[](https://leetcode.com/archit91/)

✅ [C++/Python] 5 Simple Solutions w/ Explanation | Optimization from Brute-Force to DP to Math

[archit91](https://leetcode.com/archit91/)



28477

45396

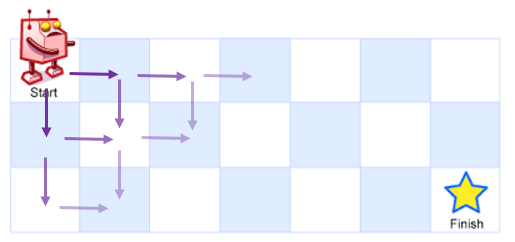
Nov 17, 2021

We are given a m x n grid where we start at cell (0, 0) (top-left) and are required to move to the cell (m-1, n-1) (bottom-right). We can only move to the right or to the bottom. We need to return total unique paths from start to end using these moves.

❌ ***Solution - I (Brute-Force) [TLE]***

Let's start with brute-force solution. For a path to be unique, at atleast 1 of move must differ at some cell within that path.

* At each cell we can either move down or move right.
* Choosing either of these moves could lead us to an unique path
* So we consider both of these moves.
* If the series of moves leads to a cell outside the grid's boundary, we can return 0 denoting no valid path was found.
* If the series of moves leads us to the target cell (m-1, n-1), we return 1 denoting we found a valid unique path from start to end.



**C++**

class Solution {

public:

int uniquePaths(int m, int n, int i = 0, int j = 0) {

if(i >= m || j >= n) return 0; // reached out of bounds - invalid

if(i == m-1 && j == n-1) return 1; // reached the destination - valid solution

return uniquePaths(m, n, i+1, j) + uniquePaths(m, n, i, j+1); // try both down and right

}

};

**Python**

class Solution:

def uniquePaths(self, m, n, i=0, j=0):

if i >= m or j >= n: return 0

if i == m-1 and j == n-1: return 1

return self.uniquePaths(m, n, i+1, j) + self.uniquePaths(m, n, i, j+1)

***Time Complexity :*** **O(2m+n)**, where m and n are the given input dimensions of the grid  
***Space Complexity :*** **O(m+n)**, required by implicit recursive stack

✔️ ***Solution - II (Dynamic Programming - Memoization)***

The above solution had a lot of redundant calculations. There are many cells which we reach multiple times and calculate the answer for it over and over again. However, the number of unique paths from a given cell (i,j) to the end cell is always fixed. So, we dont need to calculate and repeat the same process for a given cell multiple times. We can just store (or memoize) the result calculated for cell (i, j) and use that result in the future whenever required.

Thus, here we use a 2d array dp, where dp[i][j] denote the number of unique paths from cell (i, j) to the end cell (m-1, n-1). Once we get an answer for cell (i, j), we store the result in dp[i][j] and reuse it instead of recalculating it.

class Solution {

public:

int dp[101][101]{};

int uniquePaths(int m, int n, int i = 0, int j = 0) {

if(i >= m || j >= n) return 0;

if(i == m-1 && j == n-1) return 1;

if(dp[i][j]) return dp[i][j];

return dp[i][j] = uniquePaths(m, n, i+1, j) + uniquePaths(m, n, i, j+1);

}

};

A more generalized solution should be as follows -

class Solution {

public:

int uniquePaths(int m, int n) {

vector<vector<int>> dp(m, vector<int>(n));

return dfs(dp, 0, 0);

}

int dfs(vector<vector<int>>& dp, int i, int j) {

if(i >= size(dp) || j >= size(dp[0])) return 0; // out of bounds - invalid

if(i == size(dp)-1 && j == size(dp[0])-1) return 1; // reached end - valid path

if(dp[i][j]) return dp[i][j]; // directly return if already calculated

return dp[i][j] = dfs(dp, i+1, j) + dfs(dp, i, j+1); // store the result in dp[i][j] and then return

}

};

**Python**

class Solution:

def uniquePaths(self, m, n):

@cache

def dfs(i, j):

if i >= m or j >= n: return 0

if i == m-1 and j == n-1: return 1

return dfs(i+1, j) + dfs(i, j+1)

return dfs(0, 0)

***Time Complexity :*** **O(m\*n)**, the answer to each of cell is calculated only once and memoized. There are m\*n cells in total and thus this process takes O(m\*n) time.  
***Space Complexity :*** **O(m\*n)**, required to maintain dp.

