63. Unique Paths II

Medium Array Dynamic Programming Matrix

Leetcode Link

Problem Description

You are presented with a grid represented by an `m` x `n` integer array called `grid`. This grid is filled with either 0's or 1's, which denote empty spaces and obstacles, respectively. There is a robot situated at the top-left corner of the grid, i.e., at `grid[0][0]`. The goal for the robot is to reach the bottom-right corner of the grid, namely `grid[m - 1][n - 1]`.

The robot can only move in two directions at any given time: either down or to the right. Given these movement restrictions and the presence of obstacles (denoted by 1's), the task is to calculate the number of unique paths the robot can take to reach its destination without traversing any of the obstacles.

To provide a solution, the assumption is made that the number of unique paths will not exceed `2 * 10^9`.

Intuition

When thinking about the robot's journey from the top-left to the bottom-right corner, we can frame the problem using dynamic programming. The approach focuses on the concept that the number of ways to reach a particular cell in the grid is the sum of the ways to reach the cell directly above it and the cell to its immediate left, since the robot can only move down or right.

However, when a cell contains an obstacle, the robot cannot travel through it, which means this cell contributes zero paths to its adjacent cells.

Here's a breakdown of the dynamic programming solution:

1. Start by initializing a 2D array `dp` (same dimensions as `grid`) to store the number of ways to reach each cell.
2. Set the value of `dp[1][0]` (first column) and `dp[0][1]` (first row) to 1 for as long as there are no obstacles. This represents the fact that there is only one way to move along the top row or the leftmost column when there are no obstacles in those positions.
3. Loop through the grid starting from cell `dp[1][1]` and move rightward and downward.
4. For every cell, check if it is not an obstacle. If it isn't, set `dp[i][j]` to the sum of `dp[i - 1][j]` (the number of ways to reach from above) and `dp[i][j - 1]` (the number of ways to reach from the left).
5. If you encounter an obstacle, set the number of ways to 0 because the robot can't pass through obstacles.
6. Continue this process until you reach the bottom-right corner of the grid.
7. The final answer, the number of unique paths to the bottom-right corner avoiding obstacles, will be found in `dp[m-1][n-1]`.

With this approach, we can systematically compute the number of unique paths available to the robot, taking into consideration the positioning of the obstacles.

Solution Approach

The solution approach for the problem is based on dynamic programming, a method for solving complex problems by breaking them down into simpler subproblems. It is a powerful technique for optimization problems like this, where we aim to count all possible unique paths in a grid with obstacles.

1. **Data Structure:** A 2-dimensional list `dp` of size `m` x `n` (where `m` and `n` are the dimensions of the input `grid`) is initialized with zeros. This `dp` list is used to store the number of unique paths to reach each cell (i, j) from the start (0, 0).

2. **Base Cases Initialization:** The robot can only move down or right. Therefore, if there are no obstacles in the first column and first row, there will be only one path to each of those cells — just keep moving right or down respectively.

```
for i in range(m):
    if obstacleGrid[i][0] == 1:
        break
    dp[i][0] = 1

for j in range(n):
    if obstacleGrid[0][j] == 1:
        break
    dp[0][j] = 1
```

   These loops set the base conditions for the first row and column, stopping whenever an obstacle is encountered, since there would be no paths passing through the obstacle.

3. **Dynamic Programming Loop:** Starting from the cell (1, 1), the algorithm iteratively computes the number of paths to each cell, adding the number of paths from the cell directly above and the number of paths from the cell to the left, as long as those cells are not obstacles.

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

   This code is the heart of our dynamic programming approach. It succinctly captures the essence that at each step, the robot could have come from the left cell or the cell above.

4. **Construct the Return Value:** The value at `dp[m-1][n-1]` will be the total number of unique paths from the start to the bottom-right corner. It accounts for all possible paths that the robot could take without hitting an obstacle.

```
return dp[-1][-1]
```

In terms of time and space complexity, this algorithm runs in O(m * n), where `m` and `n` are the dimensions of the input grid, since we have to visit each cell once to calculate the number of paths to it. Space complexity is also O(m * n) due to the additional `dp` list used to store the number of paths to each cell.

Example Walkthrough

Let's go through a small example to illustrate the dynamic programming approach described in the solution.

Consider the following grid `grid` where `0` denotes an empty space and `1` denotes an obstacle:

```
grid = [
  [0, 0, 0],
  [0, 1, 0],
  [0, 0, 0]
]
```

We aim to calculate the number of unique paths from `grid[0][0]` to `grid[2][2]`.

