62. Unique Paths

Medium Math Dynamic Programming Combinatorics

Problem Description

The problem presents a scenario where we have a `m x n` grid, and a robot is positioned at the top-left corner of this grid, which is marked by `grid[0][0]`. The goal for the robot is to reach the bottom-right corner of the grid, designated by `grid[m - 1][n - 1]`. The robot is restricted to move only in two directions, either down or right, at any given point in the time.

The main question is to calculate the total number of unique paths the robot can take to reach from the top-left corner to the bottom-right corner. A path is considered unique if it follows a different sequence of moves. The constraints of the problem ensure that the final answer will be a value that is less than or equal to `2 * 10^9`.

Intuition

To approach this problem, we must recognize that it's a classic example of a combinatorial problem that can be solved using dynamic programming (DP). The key insight for a DP solution is that the number of unique paths to reach a particular cell in the grid is the sum of the unique paths to reach the cell directly above it and the cell to its left. This is because the robot can only move down or right, so any path to a cell must come from one of these two adjacent cells.

A recursive solution following this approach would involve a lot of repeated calculations, so a better approach is to use an iterative DP algorithm, which builds up the solution bottom-up and avoids recomputation.

Specifically, we can track the number of ways to reach each cell in a single row as we iterate across the grid. We initialize a list `f` with a length of `n` (number of columns) and set all values to 1, because there's only one way to reach any cell in the first row by moving right at every step.

Then, as we iterate row by row starting from the second row, we update each cell's value in the list `f` with the sum of its own value (the number of paths coming from the left) and the value of the cell before it in the list (the number of paths coming from above). This is done in-place to make the algorithm more space-efficient.

Finally, after we have processed all the rows, the last cell in the list will contain the total number of unique paths to reach the bottom-right cell of the grid.

The `uniquePaths` function encapsulates this algorithm and returns the number of unique paths as the result.

Solution Approach

The implementation of the solution employs dynamic programming, which is an algorithmic technique used to solve problems by breaking them down into simpler subproblems. It is especially powerful for problems that involve making a sequence of interrelated decisions, like in this scenario where the robot makes a series of moves either down or to the right.

Here's how dynamic programming is applied in this solution:

- The use of a one-dimensional array `f` of size `n` serves as a space-optimized way to store the number of unique paths to each cell in the current row from the start cell. Initially, this list is initialized with all `1`s because in the first row, the robot can only move right, so there's exactly one path to each cell in the row.

```
f = [1] * n
```

- We iterate over each additional row in the grid, starting from the second row (since the first row is already initialized). The number of unique paths to reach a cell in the grid at a position `[i, j]` is equal to the sum of the paths to reach `[i, j - 1]` and `[i - 1, j]` (the cell to its left and the cell above it, respectively). Since we are only interested in the current and previous row at any time, we can simply use the same list `f` to accumulate the count of paths.

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

  In the nested loop, each cell `f[j]` gets updated by adding the value `f[j - 1]` to it, which effectively means the current value of `f[j]` corresponds to the cell above `[i - 1, j]`, and `f[j - 1]` corresponds to the cell to the left `[i, j - 1]`. By the end of the loop, `f[j]` will represent the correct number of paths to the cell `[i, j]`.

- This process is repeated until we have gone through all the rows. As a result, the accumulated number in the last cell of our list `f[-1]` will indicate the number of unique paths to reach the bottom-right corner of the grid.

```
return f[-1]
```

The space complexity of this algorithm is `O(n)`, where `n` is the number of columns in the grid. The time complexity is `O(m * n)` as every cell gets computed exactly once. By using dynamic programming, this solution efficiently calculates the result without redundancy in computation and with minimal space usage.

Example Walkthrough

Let's consider a small grid of size `3 x 4` (i.e., `m = 3` and `n = 4`) to illustrate how the solution approach works. For this example, the goal is to determine the number of unique paths from the top-left corner `grid[0][0]` to the bottom-right corner `grid[2][3]`.

1. We start by initializing our `f` array with `4` elements (since `n = 4`), and each element in `f` is set to `1`, as there is only one way to reach each cell in the top row by moving to the right.

   Initial `f` array: `[1, 1, 1, 1]`

   This corresponds to the paths in the first row of the grid, where the robot can only move right.

2. Then, we proceed to iterate over the rest of the rows in the grid (starting with the second row). For each cell we visit, we update its corresponding entry in `f` with the sum of the paths that can reach this cell from above and left.

   Since we are starting from the second row, `f[j]` contains the number of paths to the cell above `[i - 1, j]`, and `f[j - 1]` contains the number of paths to the cell to the left `[i, j - 1]`. For the first cell in each row (the leftmost column), we don't need to update the value since there is still only one path to reach it by moving down from above.

