Climbing stairs to reach the top - GeeksforGeeks

Source: https://www.geeksforgeeks.org/count-ways-reach-nth-stair/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Climbing stairs to reach the top Last Updated : 29 Jan, 2026 Given n stairs, and a person standing at the ground wants to climb stairs to reach the top. The person can climb either 1 stair or 2 stairs at a time, count the number of ways the person can reach at the top. Examples: Input: n = 1 Output: 1 Explanation : There is only one way to climb 1 stair. Input: n = 2 Output: 2 Explanation : There are two ways to reach 2th stair: {1, 1} and {2}. Input: n = 4 Output: 5 Explanation : There are five ways to reach 4th stair: {1, 1, 1, 1}, {1, 1, 2}, {2, 1, 1}, {1, 2, 1} and {2, 2}. Try it on GfG Practice Table of Content [Naive Approach] Using Recursion – O(2^n) Time and O(n) Space [Better Approach - 1] Using Top-Down DP (Memoization) – O(n) Time and O(n) Space [Better Approach - 2] Using Bottom-Up DP (Tabulation) – O(n) Time and O(n) Space [Space Optimised] Using Space Optimized DP – O(n) Time and O(1) Space [Expected Approach] Using Matrix Exponentiation – O(logn) Time and O(1) Space [Naive Approach] Using Recursion – O(2^n) Time and O(n) Space In this approach, we observe that to reach the nth stair, one can come either from the (n–1)th or (n–2)th stair. Similarly, to reach the (n–1)th stair, we must know the ways to reach the (n–2)th and (n–3)rd stairs, and so on. This way we can observe that this can be done using recursion. Note: The above recurrence relation is same as Fibonacci numbers . C++ //Driver Code Starts #include <iostream> using namespace std ; //Driver Code Ends int countWays (int n) { // Base cases if (n == 0 || n == 1) return 1 ; return countWays (n - 1) + countWays (n - 2); } //Driver Code Starts int main () { int n = 4 ; cout << countWays (n); return 0 ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> //Driver Code Ends int countWays (int n) { // Base cases if (n == 0 || n == 1) return 1 ; return countWays (n - 1) + countWays (n - 2); } //Driver Code Starts int main () { int n = 4 ; printf ("%d " , countWays (n)); return 0 ; } //Driver Code Ends Java //Driver Code Starts class GfG { //Driver Code Ends static int countWays (int n) { // Base cases if (n == 0 || n == 1) return 1 ; return countWays (n - 1) + countWays (n - 2); } //Driver Code Starts public static void main (String [] args) { int n = 4 ; System . out . println (countWays (n)); } } //Driver Code Ends Python def countWays (n): # Base cases if n == 0 or n == 1 : return 1 return countWays (n - 1) + countWays (n - 2) // Driver Code Starts n = 4 print (countWays (n)) // Driver Code Ends C# //Driver Code Starts using System ; class GfG { //Driver Code Ends static int countWays (int n) { // Base cases if (n == 0 || n == 1) return 1 ; return countWays (n - 1) + countWays (n - 2); } //Driver Code Starts static void Main () { int n = 4 ; Console . WriteLine (countWays (n)); } } //Driver Code Ends JavaScript function countWays (n) { // Base cases if (n === 0 || n === 1) return 1 ; return countWays (n - 1) + countWays (n - 2); } //Driver Code Starts const n = 4 ; console . log (countWays (n)); //Driver Code Ends Output 5 Time Complexity: O(2 n), as we are making two recursive calls for each stair. Auxiliary Space: O(n), as we are using recursive stack space. [Better Approach - 1] Using Top-Down DP (Memoization) – O(n) Time and O(n) Space In this approach, we notice that the same subproblems are being solved multiple times. For example, countWays(n) calls countWays(n-1) and countWays(n-2), and countWays(n-1) again calls countWays(n-2) and countWays(n-3). Here, countWays(n-2) is computed more than once. To avoid this redundant work, we can store the results of subproblems in a dp array and reuse them whenever needed. Here is the visualization of overlapping subproblems for n=4. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int countWaysRec (int n , vector < int >& dp) { // Base cases if (n == 0 || n == 1) return 1 ; // if the result for this subproblem is // already computed then return it if (dp [n] != -1) return dp [n]; return dp [n] = countWaysRec (n - 1 , dp) + countWaysRec (n - 2 , dp); } int countWays (int n) { // dp array to store the results vector < int > dp (n + 1 , -1); return countWaysRec (n , dp); } //Driver Code Starts int main () { int n = 4 ; cout << countWays (n); return 0 ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> #include <stdlib.h> //Driver Code Ends int countWaysRec (int n , int dp []) { // Base cases if (n == 0 || n == 1) return 1 ; // if the result for this subproblem is // already computed then return it if (dp [n] != -1) return dp [n]; return dp [n] = countWaysRec (n - 1 , dp) + countWaysRec (n - 2 , dp); } int countWays (int n) { // dp array to store the results int * dp = (int *) malloc ((n + 1) * sizeof (int)); for (int i = 0 ; i <= n ; i ++) dp [i] = -1 ; int result = countWaysRec (n , dp); free (dp); return result ; } //Driver Code Starts int main () { int n = 4 ; printf ("%d " , countWays (n)); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; class GFG { //Driver Code Ends static int countWaysRec (int n , int [] dp) { // Base cases if (n == 0 || n == 1) return 1 ; // if the result for this subproblem is // already computed then return it if (dp [n] != - 1) return dp [n] ; return dp [n] = countWaysRec (n - 1 , dp) + countWaysRec (n - 2 , dp); } static int countWays (int n) { // dp array to store the results int [] dp = new int [n + 1] ; Arrays . fill (dp , - 1); return countWaysRec (n , dp); } //Driver Code Starts public static void main (String [] args) { int n = 4 ; System . out . println (countWays (n)); } } //Driver Code Ends Python def countWaysRec (n , dp): # Base cases if n == 0 or n == 1 : return 1 # if the result for this subproblem is # already computed then return it if dp [n] != - 1 : return dp [n] dp [n] = countWaysRec (n - 1 , dp) + countWaysRec (n - 2 , dp) return dp [n] def countWays (n): # dp array to store the results dp = [- 1] * (n + 1) return countWaysRec (n , dp) #Driver Code Starts if __name__ == "__main__" : n = 4 print (countWays (n)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int countWaysRec (int n , int [] dp) { // Base cases if (n == 0 || n == 1) return 1 ; // if the result for this subproblem is // already computed then return it if (dp [n] != - 1) return dp [n]; return dp [n] = countWaysRec (n - 1 , dp) + countWaysRec (n - 2 , dp); ; } static int countWays (int n) { // dp array to store the results int [] dp = new int [n + 1]; Array . Fill (dp , - 1); return countWaysRec (n , dp); } //Driver Code Starts static void Main () { int n = 4 ; Console . WriteLine (countWays (n)); } } //Driver Code Ends JavaScript function countWaysRec (n , dp) { // Base cases if (n === 0 || n === 1) return 1 ; // if the result for this subproblem is // already computed then return it if (dp [n] !