Count Unique Paths in a Grid - GeeksforGeeks

Source: https://www.geeksforgeeks.org/count-possible-paths-top-left-bottom-right-nxm-matrix/

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 Count Unique Paths in a Grid Last Updated : 21 Jan, 2026 Given a grid of size m x n , determine the number of distinct paths from the top-left corner (0,0) to the bottom-right corner (m-1, n-1). At each step, one can either move down or right. Input: m = 3, n = 3 Output: 6 Explanation: Let the given input 3*3 grid is filled as such: A B C D E F G H I The possible unique paths which exists to reach 'I' from 'A' following above conditions are as follows: ABCFI, ABEHI, ADGHI, ADEFI, ADEHI, ABEFI. Input: m = 2, n = 3 Output: 3 Explanation: Let the given input 2*3 grid is filled as such: A B C D E F The possible unique paths which exists to reach 'F' from 'A' following above conditions are as follows: ABCF, ADEF and ABEF. Input: m = 1, n = 4 Output: 1 Explanation: Let the given input 1*4 grid is filled as such: A B C D The only possible unique path is ABCD. Try it on GfG Practice Table of Content [Naive Approach] Using Recursion – O(2^(m+n)) Time and O(m+n) Space [Better Approach - 1] Using Top-Down DP (Memoization) - O(m*n) Time and O(m*n) Space [Better Approach - 2] Using Bottom-Up DP (Tabulation) – O(m * n) Time and O(m * n) Space [Space Optimised] Using Space Optimized DP - O(n*m) Time and O(n) Space [Expected Approach] Using Combinatorics – O(n) Time and O(1) Space The main idea is that the number of unique paths to cell (i, j) equals the sum of paths from the left cell (i, j-1) and the upper cell (i-1, j). [Naive Approach] Using Recursion – O(2^(m+n)) Time and O(m+n) Space The idea is to use recursion to explore all possible paths from the top-left cell to the bottom-right cell. At each step, we can move either down or right until we reach the destination. The base case occurs when we move outside the grid (invalid) or when we reach the last cell (valid path). By recursively exploring both choices, we count the total number of valid paths. C++ #include <iostream> using namespace std ; int countPaths (int i , int j , int m , int n) { // Base case - reached bottom-right cell if (i == m - 1 && j == n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Move down or right return countPaths (i + 1 , j , m , n) + countPaths (i , j + 1 , m , n); } int uniquePaths (int m , int n) { return countPaths (0 , 0 , m , n); } int main () { int m = 3 , n = 3 ; cout << uniquePaths (m , n); return 0 ; } Java class GfG { static int countPaths (int i , int j , int m , int n) { // Base case - reached bottom-right cell if (i == m - 1 && j == n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Move down or right return countPaths (i + 1 , j , m , n) + countPaths (i , j + 1 , m , n); } static int uniquePaths (int m , int n) { return countPaths (0 , 0 , m , n); } public static void main (String [] args) { int m = 3 , n = 3 ; System . out . println (uniquePaths (m , n)); } } Python def countPaths (i , j , m , n): # Base case - reached bottom-right cell if i == m - 1 and j == n - 1 : return 1 # Out of bounds if i >= m or j >= n : return 0 # Move down or right return countPaths (i + 1 , j , m , n) + countPaths (i , j + 1 , m , n) def uniquePaths (m , n): return countPaths (0 , 0 , m , n) if __name__ == "__main__" : m , n = 3 , 3 print (uniquePaths (m , n)) C# using System ; class Solution { static int countPaths (int i , int j , int m , int n) { // Base case - reached bottom-right cell if (i == m - 1 && j == n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Move down or right return countPaths (i + 1 , j , m , n) + countPaths (i , j + 1 , m , n); } static int uniquePaths (int m , int n) { return countPaths (0 , 0 , m , n); } static void Main () { int m = 3 , n = 3 ; Console . WriteLine (uniquePaths (m , n)); } } JavaScript function countPaths (i , j , m , n) { // Base case - reached bottom-right cell if (i === m - 1 && j === n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Move down or right return countPaths (i + 1 , j , m , n) + countPaths (i , j + 1 , m , n); } function uniquePaths (m , n) { return countPaths (0 , 0 , m , n); } //Driver Code let m = 3 , n = 3 ; console . log (uniquePaths (m , n)); Output 6 [Better Approach - 1] Using Top-Down DP (Memoization) - O(m*n) Time and O(m*n) Space In the recursive approach, the

