<https://leetcode.com/problems/unique-paths/>

There is a robot on an m x n grid. The robot is initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to 2 \* 109.

**Example 1:**



Input: m = 3, n = 7

Output: 28

**Example 2:**

Input: m = 3, n = 2

Output: 3

Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down

**Constraints:**

* 1 <= m, n <= 100

**Attempt 1: 2023-06-06**

**Solution 1: Native DFS (10 min, TLE 38/63, m = 19, n = 13)**

class Solution {

public int uniquePaths(int m, int n) {

return helper(0, 0, m, n);

}

private int helper(int x, int y, int m, int n) {

if(x >= m || y >= n) {

return 0;

}

if(x == m - 1 && y == n - 1) {

return 1;

}

return helper(x + 1, y, m, n) + helper(x, y + 1, m, n);

}

}

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

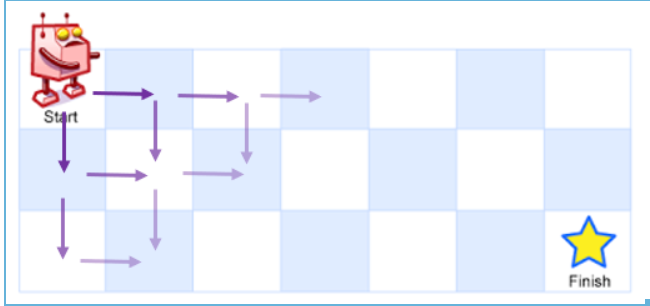
**Refer to**

<https://leetcode.com/problems/unique-paths/solutions/1581998/c-python-5-simple-solutions-w-explanation-optimization-from-brute-force-to-dp-to-math/>

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

Let's start with brute-force solution. For a path to be unique, at least 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

}

};

***Time Complexity :*** **O(2^(m+n))**, where m and n are the given input dimensions of the grid

***Space Complexity :*** **O(m+n)**, required by implicit recursive stack

**Solution 2: DFS + Memoization (10 min)**

class Solution {

public int uniquePaths(int m, int n) {

Integer[][] memo = new Integer[m][n];

return helper(0, 0, m, n, memo);

}

private int helper(int x, int y, int m, int n, Integer[][] memo) {

if(x >= m || y >= n) {

return 0;

}

if(memo[x][y] != null) {

return memo[x][y];

}

if(x == m - 1 && y == n - 1) {

return 1;

}

return memo[x][y] = helper(x + 1, y, m, n, memo) + helper(x, y + 1, m, n, memo);

}

}

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.

**Refer to**

<https://leetcode.com/problems/unique-paths/solutions/1581998/c-python-5-simple-solutions-w-explanation-optimization-from-brute-force-to-dp-to-math/>

✔️ ***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 don't 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

}

};

***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 3: DP (10 min)**

class Solution {

public int uniquePaths(int m, int n) {

// dp[i][j] will denote the number of unique paths from

// cell (0, 0) to the cell (i, j). (Note this differs from

// memoization approach where dp[i][j] denoted number of

// unique paths from cell (i, j) to the cell (m-1, n-1))

int[][] dp = new int[m][n];

// We start at cell (0, 0), so dp[0][0] = 1, means number

// of unique paths from cell (0, 0) to cell(i, j) as

// cell(0, 0) is 1 as itself

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

dp[i][0] = 1;

}

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

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].

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];

}

}

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

**Refer to**

<https://leetcode.com/problems/unique-paths/solutions/1581998/c-python-5-simple-solutions-w-explanation-optimization-from-brute-force-to-dp-to-math/>

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

We can also convert the above approach 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) to the cell (i, j). (Note this differs from memoization approach 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)

}

};

***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 4: Space Optimized to two rows DP (10 min)**

class Solution {

public int uniquePaths(int m, int n) {

int[] dp = new int[n];

int[] dpPrev = new int[n];

for(int i = 0; i < n; i++) {

dp[i] = 1;

dpPrev[i] = 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];

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

}

dpPrev = dp;

}

return dpPrev[n - 1];

}

}

Time Complexity: O(M\*N), where M <= 100 is number of rows, N <= 100 is number of columns.

Space Complexity: O(M)

**Refer to**

<https://leetcode.com/problems/unique-paths/solutions/254228/python-3-solutions-bottom-up-dp-math-picture-explained-clean-concise/>

Since we only access 2 states: current state dp and previous state dpPrev, we can reduce the space complexity to O(M).

class Solution:

def uniquePaths(self, m: int, n: int) -> int:

dp, dpPrev = [0] \* n, [0] \* n

for r in range(m):

for c in range(n):

if r == 0 or c == 0:

dp[c] = 1

else:

dp[c] = dpPrev[c] + dp[c-1]

dp, dpPrev = dpPrev, dp

return dpPrev[n-1]

**Complexity**

* Time: O(M\*N), where M <= 100 is number of rows, N <= 100 is number of columns.
* Space: O(M)

**Solution 5: Space Optimized to single row DP (10 min)**

class Solution {

public int uniquePaths(int m, int n) {

int[] dp = new int[n];

for(int i = 0; i < n; i++) {

dp[i] = 1;

}

// Truncate first dimension as row and only needs second

// dimension as column to compute

// dp[i][j] => dp[j], dp[i - 1][j] => dp[j], dp[i][j - 1] => dp[j - 1]

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

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

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

}

}

return dp[n - 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).

