

## 70. Climbing Stairs

Jan. 31, 2017

Question

Editorial Solution

### 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
- Solution
  - Approach #1 Brute Force [Time Limit Exceeded]
  - Approach #2 Recursion with memorization [Accepted]
  - Approach #3 Dynamic Programming [Accepted]
  - Approach #4 Fibonacci Number [Accepted]:
  - Approach #5 Binets Method [Accepted]:
  - Approach #6 Fibonacci Formula [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 *climbStairs* for step 1 and 2, and return the sum of returned values of both functions.

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

##### Java

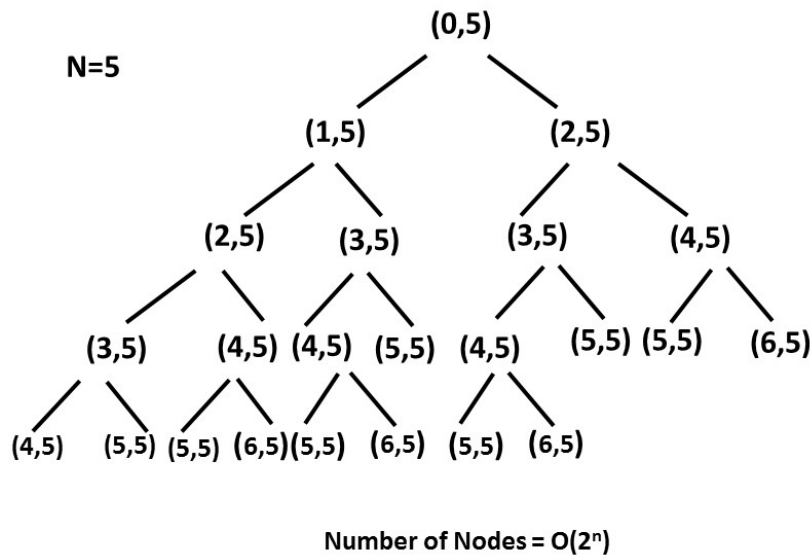
```
public class Solution {  
    public int climbStairs(int n) {  
        climb_Stairs(0, n);  
    }  
    public int climb_Stairs(int i, int n) {  
        if (i > n) {  
            return 0;  
        }  
        if (i == n) {  
            return 1;  
        }  
        return climb_Stairs(i + 1, n) + climb_Stairs(i + 2, n);  
    }  
}
```

[✉ Send Feedback \(mailto:admin@leetcode.com?subject=Feedback\)](mailto:admin@leetcode.com?subject=Feedback)

**Complexity Analysis**

- Time complexity :  $O(2^n)$ . Size of recursion tree will be  $2^n$ .

Recursion tree for  $n=5$  would be like this:



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

**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 *memo* 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 *memo* array and reducing the size of recursion tree upto  $n$ .

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

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

✉ Send Feedback (mailto:admin@leetcode.com?subject=Feedback)