1. **Data Structure Initialization:** Initialize the `dp` array with zeros.

```
dp = [
  [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0]
]
```

2. **Base Cases Initialization:** Populate the first row and first column as per the approach, considering the obstacles.

   After initialization:

```
dp = [
  [1, 1, 1],
  [1, 0, 0],
  [1, 0, 0]
]
```

   The obstacle in `grid[1][1]` means the robot cannot go through it, so we leave `dp[1][1]` as `0`.

3. **Dynamic Programming Loop:** We compute the values of the remaining cells.

   For `dp[1][2]`: Since the cell above it (`dp[1][1]`) is blocked by an obstacle, we only add the paths from the left (`dp[1][1]`), resulting in `dp[1][2] = 1`.

   For `dp[2][1]`: Similar to the previous, we only add the paths from the above (`dp[1][1]` is 0 due to the obstacle) and the left (`dp[2][0]` is 1), resulting in `dp[2][1] = 1`.

   Recalculated `dp` after iteration:

```
dp = [
  [1, 1, 1],
  [1, 0, 1],
  [1, 1, 0]
]
```

   For `dp[2][2]`: We add the values from above (`dp[1][2] = 1`) and the left (`dp[2][1] = 1`), giving us `dp[2][2] = 2`.

   Final `dp` array:

```
dp = [
  [1, 1, 1],
  [1, 0, 1],
  [1, 1, 2]
]
```

4. **Construct the Return Value:** The value at `dp[2][2]` is 2, signifying there are two unique paths from start to destination that avoid the obstacles.

Therefore, for the given grid, the robot has exactly two unique paths to reach the destination from the starting point.

Python Solution

```
class Solution:
    def uniquePathsWithObstacles(self, obstacle_grid: List[List[int]]) -> int:
        # Get the number of rows and columns in the obstacle grid.
        rows, cols = len(obstacle_grid), len(obstacle_grid[0])

        # Initialize the DP (Dynamic Programming) table with zeros.
        dp_table = [[0] * cols for _ in range(rows)]

        # Set the value for the first cell to 1 if it is not an obstacle.
        if obstacle_grid[0][0] == 0:
            dp_table[0][0] = 1

        # Populate the first column of the DP table.
        for i in range(1, rows):
            if obstacle_grid[i][0] == 0:  # If there is no obstacle,
                dp_table[i][0] = dp_table[i-1][0]  # the value from cell above.

        # Populate the first row of the DP table.
        for j in range(1, cols):
            if obstacle_grid[0][j] == 0:  # If there is no obstacle,
                dp_table[0][j] = dp_table[0][j-1]  # the value from cell to the left.

        # Fill in the rest of the DP table.
        for i in range(1, rows):
            for j in range(1, cols):
                if obstacle_grid[i][j] == 0:  # If the current cell is not an obstacle,
                    dp_table[i][j] = dp_table[i-1][j] + dp_table[i][j-1]  # Sum of top and left cells.

        # The bottom-right cell of the DP table will hold the number of unique paths.
        return dp_table[-1][-1]
```

Java Solution

```
class Solution {

    // Function to calculate the unique paths in a grid with obstacles
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        // Get the dimensions of the grid
        int numRows = obstacleGrid.length;
        int numCols = obstacleGrid[0].length;

        // Initialize a DP table with dimensions equivalent to the obstacle grid
        int[][] dp = new int[numRows][numCols];

        // Set up the first column of the DP table. If there is an obstacle,
        // paths beyond that point are not possible, so the loop will break.
        for (int row = 0; row < numRows && obstacleGrid[row][0] == 0; ++row) {
            dp[row][0] = 1;
        }

        // Set up the first row of the DP table. If there is an obstacle,
        // paths beyond that point are not possible, so the loop will break.
        for (int col = 0; col < numCols && obstacleGrid[0][col] == 0; ++col) {
            dp[0][col] = 1;
        }

        // Iterate over the grid starting from cell (1, 1) to calculate the
        // number of unique paths to each cell, considering the obstacles.
        for (int row = 1; row < numRows; ++row) {
            for (int col = 1; col < numCols; ++col) {
                // If the current cell is not an obstacle
                if (obstacleGrid[row][col] == 0) {
                    // Number of paths to current cell is the sum of paths to the
                    // cell above it and the cell to the left of it.
                    dp[row][col] = dp[row - 1][col] + dp[row][col - 1];
                }
                // If the current cell is an obstacle, dp[row][col] remains 0
            }
        }

        // Return the number of unique paths to the bottom-right corner of the grid
        return dp[numRows - 1][numCols - 1];
    }
}
```

