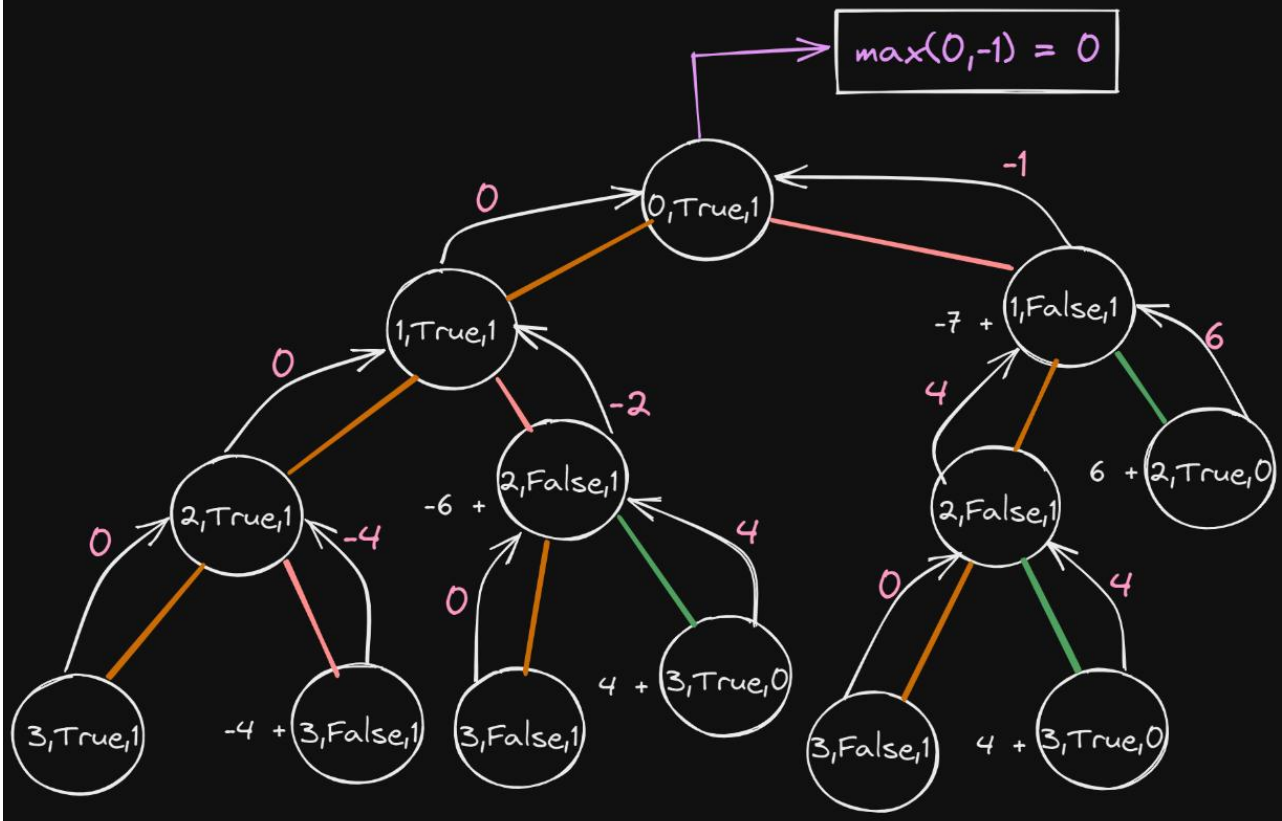
Note:- we have to check multiple ways (Recursion) to get maximum amount of profit.

Best Time to Buy and Sell Stock I



Ideal
Buy
Sell

```
Indexs =  0   1   2
prices = [ 7 , 6 , 4 ]
```



Conditions:--

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

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)

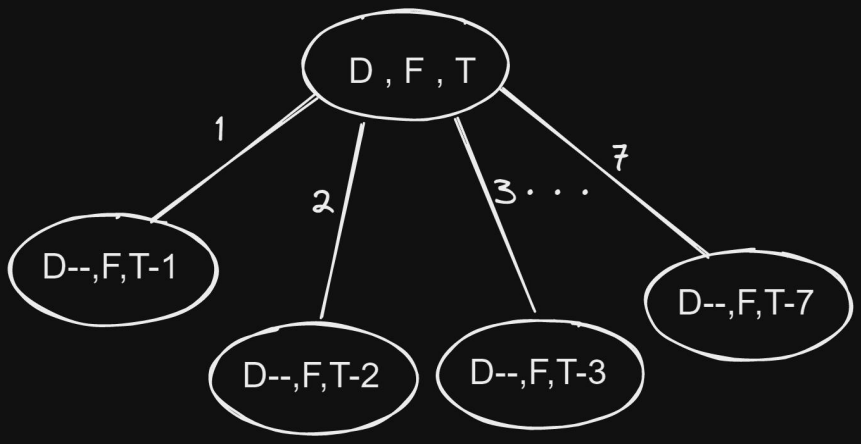
Number of Dice Rolls With Target Sum

You have d dice and each die has f faces numbered 1, 2, ..., f. You are given three integers d, f, and target.

F ⟶ faces(1,2,3....)

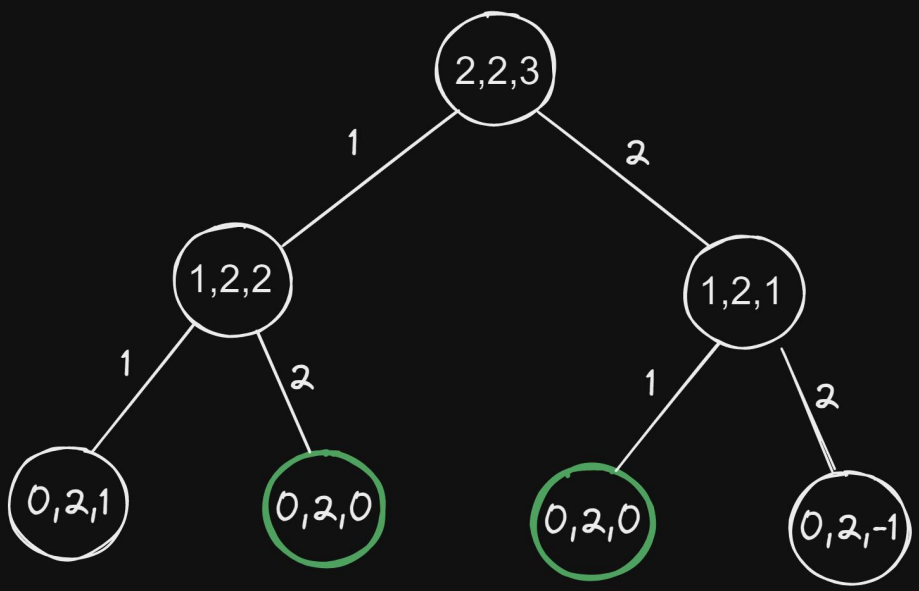
D ⟶ no. of dice

T ⟶ target sum



Dry-Run

D = 2 , F = 2 , target = 3



Recursion

Time-complexity	O(F^N)
Space-complexity	O(N)

Dynamic-programming

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