same subproblems (cell positions) are computed multiple times. To optimize, we store the result of each (i, j) state in a DP table. At each cell (i, j), the number of unique paths is the sum of paths from the cell below (i+1, j) and the cell to the right (i, j+1). By reusing previously computed results, we avoid redundant work and achieve efficient computation. C++ #include <iostream> #include <vector> using namespace std ; int countPaths (int i , int j , int m , int n , vector < vector < int >> & dp) { // Base case: reached bottom-right cell if (i == m - 1 && j == n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Memoized result if (dp [i][j] != -1) return dp [i][j]; // Move down and right return dp [i][j] = countPaths (i + 1 , j , m , n , dp) + countPaths (i , j + 1 , m , n , dp); } int uniquePaths (int m , int n) { vector < vector < int >> dp (m , vector < int > (n , -1)); return countPaths (0 , 0 , m , n , dp); } int main () { int m = 3 , n = 3 ; cout << uniquePaths (m , n); return 0 ; } Java class GFG { static int countPaths (int i , int j , int m , int n , int [][] dp) { // Base case: reached bottom-right cell if (i == m - 1 && j == n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Memoized result if (dp [i][j] != - 1) return dp [i][j] ; // Move down and right return dp [i][j] = countPaths (i + 1 , j , m , n , dp) + countPaths (i , j + 1 , m , n , dp); } static int uniquePaths (int m , int n) { int [][] dp = new int [m][n] ; for (int i = 0 ; i < m ; i ++) for (int j = 0 ; j < n ; j ++) dp [i][j] = - 1 ; return countPaths (0 , 0 , m , n , dp); } public static void main (String [] args) { int m = 3 , n = 3 ; System . out . println (uniquePaths (m , n)); } } Python def countPaths (i , j , m , n , dp): # Base case: reached bottom-right cell if i == m - 1 and j == n - 1 : return 1 # Out of bounds if i >= m or j >= n : return 0 # Memoized result if dp [i][j] != - 1 : return dp [i][j] # Move down and right dp [i][j] = countPaths (i + 1 , j , m , n , dp) \ + countPaths (i , j + 1 , m , n , dp) return dp [i][j] def uniquePaths (m , n): dp = [[- 1 for _ in range (n)] for _ in range (m)] return countPaths (0 , 0 , m , n , dp) if __name__ == "__main__" : m , n = 3 , 3 print (uniquePaths (m , n)) C# using System ; class GFG { static int countPaths (int i , int j , int m , int n , int [,] dp) { // Base case: reached bottom-right cell if (i == m - 1 && j == n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Memoized result if (dp [i , j] != - 1) return dp [i , j]; // Move down and right return dp [i , j] = countPaths (i + 1 , j , m , n , dp) + countPaths (i , j + 1 , m , n , dp); } static int uniquePaths (int m , int n) { int [,] dp = new int [m , n]; for (int i = 0 ; i < m ; i ++) for (int j = 0 ; j < n ; j ++) dp [i , j] = - 1 ; return countPaths (0 , 0 , m , n , dp); } static void Main () { int m = 3 , n = 3 ; Console . WriteLine (uniquePaths (m , n)); } } JavaScript function countPaths (i , j , m , n , dp) { // Base case: reached bottom-right cell if (i === m - 1 && j === n - 1) return 1 ; // Out of bounds if (i >= m || j >= n) return 0 ; // Memoized result if (dp [i][j] !