3. We iterate through the second row and update the array `f`:

   For `j = 1`: `f[1] += f[0]` (2 unique paths to `grid[1][1]`) For `j = 2`: `f[2] += f[1]` (3 unique paths to `grid[1][2]`) For `j = 3`: `f[3] += f[2]` (4 unique paths to `grid[1][3]`)

   Updated `f` array after the second row: `[1, 2, 3, 4]`

4. We repeat this process for the third row:

   For `j = 1`: `f[1] += f[0]` (3 unique paths to `grid[2][1]`) For `j = 2`: `f[2] += f[1]` (6 unique paths to `grid[2][2]`) For `j = 3`: `f[3] += f[2]` (10 unique paths to `grid[2][3]`)

   Final `f` array after the third row: `[1, 3, 6, 10]`

5. After processing all the rows, the value in the last cell of `f`, which is `10`, represents the number of unique paths from the top-left corner to the bottom-right corner of the `3 x 4` grid.

6. Consequently, `f[-1]` equals `10`, so the function `uniquePaths` returns `10` as the final answer.

The solution can be visualized as filling up a table where each cell represents the number of paths leading to it. By the end of the procedure, the bottom-right cell contains the total count of unique paths the robot can take to reach the bottom-right corner of the grid.

Python Solution

```python
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        # Initialize a list that will hold the number of unique paths to each cell
        # in the first row. Initially, there's only one unique path to reach any cell
        # in the first row since the only possible move is to the right.
        path_counts = [1] * n

        # Iterate over the rows of the grid starting from the second row,
        # since the first row has been initialized already.
        for row in range(1, m):
            # Iterate over the columns starting from the second column,
            # since the first column only has one unique path (move down).
            for col in range(1, n):
                # The number of unique paths to reach this cell is the sum of
                # the number of unique paths to reach the cell directly above
                # and the number of unique paths to reach the cell to the left.
                path_counts[col] += path_counts[col - 1]

        # Return the number of unique paths to reach the bottom-right corner of the grid,
        # which is the last element in the path_counts list.
        return path_counts[-1]

```

Java Solution

```java
class Solution {
    public int uniquePaths(int m, int n) {
        // Create an array to store the number of unique paths to each cell in the bottom row.
        int[] pathCounts = new int[n];

        // Initialize the bottom row with 1s since there's only one way to reach each cell in the bottom row
        // when only moving right.
        Arrays.fill(pathCounts, 1);

        // Loop over each cell starting from the second row up to the top row (since the bottom row is already filled).
        for (int row = 1; row < m; ++row) {
            // For each cell in a row, start from the second column since the first column of any row
            // will only have one unique path (i.e., moving down from the cell above).
            for (int col = 1; col < n; ++col) {
                // The number of unique paths to the current cell is the sum of the unique paths to the cell
                // directly above it and to the cell to the left of it.
                pathCounts[col] += pathCounts[col - 1];
            }
        }

        // Return the number of unique paths to the top-right corner of the grid.
        return pathCounts[n - 1];
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to calculate the unique paths in an m x n grid
    int uniquePaths(int m, int n) {
        // Create a vector with size 'n', initialize all elements to 1.
        // The vector represents the number of ways to reach each cell in the grid.
        vector<int> pathCounts(n, 1);

        // Iterate over each row starting from the second one
        for (int row = 1; row < m; ++row) {
            // Iterate over each column starting from the second one
            for (int col = 1; col < n; ++col) {
                // Update the path count for the current cell
                // The number of paths to the current cell is the sum of the paths
                // from the cell directly above and the cell to the left.
                pathCounts[col] += pathCounts[col - 1];
            }
        }

        // Return the number of unique paths to the bottom-right cell
        return pathCounts[n - 1];
    }
};

```

Typescript Solution

```typescript
function uniquePaths(rows: number, columns: number): number {
  // Initialize an array 'dp' to store the number of unique paths to each cell in the first row
  const dp: number[] = Array(columns).fill(1);

  // Iterate over the grid starting from the second row, as the first row is preset to 1s
  for (let row = 1; row < rows; ++row) {
    // Iterate over each column starting from the second column
    for (let col = 1; col < columns; ++col) {
      // Update the dp array for the current cell based on the
      // sum of the cell directly above and the cell to the left
      dp[col] += dp[col - 1];
    }
  }
  // Return the number of unique paths to the bottom-right cell
  return dp[columns - 1];
}

```

Time and Space Complexity

The time complexity of the provided solution is `O(m * n)`, where `m` is the number of rows and `n` is the number of columns. This is because there are two nested loops: the outer loop runs `m - 1` times (as the first row's values are all initialized to 1), and the inner loop runs `n - 1` times for each iteration of the outer loop, resulting in a total of `(m - 1) * (n - 1)` iterations of the inner loop's body. However, since we're only interested in big O notation, this simplifies to `O(m * n)`.

The space complexity of the solution is `O(n)`, as it uses a single list `f` of size `n` to store intermediary results. This list is updated in-place, and no additional space that is dependent on `m` or `n` is used. As a result, the space complexity does not change with `m`, only with `n`.