✔️ ***Solution - III (Dynamic Programming - Tabulation)***

We can also convert the above appraoch to an iterative version. Here, we will solve it in bottom-up manner by iteratively calculating the number of unique paths to reach cell (i, j) starting from (0, 0) where 0 <= i <= m-1 and 0 <= j <= n-1. We will again use dynamic programming here using a dp matrix where dp[i][j] will denote the number of unique paths from cell (0, 0) the cell (i, j). (Note this differs from memoization appraoch where dp[i][j] denoted number of unique paths from cell (i, j) to the cell (m-1,n-1))

In this case, we first establish some base conditions first.

* We start at cell (0, 0), so dp[0][0] = 1.
* Since we can only move right or down, there is only one way to reach a cell (i, 0) or (0, j). Thus, we also initialize dp[i][0] = 1 and dp[0][j]=1.
* For every other cell (i, j) (where 1 <= i <= m-1 and 1 <= j <= n-1), we can reach here either from the top cell (i-1, j) or the left cell (i, j-1). So the result for number of unique paths to arrive at (i, j) is the summation of both, i.e, dp[i][j] = dp[i-1][j] + dp[i][j-1].

**C++**

class Solution {

public:

int uniquePaths(int m, int n) {

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]; // sum of unique paths ending at adjacent top and left cells

return dp[m-1][n-1]; // return unique paths ending at cell (m-1, n-1)

}

};

**Python**

class Solution:

def uniquePaths(self, m, n):

dp = [[1]\*n for i in range(m)]

for i, j in product(range(1, m), range(1, n)):

dp[i][j] = dp[i-1][j] + dp[i][j-1]

return dp[-1][-1]

***Time Complexity :*** **O(m\*n)**, we are computing dp values for each of the m\*n cells from the previous cells value. Thus, the total number of iterations performed is requires a time of O(m\*n).  
***Space Complexity :*** **O(m\*n)**, required to maintain the dp matrix

✔️ ***Solution - IV (Space Optimized Dynamic Programming)***

In the above solution, we can observe that to compute the dp matrix, we are only ever using the cells from previous row and the current row. So, we don't really need to maintain the entire m x n matrix of dp. We can optimize the space usage by only keeping the current and previous rows.

A common way in dp problems to optimize space from 2d dp is just to convert the dp matrix from m x n grid to 2 x n grid denoting the values for current and previous row. We can just overwrite the previous row and use the current row as the previous row for next iteration. We can simply alternate between these rows using the & (AND) operator as can be seen below -

**C++**

class Solution {

public:

int uniquePaths(int m, int n) {

vector<vector<int>> dp(2, vector<int>(n,1));

for(int i = 1; i < m; i++)

for(int j = 1; j < n; j++)

dp[i & 1][j] = dp[(i-1) & 1][j] + dp[i & 1][j-1]; // <- & used to alternate between rows

return dp[(m-1) & 1][n-1];

}

};

**Python**

class Solution:

def uniquePaths(self, m, n):

dp = [[1]\*n for i in range(2)]

for i in range(1,m):

for j in range(1,n):

dp[i&1][j] = dp[(i-1)&1][j] + dp[i&1][j-1]

return dp[(m-1)&1][-1]