== - 1) return dp [n]; dp [n] = countWaysRec (n - 1 , dp) + countWaysRec (n - 2 , dp); return dp [n]; } function countWays (n) { // dp array to store the results let dp = Array (n + 1). fill (- 1); return countWaysRec (n , dp); } //Driver Code Starts const n = 4 ; console . log (countWays (n)); //Driver Code Ends Output 5 Time Complexity : O(n), as we compute each subproblem only once using memoization. Auxiliary Space : O(n), due to the memoization array and the recursive stack space(both take linear space only). [Better Approach - 2] Using Bottom-Up DP (Tabulation) – O(n) Time and O(n) Space In this approach, we are iteratively calculating the answers from the ground up, beginning with the smallest subproblems - the first two stairs. Using these base values, we then find the answers for subsequent stairs one by one. Each new value is computed using the results of the previous two (which have already been calculated) and stored for further calculations. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int countWays (int n) { vector < int > dp (n + 1 , 0); // Base cases dp [0] = 1 ; dp [1] = 1 ; for (int i = 2 ; i <= n ; i ++) dp [i] = dp [i - 1] + dp [i - 2]; return dp [n]; } //Driver Code Starts int main () { int n = 4 ; cout << countWays (n); return 0 ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> //Driver Code Ends int countWays (int n) { int dp [n + 1]; // Base cases dp [0] = 1 ; dp [1] = 1 ; for (int i = 2 ; i <= n ; i ++) dp [i] = dp [i - 1] + dp [i - 2]; return dp [n]; } //Driver Code Starts int main () { int n = 4 ; printf ("%d " , countWays (n)); return 0 ; } //Driver Code Ends Java //Driver Code Starts class GFG { //Driver Code Ends static int countWays (int n) { int [] dp = new int [n + 1] ; // Base cases dp [0] = 1 ; dp [1] = 1 ; for (int i = 2 ; i <= n ; i ++) dp [i] = dp [i - 1] + dp [i - 2] ; return dp [n] ; } //Driver Code Starts public static void main (String [] args) { int n = 4 ; System . out . println (countWays (n)); } } //Driver Code Ends Python def countWays (n): dp = [0] * (n + 1) # Base cases dp [0] = 1 dp [1] = 1 for i in range (2 , n + 1): dp [i] = dp [i - 1] + dp [i - 2]; return dp [n] // Driver Code Starts n = 4 print (countWays (n)) // Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static int countWays (int n) { int [] dp = new int [n + 1]; // Base cases dp [0] = 1 ; dp [1] = 1 ; for (int i = 2 ; i <= n ; i ++) dp [i] = dp [i - 1] + dp [i - 2]; return dp [n]; } //Driver Code Starts static void Main () { int n = 4 ; Console . WriteLine (countWays (n)); } } //Driver Code Ends JavaScript function countWays (n) { const dp = new Array (n + 1). fill (0); // Base cases dp [0] = 1 ; dp [1] = 1 ; for (let i = 2 ; i <= n ; i ++) dp [i] = dp [i - 1] + dp [i - 2]; return dp [n]; } //Driver Code Starts const n = 4 ; console . log (countWays (n)); //Driver Code Ends Output 5 [Space Optimised] Using Space Optimized DP – O(n) Time and O(1) Space In this approach, we observe that in order to calculate value for a particular state, we only need answers for previous two states. Hence, we store the results of the previous two states, and as we move forward, we update them to compute the next values. This means keeping track of the last two states and using them to calculate the next one. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int countWays (int n) { // to store values of last two states int prev1 = 1 ; int prev2 = 1 ; for (int i = 2 ; i <= n ; i ++) { int curr = prev1 + prev2 ; prev2 = prev1 ; prev1 = curr ; } // In last iteration final value // of curr is stored in prev. return prev1 ; } //Driver Code Starts int main () { int n = 4 ; cout << countWays (n); return 0 ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> //Driver Code Ends int countWays (int n) { // to store values of last two states int prev1 = 1 ; int prev2 = 1 ; for (int i = 2 ; i <= n ; i ++) { int curr = prev1 + prev2 ; prev2 = prev1 ; prev1 = curr ; } // In last iteration final value // of curr is stored in prev. return prev1 ; } //Driver Code Starts int main () { int n = 4 ; printf ("%d " , countWays (n)); return 0 ; } //Driver Code Ends Java //Driver Code Starts class GfG { //Driver Code Ends static int countWays (int n) { // to store values of last two states int prev1 = 1 ; int prev2 = 1 ; for (int i = 2 ; i <= n ; i ++) { int curr = prev1 + prev2 ; prev2 = prev1 ; prev1 = curr ; } // In last iteration final value // of curr is stored in prev. return prev1 ; } //Driver Code Starts public static void main (String [] args) { int n = 4 ; System . out . println (countWays (n)); } } //Driver Code Ends Python def countWays (n): # to store values of last two states prev1 = 1 prev2 = 1 for i in range (2 , n + 1): curr = prev1 + prev2 prev2 = prev1 prev1 = curr # In last iteration final value # of curr is stored in prev. return prev1 #Driver Code Starts n = 4 print (countWays (n)) #Driver Code Ends C# //Driver Code Starts using System ; class GfG { //Driver Code Ends static int countWays (int n) { // to store values of last two states int prev1 = 1 ; int prev2 = 1 ; for (int i = 2 ; i <= n ; i ++) { int curr = prev1 + prev2 ; prev2 = prev1 ; prev1 = curr ; } // In last iteration final value // of curr is stored in prev. return prev1 ; } //Driver Code Starts static void Main () { int n = 4 ; Console . WriteLine (countWays (n)); } } //Driver Code Ends JavaScript function countWays (n) { // to store values of last two states let prev1 = 1 ; let prev2 = 1 ; for (let i = 2 ; i <= n ; i ++) { let curr = prev1 + prev2 ; prev2 = prev1 ; prev1 = curr ; } // In last iteration final value // of curr is stored in prev. return prev1 ; } //Driver Code Starts const n = 4 ; console . log (countWays (n)); //Driver Code Ends Output 5 [Expected Approach] Using Matrix Exponentiation – O(logn) Time and O(1) Space In this approach, we observe that the problem is similar to Fibonacci series. The traditional way is to repeatedly add two previous numbers using loop or recursive call. But, with matrix exponentiation , we can calculate Fibonacci numbers much faster by working with matrices. There's a special matrix (transformation matrix) that represents how Fibonacci numbers work. To know more about how to calculate Fibonacci number using Matrix Exponentiation, refer to this post . C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends void multiply (vector < vector < int >>& a , vector < vector < int >>& b) { vector < vector < int >> res (2 , vector < int > (2)); // Matrix Multiplication res [0][0] = a [0][0] * b [0][0] + a [0][1] * b [1][0]; res [0][1] = a [0][0] * b [0][1] + a [0][1] * b [1][1]; res [1][0] = a [1][0] * b [0][0] + a [1][1] * b [1][0]; res [1][1] = a [1][0] * b [0][1] + a [1][1] * b [1][1]; // Copy the result back to the first matrix a [0][0] = res [0][0]; a [0][1] = res [0][1]; a [1][0] = res [1][0]; a [1][1] = res [1][1]; } // Function to find (m[][] ^ expo) vector < vector < int >> power (vector < vector < int >>& m , int expo) { // Initialize result with identity