70. Climbing Stairs

09 March 2022 06:26 PM

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?

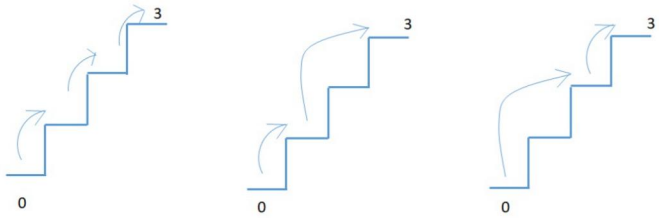
Example 1:

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

Example 2:

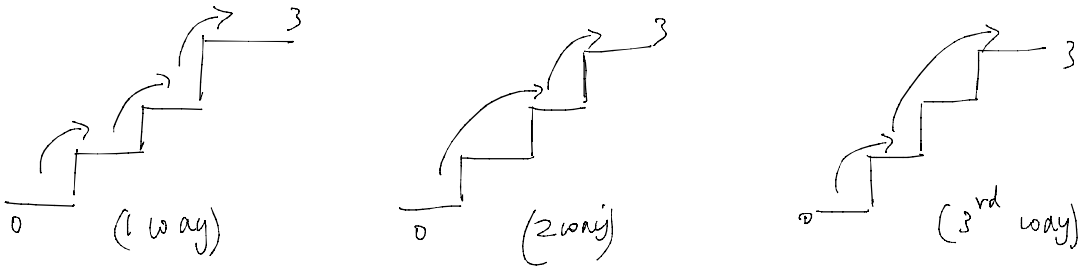
```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the
top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Problem Statement: Given a number of stairs. Starting from the 0th stair we need to climb to the "Nth" stair. At a time we can climb either one or two steps. We need to return the total number of distinct ways to reach from 0th to Nth stair.



Given N=3, there are ways to reach stair 3 from stair 0

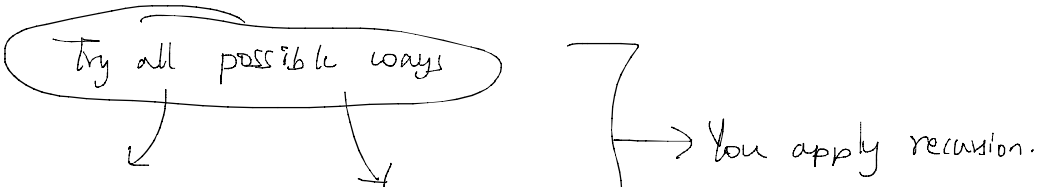
for n = 3

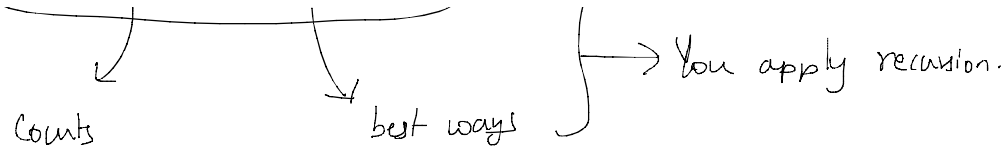


1D problem

Identify a DP problem ?

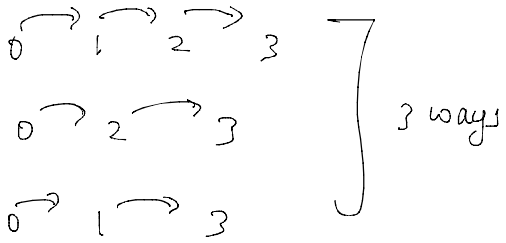
* Count the total no of ways
* Given multiple way of doing a task, which way will give the minimum/maximum output.





After recursion → then do dynamic programming.

n=3



Steps to solve the problem after identification:

S 1) Try to represent the problem in term of indexes.

S 2) Try all possible ways / choice at every index according to the problem statement.

S 3) If the question states:

- Count all ways : return sum of all choice / ways
- Find max/mini : return all the choice/ways with max/mini output

Climbing Stairs:

S1 – we will assume n stairs as index from 0 to N

$f(n) \rightarrow$ no of ways $(0 \rightarrow n)$

```
f(index)
{
    if (ind == 0) return 1
```

S2 — Do all possible stuff, in the question you can jump with '1' or '2'.

S3 — We asks to count the total no of distinct ways, we will return the sum of all the choice in our recursive function.

```
f(ind)
{
  if (ind == 0) return 1;
  if (ind == 1) return 0;
  left = f(ind - 1);
  right = f(ind - 2)
  return left + right
}
```

→ edge case because if you call f(1) then we will reach stair number '-1' f(1-2) = -1 which is not defined.

→ This is similar to fibo.

Tabulation Approach :

1) Declare a dp[] array of size n+1
2) Initialize the base condition value.
3) for loop from 2 to n and for every index set its value as dp[i-1] dp[i-2]

```
class Solution {
public:
    int climbStairs(int n) {
        // Tabulation
        vector<int> dp(n+1, -1);
        dp[0] = 1;
        dp[1] = 1;

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

Java • Autocomplete

```
class Solution {
    public int climbStairs(int n) {
        int dp[] = new int[n+1];
        Arrays.fill(dp, -1);
        dp[0] = 1;
        dp[1] = 1;
        for(int i=2; i<=n; i++){
            dp[i] = dp[i-1] + dp[i-2];
        }
        return dp[n];
    }
}
```

T.C ⇒ O(n) S.C ⇒ O(n) → for the array

Space Optimization :

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

for any i, we do need only the last two values in the array. So, there is a need to maintain a whole array for it ?

The answer is NO, let us call $dp[i-1]$ as prev and $dp[i-2]$ as prev2

→ for each iteration curi and prev becomes the next iteration prev and prev2 respectively

→ Therefore after calculating curi, if we update prev and prev2 according to the next step, we will always get the answer.

→ After the iterative op has ended we can simply return prev as our answer.

```
class Solution {
    public int climbStairs(int n) {
        // Space Optimization
        int prev2 = 1;
        int prev = 1;
        for(int i=2; i<=n; i++){
            int curi = prev2 + prev;
            prev2 = prev;
            prev = curi;
        }
        return prev;
    }
}
```

T.C ⇒ $O(n)$

S.C ⇒ $O(1)$