## 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  $i^{th}$  step in one of the two ways:

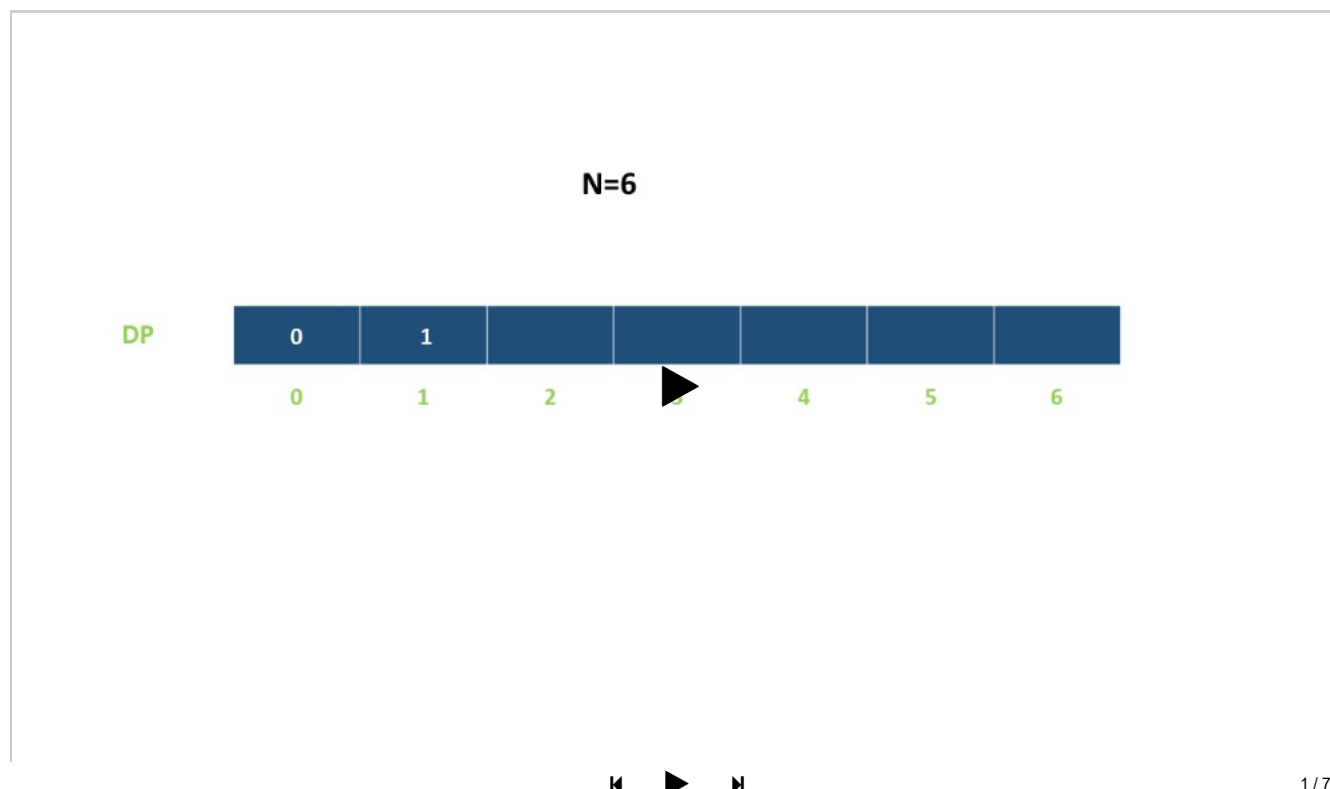
1. Taking a single step from  $(i - 1)^{th}$  step.
2. Taking a step of 2 from  $(i - 2)^{th}$  step.

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

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

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

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 :  $O(n)$ . Single loop upto  $n$ .
- Space complexity :  $O(n)$ .  $dp$  array of size  $n$  is used.

## Approach #4 Fibonacci Number [Accepted]:

✉ Send Feedback (mailto:admin@leetcode.com?subject=Feedback)

### Algorithm

In the above approach we have used  $dp$  array where  $dp[i] = dp[i - 1] + dp[i - 2]$ . It can be easily analysed that  $dp[i]$  is nothing but  $i^{th}$  fibonacci number.

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$

Now we just have to find  $n^{th}$  number of the fibonacci series having 1 and 2 their first and second term respectively i.e.  $Fib(1) = 1$  and  $Fib(2) = 2$ .

### Java

```
public class Solution {
    public int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }
        int first = 1;
        int second = 2;
        for (int i = 3; i <= n; i++) {
            int third = first + second;
            first = second;
            second = third;
        }
        return second;
    }
}
```

### Complexity Analysis

- Time complexity :  $O(n)$ . Single loop upto  $n$  is required to calculate  $n^{th}$  fibonacci number.
- Space complexity :  $O(1)$ . Constant space is used.