== - 1) return dp [i][j]; // Move down and right dp [i][j] = countPaths (i + 1 , j , m , n , dp) + countPaths (i , j + 1 , m , n , dp); return dp [i][j]; } function uniquePaths (m , n) { let dp = Array . from ({ length : m }, () => Array (n). fill (- 1)); return countPaths (0 , 0 , m , n , dp); } //Driver Code let m = 3 , n = 3 ; console . log (uniquePaths (m , n)); Output 6 [Better Approach - 2] Using Bottom-Up DP (Tabulation) – O(m * n) Time and O(m * n) Space In this approach, we iteratively build a DP table where dp[i][j] stores the number of ways to reach cell (i, j) from (0, 0). The first row and first column each have only one path since movement is restricted to right or down. Each cell's value is the sum of its top and left cells. This leads to the recurrence relation dp[i][j] = dp[i - 1][j] + dp[i][j - 1]. C++ #include <iostream> #include <vector> using namespace std ; int uniquePaths (int m , int n) { // Count of paths to reach any cell in // first column and row is 1 vector < vector < int >> dp (m , vector < int > (n , 1)); for (int i = 1 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { dp [i][j] = dp [i - 1][j] + dp [i][j - 1]; } } return dp [m - 1][n - 1]; } int main () { int m = 3 , n = 3 ; cout << uniquePaths (m , n); return 0 ; } Java class GFG { static int uniquePaths (int m , int n) { // Count of paths to reach any cell in // first column and row is 1 int [][] dp = new int [m][n] ; for (int i = 0 ; i < m ; i ++) dp [i][0] = 1 ; for (int j = 0 ; j < n ; j ++) dp [0][j] = 1 ; for (int i = 1 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { dp [i][j] = dp [i - 1][j] + dp [i][j - 1] ; } } return dp [m - 1][n - 1] ; } public static void main (String [] args) { int m = 3 , n = 3 ; System . out . println (uniquePaths (m , n)); } } Python def uniquePaths (m , n): # Count of paths to reach any cell in # first column and row is 1 dp = [[1 for _ in range (n)] for _ in range (m)] for i in range (1 , m): for j in range (1 , n): dp [i][j] = dp [i - 1][j] + dp [i][j - 1] return dp [m - 1][n - 1] if __name__ == "__main__" : m , n = 3 , 3 print (uniquePaths (m , n)) C# using System ; class GFG { static int uniquePaths (int m , int n) { // Count of paths to reach any cell in // first column and row is 1 int [,] dp = new int [m , n]; for (int i = 0 ; i < m ; i ++) dp [i , 0] = 1 ; for (int j = 0 ; j < n ; j ++) dp [0 , j] = 1 ; for (int i = 1 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { dp [i , j] = dp [i - 1 , j] + dp [i , j - 1]; } } return dp [m - 1 , n - 1]; } static void Main () { int m = 3 , n = 3 ; Console . WriteLine (uniquePaths (m , n)); } } JavaScript function uniquePaths (m , n) { // Count of paths to reach any cell in // first column and row is 1 let dp = Array . from ({ length : m }, () => Array (n). fill (1)); for (let i = 1 ; i < m ; i ++) { for (let j = 1 ; j < n ; j ++) { dp [i][j] = dp [i - 1][j] + dp [i][j - 1]; } } return dp [m - 1][n - 1]; } let m = 3 , n = 3 ; console . log (uniquePaths (m , n)); Output 6 [Space Optimised] Using Space Optimized DP - O(n*m)

Time and O(n) Space We can further optimize space by observing that each cell depends only on the value from the same column in the previous row and the previous column in the current row. Therefore, instead of maintaining a full 2D DP table, we use a single 1D array to store path counts for the current row. The array is initialized such that the first column always has exactly one way to be reached. While iterating row by row, each cell is updated as the sum of paths from the top and to its left. C++ #include <iostream> using namespace std ; int uniquePaths (int m , int n) { // dp[j] stores paths to current cell in the row int dp [n] = { 1 }; // Only one way to reach first column dp [0] = 1 ; for (int i = 0 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { // Paths from top (dp[j]) + left (dp[j-1]) dp [j] += dp [j - 1]; } } // Paths to bottom-right cell return dp [n - 1]; } int main () { int res = uniquePaths (3 , 3); cout << res << endl ; } Java class GFG { static int uniquePaths (int m , int n) { // dp[j] stores paths to current cell in the row int [] dp = new int [n] ; // Only one way to reach first column dp [0] = 1 ; for (int i = 0 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { // Paths from top (dp[j]) + left (dp[j-1]) dp [j] += dp [j - 1] ; } } // Paths to bottom-right cell return dp [n - 1] ; } public static void main (String [] args) { int res = uniquePaths (3 , 3); System . out . println (res); } } Python def uniquePaths (m , n): # dp[j] stores paths to current cell in the row dp = [0] * n # Only one way to reach first column dp [0] = 1 for i in range (m): for j in range (1 , n): # Paths from top (dp[j]) + left (dp[j-1]) dp [j] += dp [j - 1] # Paths to bottom-right cell return dp [n - 1] if __name__ == "__main__" : res = uniquePaths (3 , 3) print (res) C# using System ; class GFG { static int uniquePaths (int m , int n) { // dp[j] stores paths to current cell in the row int [] dp = new int [n]; // Only one way to reach first column dp [0] = 1 ; for (int i = 0 ; i < m ; i ++) { for (int j = 1 ; j < n ; j ++) { // Paths from top (dp[j]) + left (dp[j-1]) dp [j] += dp [j - 1]; } } // Paths to bottom-right cell return dp [n - 1]; } static void Main () { int res = uniquePaths (3 , 3); Console . WriteLine (res); } } JavaScript function uniquePaths (m , n) { // dp[j] stores paths to current cell in the row let dp = new Array (n). fill (0); // Only one way to reach first column dp [0] = 1 ; for (let i = 0 ; i < m ; i ++) { for (let j = 1 ; j < n ; j ++) { // Paths from top (dp[j]) + left (dp[j-1]) dp [j] += dp [j - 1]; } } // Paths to bottom-right cell return dp [n - 1]; } let res = uniquePaths (3 , 3); console . log (res); Output 6 [Expected Approach] Using Combinatorics – O(n) Time and O(1) Space Instead of visiting each cell, we can solve the problem with maths using combinatorics by observing that every valid path from the top-left to the bottom-right cell consists of a fixed number of moves. To reach cell (n-1, m-1) from (0, 0): We must make exactly (n – 1) downward moves. We must make exactly (m – 1) rightward moves. Thus, the total number of moves is: (n – 1) + (m – 1) = n + m – 2. Each unique path corresponds to a unique arrangement of these moves. So, the total number of distinct paths is the number of ways to choose positions for the downward (or rightward) moves among all moves. We use C(n + m – 2, n – 1) or C(n + m – 2, m – 1) since choosing positions for down moves or right moves gives the same count due to C(n, r) = C(n, n – r). C++ #include <iostream> using namespace std ; int uniquePaths (int m , int n) { long long path = 1 ; int total = m + n - 2 ; // Since, C(n, r) = C(n, n – r) int r = min (m - 1 , n - 1); for (int i = 1 ; i <= r ; i ++) { path = path * (total - r + i) / i ; } return (int) path ; } int main () { int res = uniquePaths (3 , 3); cout << res << endl ; return 0 ; } Java class GFG { static int uniquePaths (int m , int n) { long path = 1 ; int total = m + n - 2 ; // Since, C(n, r) = C(n, n – r) int r = Math . min (m - 1 , n - 1); for (int i = 1 ; i <= r ; i ++) { path = path * (total - r + i) / i ; } return (int) path ; } public static void main (String [] args) { int res = uniquePaths (3 , 3); System . out . println (res); } } Python def uniquePaths (m , n): path = 1 total = m + n - 2 # Since, C(n, r) = C(n, n – r) r = min (m - 1 , n - 1) for i in range (1 , r + 1): path = path * (total - r + i) // i return path res = uniquePaths (3 , 3) print (res) C# using System ; class GFG { static int uniquePaths (int m , int n) { long path = 1 ; int total = m + n - 2 ; // Since, C(n, r) = C(n, n – r) int r = Math . Min (m - 1 , n - 1); for (int i = 1 ; i <= r ; i ++) { path = path * (total - r + i) / i ; } return (int) path ; } static void Main () { int res = uniquePaths (3 , 3); Console . WriteLine (res); } } JavaScript function uniquePaths (m , n) { let path = 1 ; const total = m + n - 2 ; // Since, C(n, r) = C(n, n – r) const r = Math . min (m - 1 , n - 1); for (let i = 1 ; i <= r ; i ++) { path = path * (total - r + i) / i ; } return path ; } //Driver Code const res = uniquePaths (3 , 3); console . log (res); Output 6 Comment Article Tags: Article Tags: Dynamic Programming Mathematical Matrix Recursion DSA Microsoft Amazon Walmart Paytm Linkedin + 6 More