# 62. Unique Paths

Let's look at a classic **2D Dynamic Programming** problem:

We are given an `m x n` (m by n) grid. A robot starts at the **top-left corner** and can only move **right** or **down**. Our task is to determine how many **unique paths** the robot can take to reach the **bottom-right corner**.

## Brute-force Recursion

At first glance, this seems like a basic enumeration problem.

At the point (x, y), the robot has two choices:

- Move right to the position `(x, y + 1)`
- Move down to the position `(x + 1, y)`

So, a natural way to approach it is using recursion to explore all possible paths:

```
def dfs(x, y):
            if x == m - 1 and y == n - 1:
                return 1
            if x >= m or y >= n:
                return 0
            return dfs(x + 1, y) + dfs(x, y + 1)
```

This function calculates the total number of paths from `(x, y)` to the bottom-right by summing the number of paths going down and going right.

The logic is correct. But the performance is poor due to **repeated calculations**.

For example, suppose we reach the cell `(1, 1)` — one direction coming from `(0, 1)` and another direction from `(1, 0)`. In both cases, we need to compute the number of paths from `(1, 1)` to our destination cell. But since we didn't store the result, we'll **recomputing all the paths from `(1, 1)` twice**, even though it's the exact same subproblem.

This is a case of **overlapping subproblems**, and that's exactly where dynamic programming comes in.

## Memoization

So the next step is to think: "If we already counted the number of paths from (1,1) to the end point, can I save it?" Let's try to store results to avoid recomputation.

We introduce a 2D array `memo[i][j]` to store the number of paths from cell `(i, j)` to the most bottom-right. Initially, all values are `-1`, meaning "not computed yet":

```
memo = [[-1] * n for _ in range(m)]
```

Then we update our recursive function:

```
def dfs(x, y):
            if x == m - 1 and y == n - 1:
                return 1
            if x >= m or y >= n:
                return 0
            if memo[x][y] != -1:
                return memo[x][y]
            memo[x][y] = dfs(x + 1, y) + dfs(x, y + 1)
            return memo[x][y]
```

Now, each subproblem is only computed **once**, giving us a time complexity of `O(m * n)` (big O of m times n).

## Tabulation – Bottom-Up DP

So we've optimized the performance using **memoization**. But this approach is still fundamentally based on **recursion and the call stack** — in other words, we’re still solving the problem by calling functions layer by layer, from the top down. Essentially, it's just **"recursion + caching"** to avoid redundant computations.

But what if we **change our perspective**?

Since we already understand the relationship between states — that is, the number of paths to reach a given cell equals the sum of the paths from the cell **above it** and the cell **to its left**. Why not reverse the process?

Instead of using recursion to explore all possible paths, we can **start from the beginning point**, build up from the simplest subproblem, and **fill in the entire DP table step by step**.

This shift in thinking — from recursive memoization to iterative/ˈɪtərətɪv/ table-filling — is exactly what we call **bottom-up dynamic programming**, or **Tabulation**.

Now let's Building the DP Table — there is 4 Key Components.

### 1.State Definition

Because we ultimately want to know: how many ways are there to walk from the starting point to the end point `(m-1,n-1)`. So I want to record the number of paths through each point. This determines the definition of the dp table. Let `dp[i][j]` represent the **number of unique paths** from the starting point `(0, 0)` to the cell `(i, j)`.

```
# dp[i][j] = number of unique paths from (0, 0) to (i, j)
        dp = [[0] * n for _ in range(m)]
```

## 2.State Transition Formula

To reach cell `(i, j)`, the robot must have come from:

- the cell above: `(i - 1, j)` (i minus one, j)
- or the cell to the left: `(i, j - 1)` (i, jminus one)

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

## 3.Initialization

Initialization is also an important part of the solution.

First, we need to define the initial state: `dp[0][0] = 1`, because there is exactly one way to be at the starting point.

Since we're working with a **two-dimensional grid**, and the state depends on **two directions** (from the top and from the left), the initialization for 2D DP is slightly more involved than 1D DP. We need to handle the boundaries carefully.

At each step, the robot has only two possible directions to move: **right** or **down**. That means:

- For the **leftmost column**, the robot can only come **from above**, so every cell in this column has exactly **one path**: `dp[i][0] = 1` (dp of i zero equals one)
- For the **top row**, the robot can only come **from the left**, so every cell in this row also has exactly **one path**: `dp[0][j] = 1` (dp of zero j equals one)

Here’s the corresponding code of initialization:

```
dp[0][0] = 1
        for i in range(1, m):
            dp[i][0] = 1
        for j in range(1, n):
            dp[0][j] = 1
```

### 4.Fill the Table

Now we use nested loops and our state transition formula to fill in the rest of the table:

```
 for i in range(1, m):
            for j in range(1, n):
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
```

### 5.Get the Final Answer

The answer we want is the number of paths to the end point `(m-1, n-1)` (m minus on, n minus one). At the beginning, we define `dp[i][j]` (dp of i j) as the number of paths from the starting point to `(i,j)`, so `dp[m-1][n-1]` is our answer.

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

This is the thinking flow behind solving this classic 2D dynamic programming problem. This pattern — **recursion ➝ memoization ➝ tabulation** — is a great way to gradually understand and master dynamic programming.

Now, let’s hand it over to Timmons, who will walk you through the entire process in even more depth — and help you finally grasp the difference between **top-down** and **bottom-up** approaches.