## Approach #5 Binets Method [Accepted]:

### Algorithm

This is an interesting solution which uses matrix multiplication to obtain the  $n^{th}$  Fibonacci Number. The matrix takes the following form:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Let's say  $Q = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$

As per the method, the  $n^{th}$  Fibonacci Number is given by  $Q^{n-1}[0, 0]$

Let's look at the proof of this method.

We can prove this method using Mathematical Induction. We know, this matrix gives the correct result for the  $3^{rd}$  term(base case). Since

$$Q^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}. \text{ This proves that the base case holds.}$$

Assume that this method holds for finding the  $n^{th}$  Fibonacci Number i.e.  $F_n = Q^{n-1}[0, 0]$ , where  $Q^{n-1} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix}$ .

Now, we need to prove that with the above two conditions holding true, the method is valid for finding the  $(n + 1)^{th}$  Fibonacci Number i.e.

$$F_{n+1} = Q^n[0, 0]$$

$$\text{Proof: } Q^n = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot Q^n = \begin{bmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{bmatrix} Q^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Thus,  $F_{n+1} = Q^n[0, 0]$ . This completes the proof of this method.

The only variation we need to do for our problem is that we need to modify the initial terms to 2 and 1 instead of 1 and 0 in the Fibonacci series. Or, another way is to use the same initial  $Q$  matrix and use  $result = Q^n[0, 0]$  to get the final result. This happens because the initial terms we have to use are the 2nd and 3rd terms of the otherwise normal Fibonacci Series.

### Java

```

public class Solution {
    public int climbStairs(int n) {
        int[][] q = {{1, 1}, {1, 0}};
        int[][] res = pow(q, n);
        return res[0][0];
    }
    public int[][] pow(int[][] a, int n) {
        int[][] ret = {{1, 0}, {0, 1}};
        while (n > 0) {
            if ((n & 1) == 1) {
                ret = multiply(ret, a);
            }
            n >>= 1;
            a = multiply(a, a);
        }
        return ret;
    }
    public int[][] multiply(int[][] a, int[][] b) {
        int[][] c = new int[2][2];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
            }
        }
        return c;
    }
}

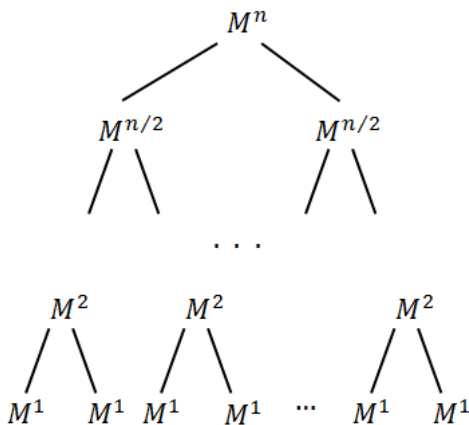
```

### Complexity Analysis

- Time complexity :  $O(\log(n))$ . Traversing on  $\log(n)$  bits.
- Space complexity :  $O(1)$ . Constant space is used.

Proof of Time Complexity:

Let's say there is a matrix  $M$  to be raised to power  $n$ . Suppose,  $n$  is the power of 2. Thus,  $n = 2^i, i \in \mathbb{N}$ , where  $\mathbb{N}$  represents the set of natural numbers(including 0). We can represent in the form of a tree:



Meaning that:

$$M^n = M^{n/2} \cdot M^{n/2} = \dots = \prod_{i=1}^n M^1$$

So, to calculate  $M^n$  matrix, we should calculate  $M^{n/2}$  matrix and multiply it by itself. To calculate  $M^{n/2}$  we would have to do the same with  $M^{n/4}$  and so on.

Obviously, the tree height is  $\log_2 n$ .

Let's estimate  $M^n$  calculation time.  $M$  matrix is of the same size in any power. Therefore, we can perform the multiplication of two matrices in any power in  $O(1)$ . We should perform  $\log_2 n$  of such multiplications. So,  $M^n$  calculation complexity is  $O(\log_2 n)$ .

