**70. Climbing Stairs**

**Question**

You are climbing a stair case. It takes *n* steps to reach to the top.

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

**Note:** Given *n* will be a positive integer.

**Quick Navigation**

* [Summary](https://leetcode.com/articles/climbing-stairs/" \l "summary)
* [Solution](https://leetcode.com/articles/climbing-stairs/" \l "solution)
  + [Approach #1 Brute Force [Time Limit Exceeded]](https://leetcode.com/articles/climbing-stairs/" \l "approach-1-brute-force-time-limit-exceeded)
  + [Approach #2 Recursion with memorization [Accepted]](https://leetcode.com/articles/climbing-stairs/" \l "approach-2-recursion-with-memorization-accepted)
  + [Approach #3 Dynamic Programming [Accepted]](https://leetcode.com/articles/climbing-stairs/" \l "approach-3-dynamic-programming-accepted)

**Summary**

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

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

**Solution**

**Approach #1 Brute Force [Time Limit Exceeded]**

**Algorithm**

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 climbStairsclimbStairsclimbStairs for step 111 and 222, and return the sum of returned values of both functions.

climbStairs(i,n)=(i+1,n)+climbStairs(i+2,n)climbStairs(i,n)=(i + 1, n) + climbStairs(i + 2, n)climbStairs(i,n)=(i+1,n)+climbStairs(i+2,n), where iii defines the current step and nnn defines the destination step.

**Java**

**public** **class** **Solution** **{**

**public** **int** **climbStairs(int** n**)** **{**

**return** climb\_Stairs**(**0**,** n**);**

**}**

**public** **int** **climb\_Stairs(int** i**,** **int** n**)** **{**

**if** **(**i **>** n**)** **{**

**return** 0**;**

**}**

**if** **(**i **==** n**)** **{**

**return** 1**;**

**}**

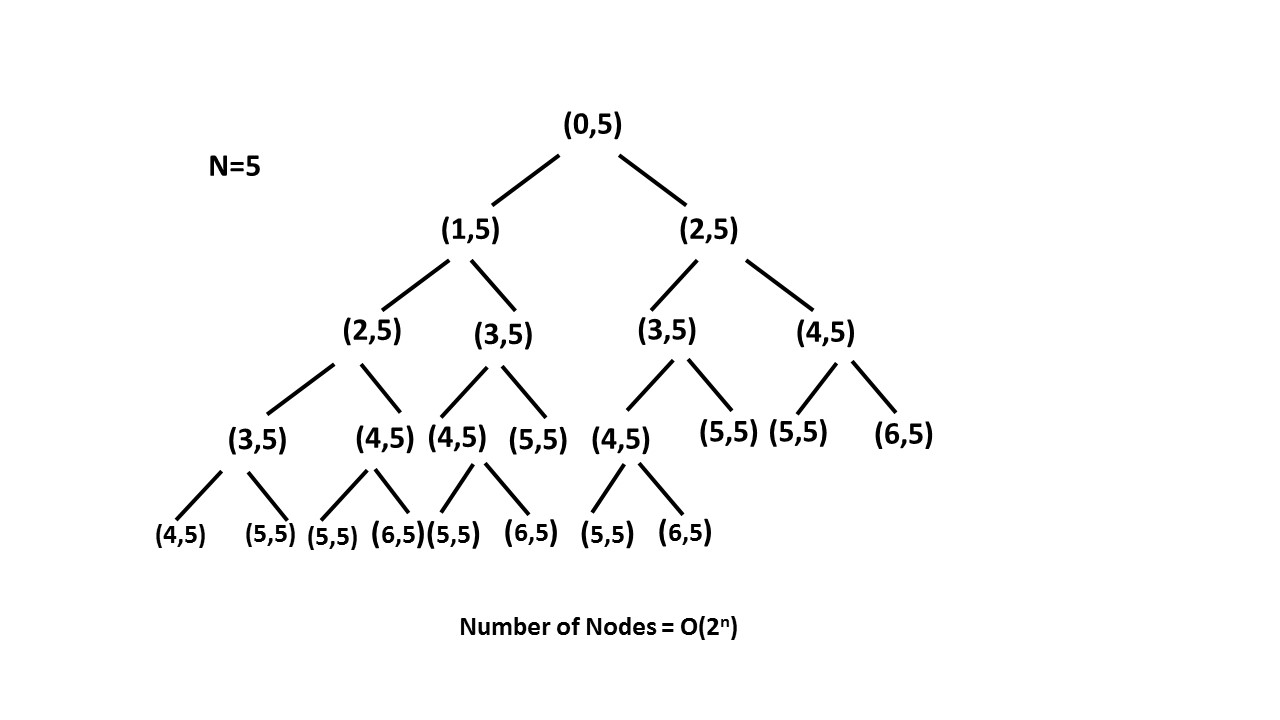
**return** climb\_Stairs**(**i **+** 1**,** n**)** **+** climb\_Stairs**(**i **+** 2**,** n**);**

**}**

**}**

**Complexity Analysis**

* Time complexity : O(2n)O(2^n)O(2​n​​). Size of recursion tree will be 2n2^n2​n​​.

Recursion tree for n=5 would be like this

* Space complexity : O(n)O(n)O(n). The depth of the recursion tree can go upto nnn.

**Approach #2 Recursion with memorization [Accepted]**

**Algorithm**

In the previous approach we are redundantly calculating the result for every step. Instead, we can store the result at each step in memomemomemo array 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 memomemomemo array and reducing the size of recursion tree upto nnn.

**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**];**

**}**

**}**

**Complexity Analysis**

* Time complexity : O(n)O(n)O(n). Size of recursion tree can go upto nnn.
* Space complexity : O(n)O(n)O(n). The depth of recursion tree can go upto nnn.

**Approach #3 Dynamic Programming [Accepted]**

**Algorithm**

As we can see this problem can be broken into subproblems, and it contains the optimal substructure property i.e. its optimal solution can be constructed efficiently from optimal solutions of its subproblems, we can use dynamic programming to solve this problem.

One can reach ithi^{th}i​th​​ step in one of the two ways:

1. Taking a single step from (i−1)th(i-1)^{th}(i−1)​th​​ step.
2. Taking a step of 222 from (i−2)th(i-2)^{th}(i−2)​th​​ step.

So, the total number of ways to reach ithi^{th}i​th​​ is equal to sum of ways of reaching (i−1)th(i-1)^{th}(i−1)​th​​ step and ways of reaching (i−2)th(i-2)^{th}(i−2)​th​​ step.

Let dp[i]dp[i]dp[i] denotes the number of ways to reach on ithi^{th}i​th​​ step.

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

Example:

**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**];**

**}**

**}**

**Complexity Analysis**

* Time complexity : O(n)O(n)O(n). Single loop upto nnn.
* Space complexity : O(n)O(n)O(n). dpdpdp array of size nnn is used.