**Example 62. Unique Paths**[Report Issue](https://github.com/LeetCode-Feedback/LeetCode-Feedback/issues)

The bottom-up approach for pathing problems is often more intuitive than bottom-up for other types of dynamic programming problems - so this is a good chance for us to practice starting with a bottom-up approach.

In this article, we'll use the framework to solve [Unique Paths](https://leetcode.com/problems/unique-paths/). This problem asks us to find the number of distinct ways to do something, which is a hint that we should consider using dynamic programming, and we discussed how to do so in the previous chapter. Let's get started:

**1. An array that contains the answers to the problem for a given state**

State variables are usually easy to find in pathing problems. Similar to how we need one index (i) for 1D array inputs, with pathing problems on a 2D matrix, we need two indices (row and col) to denote position. Some problems have added constraints that will require additional state variables, but there doesn't seem to be anything of the sort in this problem. Therefore, we will just use two state variables row which represents the current row, and col which represents the current column.

The problem is asking for the number of paths to the final square, so let's have dp[row][col] represent how many paths there are from the start (top-left corner) to the square at (row, col). We will return dp[m - 1][n - 1] where m and n are the number of rows and columns respectively.

**2. A recurrence relation to transition between states**

The problem says that we are allowed to move down or right. That means, if we are at some square, we arrived from either the square above or the square to the left. These two squares are (row - 1, col) and (row, col - 1). Since we can arrive at the current square from either of these squares, the number of ways to get to the current square is the sum of the number of ways to get to these two squares. Either of these may be out of the grid bounds, so we should make sure to check for that. This gives us our simple recurrence relation:

dp[row][col] = dp[row - 1][col] + dp[row][col - 1], where dp[row - 1][col] and dp[row][col - 1] is equal to 0 if out of bounds.

**3. Base cases**

In the previous chapter, when talking about counting DP problems, we said that the base cases need to be set to nonzero values so that the terms in the recurrence relation don't just stay stuck at zero. In this problem, we start in the top-left corner. How many ways are there for us to get to the first square? Only 1 - we start on it. Therefore, our base case is dp[0][0] = 1.

Note: If you have trouble coming up with the recurrence relation, sometimes it helps to come up with the base case(s) first. Then walk through how you would find the result for states that are slightly more complicated than the base case(s), such as dp[0][1], dp[1, 1], and dp[2, 1]. Often, this process of manually solving the problem for simple states can help you understand what the recurrence relation should be.

Bottom-up Implementation

Putting it all together for the final solution:

Here's an animation showing the algorithm in action:

class Solution {

public int uniquePaths(int m, int n) {

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

dp[0][0] = 1; // Base case

for (int row = 0; row < m; row++) {

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

if (row > 0) {

dp[row][col] += dp[row - 1][col];

}

if (col > 0) {

dp[row][col] += dp[row][col - 1];

}

}

}

return dp[m - 1][n - 1];

}

}

Top-down Implementation

As we know, one of the advantages of top-down implementations is that they are usually easier to implement. However, for some, bottom-up dynamic programming is more intuitive when it comes to pathing problems. As such, we implemented the bottom-up approach first in this example. That said, it is rare to convert bottom-up solutions to top-down solutions since the typical flow for dynamic programming problems is to come up with a top-down DP solution first, and then convert it to a bottom-up DP solution, if the bottom-up approach is more efficient. Below, for completeness, we have also included the top-down implementation.

The time and space complexity of both solutions is O(m \cdot n)*O*(*m*⋅*n*). We visit each square only once and do a constant amount of work. Here's a challenge to do on your own: try to use state reduction to improve the space complexity of the bottom-up approach to O(n)*O*(*n*) without modifying the input.

class Solution {

private int[][] memo;

private int dp(int row, int col) {

if (row + col == 0) {

return 1; // Base case

}

int ways = 0;

if (memo[row][col] == 0) {

if (row > 0) {

ways += dp(row - 1, col);

}

if (col > 0) {

ways += dp(row, col - 1);

}

memo[row][col] = ways;

}

return memo[row][col];

}

public int uniquePaths(int m, int n) {

memo = new int[m][n];

return dp(m - 1, n- 1);

}

}