In case, the number  $n$  is not a power of two, we can break it in terms of powers of 2 using its binary representation

$$n = \sum_{p \in P} 2^p, \text{ where } P \subset \mathbb{N}.$$

Thus, we can obtain the final result using:

$$M^n = \prod_{p \in P} M^{2^p}$$

This is the method we've used in our implementation. Again, the complexity remains  $O(\log_2 n)$  as we have limited the number of multiplications to  $O(\log_2 n)$ .

## Approach #6 Fibonacci Formula [Accepted]:

### Algorithm

We can find  $n^{th}$  fibonacci number using this formula:

$$F_n = 1/\sqrt{5} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right]$$

For the given problem, the Fibonacci sequence is defined by  $F_0 = 1, F_1 = 1, F_2 = 2, F_{n+2} = F_{n+1} + F_n$ . A standard method of trying to solve such recursion formulas is assume  $F_n$  of the form  $F_n = a^n$ . Then, of course,  $F_{n+1} = a^{n+1}$  and  $F_{n+2} = a^{n+2}$  so the equation becomes  $a^{n+2} = a^{n+1} + a^n$ . If we divide the entire equation by  $a^n$  we arrive at  $a^2 = a + 1$  or the quadratic equation

$$a^2 - a - 1 = 0.$$

Solving this by the quadratic formula, we get:

$$a = 1/\sqrt{5} \left( \left( \frac{1 \pm \sqrt{5}}{2} \right) \right).$$

The general solution, thus takes the form:

$$F_n = A \left( \frac{1+\sqrt{5}}{2} \right)^n + B \left( \frac{1-\sqrt{5}}{2} \right)^n$$

For  $n = 0$ , we get  $A + B = 1$

For  $n = 1$ , we get  $A \left( \frac{1+\sqrt{5}}{2} \right) + B \left( \frac{1-\sqrt{5}}{2} \right) = 1$

Solving the above equations, we get:

$$A = \left( \frac{1+\sqrt{5}}{2\sqrt{5}} \right), B = \left( \frac{1-\sqrt{5}}{2\sqrt{5}} \right)$$

Putting these values of  $A$  and  $B$  in the above general solution equation, we get:

$$F_n = 1/\sqrt{5} \left( \left( \frac{1+\sqrt{5}}{2} \right)^{n+1} - \left( \frac{1-\sqrt{5}}{2} \right)^{n+1} \right)$$

### Java

```
public class Solution {
    public int climbStairs(int n) {
        double sqrt5=Math.sqrt(5);
        double fibn=Math.pow((1+sqrt5)/2,n+1)-Math.pow((1-sqrt5)/2,n+1);
        return (int)(fibn/sqrt5);
    }
}
```

### Complexity Analysis

- Time complexity :  $O(\log(n))$ . *pow* method takes  $\log(n)$  time.
- Space complexity :  $O(1)$ . Constant space is used.

Analysis written by: @vinod23 (<https://leetcode.com/vinod23>)

(/ratings/107/84/?return=/articles/climbing-stairs/) (/ratings/107/84/?return=/articles/climbing-stairs/) (/ratings/107/84/?return=/articles/climbing-stairs/) (/ratings/107/84/?return=/articles/climbing-stairs/)

Average Rating: 4.94 (36 votes)

< Previous (/articles/non-overlapping-intervals/)

Next > (/articles/find-median-from-data-stream/)



Join the conversation

Signed in as **lixiaozheng.good**.

Post a Reply

View original thread (<https://discuss.leetcode.com/topic/77396>)

✉ Send Feedback (<mailto:admin@leetcode.com?subject=Feedback>)

---

[Frequently Asked Questions \(/faq/\)](#) | [Terms of Service \(/tos/\)](#)

[Privacy](#)

Copyright © 2017 LeetCode