C++ Solution

```
class Solution {
public:
    // Method calculates the number of unique paths from top-left to bottom-right in
    // a grid that may have obstacles. An obstacle and space are marked as 1 and 0, respectively, in the grid.
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int numberOfRows = obstacleGrid.size(); // Get the number of rows in the grid.
        int numberOfColumns = obstacleGrid[0].size(); // Get the number of columns in the grid.

        // Create a 2D dp matrix with the same dimensions as obstacleGrid to store the
        // number of ways to reach each cell.
        vector<vector<int>> dp(numberOfRows, vector<int>(numberOfColumns, 0));

        // Initialize the first column of the dp matrix. A cell in the first column can only be reached from
        // the cell above it, so if there's an obstacle in a cell, all cells below it should be 0 as well.
        for (int i = 0; i < numberOfRows && obstacleGrid[i][0] == 0; ++i) {
            dp[i][0] = 1;
        }

        // Similarly, initialize the first row of the dp matrix. A cell in the first row can only be reached from
        // the cell to the left of it, so if there's an obstacle in a cell, all cells to the right of it should be 0.
        for (int j = 0; j < numberOfColumns && obstacleGrid[0][j] == 0; ++j) {
            dp[0][j] = 1;
        }

        // Start from cell (1, 1) and fill in the dp matrix until the bottom-right corner of the grid.
        // The value of dp[i][j] is obtained by adding the values from the cell above (dp[i - 1][j]) and
        // the cell to the left (dp[i][j - 1]).
        for (int i = 1; i < numberOfRows; ++i) {
            for (int j = 1; j < numberOfColumns; ++j) {
                // If there's no obstacle in the current cell, calculate the number of paths.
                if (obstacleGrid[i][j] == 0) {
                    dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
                }
                // If there's an obstacle, the number of paths to the current cell will be 0.
            }
        }

        // The bottom-right cell of the dp matrix contains the number of unique paths
        // from the top-left corner to the bottom-right corner, which we return.
        return dp[numberOfRows - 1][numberOfColumns - 1];
    }
};
```

Typescript Solution

```
// Calculates the number of unique paths on a grid with obstacles.
// Each path moves only rightward or downward at any point, avoiding obstacles.
function uniquePathsWithObstacles(obstacleGrid: number[][]): number {
    // Number of rows and columns in the grid
    const rowCount = obstacleGrid.length;
    const colCount = obstacleGrid[0].length;

    // Initializing a 2D array to store the number of ways to reach each cell
    const dp = Array.from({ length: rowCount }, () => new Array(colCount).fill(0));

    // Filling in the first column, taking obstacles into account
    for (let row = 0; row < rowCount; row++) {
        if (obstacleGrid[row][0] === 1) {
            break; // If there's an obstacle, no path can pass through here
        }
        dp[row][0] = 1; // Without obstacles, there's 1 way to get to each cell in the first column
    }

    // Filling in the first row, taking obstacles into account
    for (let col = 0; col < colCount; col++) {
        if (obstacleGrid[0][col] === 1) {
            break; // If there's an obstacle, no path can pass through here
        }
        dp[0][col] = 1; // Without obstacles, there's 1 way to get to each cell in the first row
    }

    // Calculating paths for the rest of the grid
    for (let row = 1; row < rowCount; row++) {
        for (let col = 1; col < colCount; col++) {
            // If the current cell has an obstacle, skip and continue
            if (obstacleGrid[row][col] === 1) {
                continue;
            }
            // Number of paths to current cell is the sum of paths from the cell above and to the left
            dp[row][col] = dp[row - 1][col] + dp[row][col - 1];
        }
    }

    // Returning the total number of ways to reach the bottom-right corner of the grid
    return dp[rowCount - 1][colCount - 1];
}
```

Time and Space Complexity

The time complexity of the provided code is `O(mn)` where `m` is the number of rows and `n` is the number of columns in the `obstacleGrid`. This is because the code contains two nested loops that iterate over each cell in the `m` x `n` grid exactly once, and the operations inside the loop are constant time operations.

The space complexity of the provided code is also `O(mn)` since it uses a 2D list `dp` with the same dimensions as the `obstacleGrid` to store the number of unique paths to each cell.