Or still better yet, in this case, you can use a single vector as well. We are only accessing same column from previous row which can be given by dp[j] and previous column of current row which can be given by dp[j-1]. So the above code can be further simplfied to (Credits - [@zayne-siew](https://leetcode.com/zayne-siew)) -

**C++**

class Solution {

public:

int uniquePaths(int m, int n) {

vector<int> dp(n, 1);

for(int i = 1; i < m; i++)

for(int j = 1; j < n; j++)

dp[j] += dp[j-1];

return dp[n-1];

}

};

**Python**

class Solution:

def uniquePaths(self, m, n):

dp = [1]\*n

for \_, j in product(range(1, m), range(1, n)):

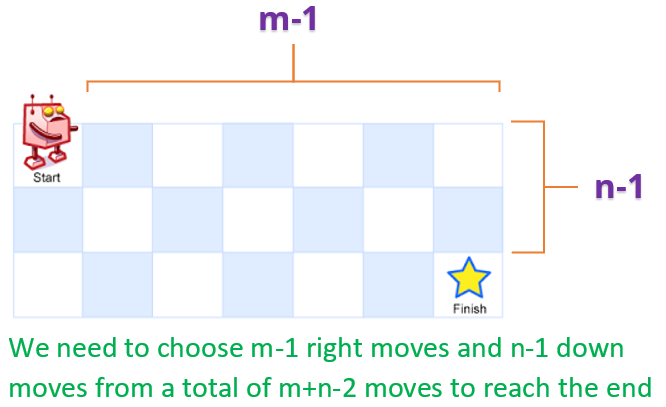
dp[j] += dp[j-1]

return dp[-1]

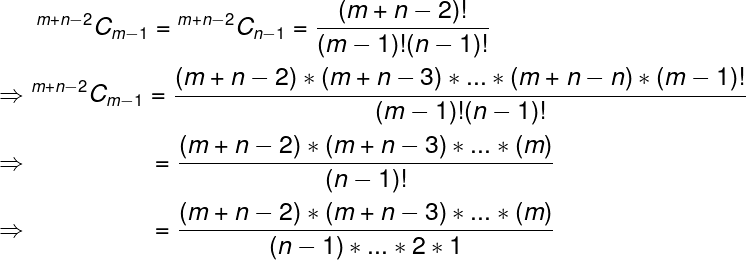
***Time Complexity :*** **O(m\*n)**, for computing dp values for each of the m\*n cells.  
***Space Complexity :*** **O(n)**, required to maintain dp. We are only keeping two rows of length n giving space complexity of O(n).  
There's a small change that can allow us to optimize the space complexity down to O(min(m, n)).  
*Comment below if you can figure it out :)*

✔️ ***Solution - V (Math)***

This problem can be modelled as a math combinatorics problem.



* We start at (0, 0) cell and move to (m-1, n-1) cell.
* We need to make **m-1 down-moves** and **n-1 right-moves** to reach the destination cell.
* Thus, we need to perform a **total number of m+n-2 moves**.
* At each cell along the path, we can choose either the right-move or down-move and we need to find the number of unique combinations of these choices (which eventually leads to unique paths).
* This is nothing but calculating the **number of different ways to choose m-1 down-moves and n-1 right-moves from a total of m+n-2 moves**. Mathematically, this can be represented as -



We could cancel out the (n-1)! as well in the above evaluation. We will do one of those based on min(m,n) to give best time complexity in the solution below.

**C++**

class Solution {

public:

int uniquePaths(int m, int n) {

long ans = 1;

for(int i = m+n-2, j = 1; i >= max(m, n); i--, j++)

ans = (ans \* i) / j;

return ans;

}

};

**Python**

class Solution:

def uniquePaths(self, m, n):

return factorial(m+n-2) // factorial(m-1) // factorial(n-1)

***Time Complexity :*** **O(min(m,n))** for C++, and **O(m+n)** for Python. We could do it in O(min(m,n)) for python as well using technique used in C++.  
***Space Complexity :*** **O(1)**