**Refer to**

<https://leetcode.com/problems/unique-paths/solutions/1581998/c-python-5-simple-solutions-w-explanation-optimization-from-brute-force-to-dp-to-math/>

✔️ ***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];

}

};

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 simplified 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];

}

};

***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).

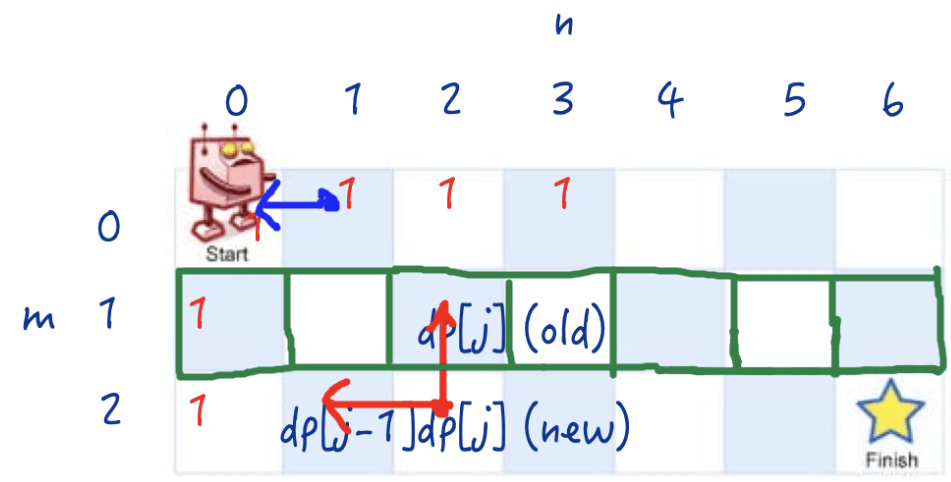
There's a small change that can allow us to optimize the space complexity down to O(min(m, n)).

*Comment below if you can figure it out*

**Refer to**

<https://leetcode.com/problems/unique-paths/solutions/405983/easy-understand-java-solutions-with-explanations-dp-top-down-bottom-up-linear-space/>

Reduce the O(MN) space complexity to O(N) (a row) or O(M)(a column). In terms of a row, we would update dp[j] by its old value plus dp[j - 1]



public int uniquePaths(int m, int n) {

if (m == 0 || n == 0) {

throw new IllegalArgumentException("m or n can't be 0");

}

int[] dp = new int[n]; // row

// init

for (int i = 0; i < n; ++i) dp[i] = 1;

// dp

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

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

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

}

}

return dp[n - 1];

}

**Time:** O(MN)

**Space:** O(N) or O(M)

**Solution 6: Math (? min)**

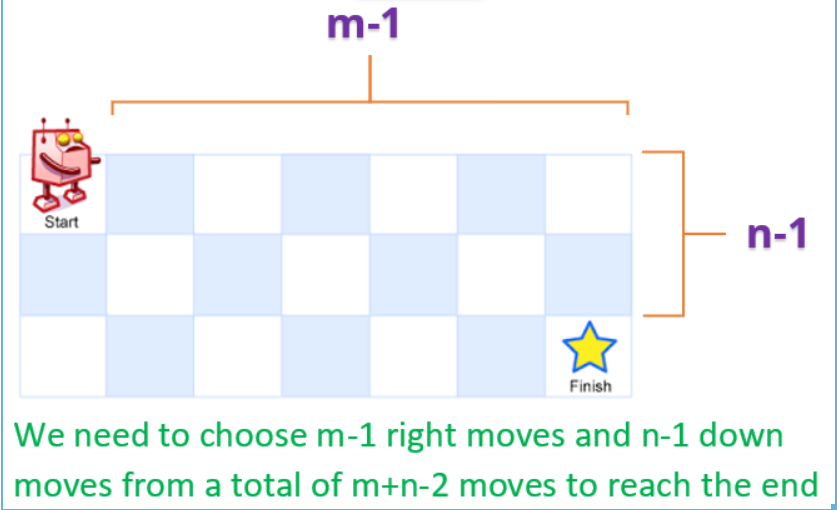
**Just for documentation**

**Refer to**

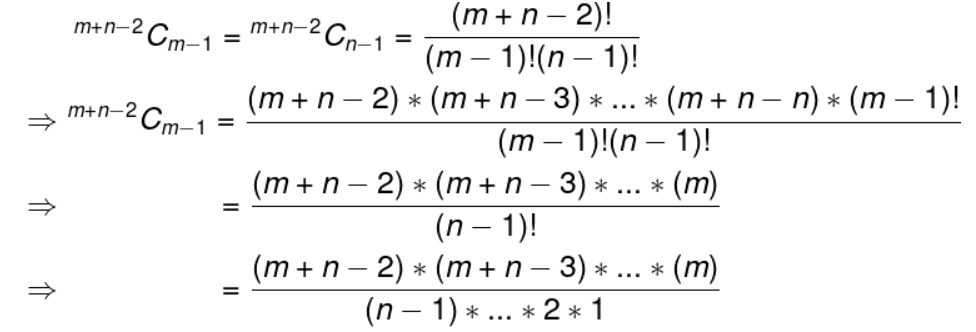
<https://leetcode.com/problems/unique-paths/solutions/1581998/c-python-5-simple-solutions-w-explanation-optimization-from-brute-force-to-dp-to-math/>

✔️ ***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;

}

};

***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)**