matrix vector < vector < int >> res = { { 1 , 0 }, { 0 , 1 } }; while (expo) { if (expo & 1) multiply (res , m); multiply (m , m); expo >>= 1 ; } return res ; } int countWays (int n) { // base case if (n == 0 || n == 1) return 1 ; vector < vector < int >> m = { { 1 , 1 }, { 1 , 0 } }; // Matrix initialMatrix = {{f(1), 0}, {f(0), 0}}, where f(0) // and f(1) are first two terms of sequence vector < vector < int >> initialMatrix = { { 1 , 0 }, { 1 , 0 } }; // Multiply matrix m - (n - 1) times vector < vector < int >> res = power (m , n - 1); multiply (res , initialMatrix); return res [0][0]; } //Driver Code Starts int main () { int n = 4 ; cout << countWays (n); return 0 ; } //Driver Code Ends C //Driver Code Starts #include <stdio.h> //Driver Code Ends void multiply (long long a [2][2], long long b [2][2]) { long long res [2][2]; // Matrix Multiplication res [0][0] = a [0][0] * b [0][0] + a [0][1] * b [1][0]; res [0][1] = a [0][0] * b [0][1] + a [0][1] * b [1][1]; res [1][0] = a [1][0] * b [0][0] + a [1][1] * b [1][0]; res [1][1] = a [1][0] * b [0][1] + a [1][1] * b [1][1]; // Copy the result back to the first matrix a [0][0] = res [0][0]; a [0][1] = res [0][1]; a [1][0] = res [1][0]; a [1][1] = res [1][1]; } void power (long long m [2][2], int expo , long long res [2][2]) { // Initialize result with identity matrix res [0][0] = 1 ; res [0][1] = 0 ; res [1][0] = 0 ; res [1][1] = 1 ; while (expo) { if (expo & 1) multiply (res , m); multiply (m , m); expo >>= 1 ; } } int countWays (int n) { // base case if (n == 0 || n == 1) return 1 ; long long m [2][2] = { { 1 , 1 }, { 1 , 0 } }; // Matrix initialMatrix = {{f(1), 0}, {f(0), 0}}, where f(0) // and f(1) are first two terms of sequence long long initialMatrix [2][2] = { { 1 , 0 }, { 1 , 0 } }; // Multiply matrix m (n - 1) times long long res [2][2]; power (m , n - 1 , res); multiply (res , initialMatrix); return res [0][0]; } //Driver Code Starts int main () { int n = 4 ; printf ("%d " , countWays (

n)); return 0 ; } //Driver Code Ends Java //Driver Code Starts class GFG { //Driver Code Ends static void multiply (int [][] a , int [][] b) { int [][] res = new int [2][2] ; // Matrix Multiplication res [0][0] = a [0][0] * b [0][0] + a [0][1] * b [1][0] ; res [0][1] = a [0][0] * b [0][1] + a [0][1] * b [1][1] ; res [1][0] = a [1][0] * b [0][0] + a [1][1] * b [1][0] ; res [1][1] = a [1][0] * b [0][1] + a [1][1] * b [1][1] ; // Copy the result back to the first matrix a [0][0] = res [0][0] ; a [0][1] = res [0][1] ; a [1][0] = res [1][0] ; a [1][1] = res [1][1] ; } static int [][] power (int [][] m , int expo) { // Initialize result with identity matrix int [][] res = { { 1 , 0 }, { 0 , 1 } }; while (expo > 0) { if ((expo & 1) == 1) multiply (res , m); multiply (m , m); expo >>= 1 ; } return res ; } static int countWays (int n) { // base case if (n == 0 || n == 1) return 1 ; int [][] m = { { 1 , 1 }, { 1 , 0 } }; // Matrix initialMatrix = {{f(1), 0}, {f(0), 0}}, where f(0) // and f(1) are first two terms of sequence int [][] initialMatrix = { { 1 , 0 }, { 1 , 0 } }; // Multiply matrix m (n - 1) times int [][] res = power (m , n - 1); multiply (res , initialMatrix); return res [0][0] ; } //Driver Code Starts public static void main (String [] args) { int n = 4 ; System . out . println (countWays (n)); } } //Driver Code Ends Python def multiply (a , b): res = [[0 , 0], [0 , 0]] # Matrix Multiplication res [0][0] = a [0][0] * b [0][0] + a [0][1] * b [1][0] res [0][1] = a [0][0] * b [0][1] + a [0][1] * b [1][1] res [1][0] = a [1][0] * b [0][0] + a [1][1] * b [1][0] res [1][1] = a [1][0] * b [0][1] + a [1][1] * b [1][1] # Copy the result back to the first matrix a [0][0] = res [0][0] a [0][1] = res [0][1] a [1][0] = res [1][0] a [1][1] = res [1][1] def power (m , expo): # Initialize result with identity matrix res = [[1 , 0], [0 , 1]] while expo > 0 : if expo & 1 : multiply (res , m) multiply (m , m) expo >>= 1 return res def countWays (n): # base case if n == 0 or n == 1 : return 1 m = [[1 , 1], [1 , 0]] # Matrix initialMatrix = {{f(1), 0}, {f(0), 0}}, where f(0) # and f(1) are first two terms of sequence initialMatrix = [[1 , 0], [1 , 0]] # Multiply matrix m - (n - 1) times res = power (m , n - 1) multiply (res , initialMatrix) # Return the final result as an integer return res [0][0] #Driver Code Starts if __name__ == "__main__" : n = 4 print (countWays (n)) #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static void multiply (int [,] a , int [,] b) { int [,] res = new int [2 , 2]; // Matrix Multiplication res [0 , 0] = a [0 , 0] * b [0 , 0] + a [0 , 1] * b [1 , 0]; res [0 , 1] = a [0 , 0] * b [0 , 1] + a [0 , 1] * b [1 , 1]; res [1 , 0] = a [1 , 0] * b [0 , 0] + a [1 , 1] * b [1 , 0]; res [1 , 1] = a [1 , 0] * b [0 , 1] + a [1 , 1] * b [1 , 1]; // Copy the result back to the first matrix a [0 , 0] = res [0 , 0]; a [0 , 1] = res [0 , 1]; a [1 , 0] = res [1 , 0]; a [1 , 1] = res [1 , 1]; } static int [,] power (int [,] m , int expo) { // Initialize result with identity matrix int [,] res = { { 1 , 0 }, { 0 , 1 } }; while (expo > 0) { if ((expo & 1) == 1) multiply (res , m); multiply (m , m); expo >>= 1 ; } return res ; } static int countWays (int n) { // base case if (n == 0 || n == 1) return 1 ; int [,] m = { { 1 , 1 }, { 1 , 0 } }; // Matrix initialMatrix = {{f(1), 0}, {f(0), 0}}, where f(0) // and f(1) are first two terms of sequence int [,] initialMatrix = { { 1 , 0 }, { 1 , 0 } }; // Multiply matrix m - (n - 1) times int [,] res = power (m , n - 1); multiply (res , initialMatrix); return res [0 , 0]; } //Driver Code Starts static void Main () { int n = 4 ; Console . WriteLine (countWays (n)); } } //Driver Code Ends JavaScript function multiply (a , b) { const res = [[0 , 0], [0 , 0]]; // Matrix Multiplication res [0][0] = a [0][0] * b [0][0] + a [0][1] * b [1][0]; res [0][1] = a [0][0] * b [0][1] + a [0][1] * b [1][1]; res [1][0] = a [1][0] * b [0][0] + a [1][1] * b [1][0]; res [1][1] = a [1][0] * b [0][1] + a [1][1] * b [1][1]; // Copy the result back to the first matrix a [0][0] = res [0][0]; a [0][1] = res [0][1]; a [1][0] = res [1][0]; a [1][1] = res [1][1]; } function power (m , expo) { // Initialize result with identity matrix const res = [[1 , 0], [0 , 1]]; while (expo > 0) { if (expo & 1) multiply (res , m); multiply (m , m); expo >>= 1 ; } return res ; } function countWays (n) { // base case if (n === 0 || n === 1) return 1 ; const m = [[1 , 1], [1 , 0]]; // Matrix initialMatrix = {{f(1), 0}, {f(0), 0}}, where f(0) // and f(1) are first two terms of sequence const initialMatrix = [[1 , 0], [1 , 0]]; // Multiply matrix m - (n - 1) times const res = power (m , n - 1); multiply (res , initialMatrix); return res [0][0]; } //Driver Code Starts // Driver code const n = 4 ; console . log (countWays (n)); //Driver Code Ends Output 5 Comment Article Tags: Article Tags: Dynamic Programming Mathematical DSA Amazon Morgan Stanley Samsung Snapdeal Accolite Intel Linkedin sliding-window + 7 More