### 63. Unique Paths II Medium Array Dynamic Programming

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

Matrix

## When thinking about the robot's journey from the top-left to the bottom-right corner, we can frame the problem using dynamic

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.

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

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.

- 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

2. Set the value of dp[i][0] (first column) and dp[0][j] (first row) to 1 for as long as there are no obstacles. This represents the

- 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

# 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.

1 for i in range(m):

break

dp[0][j] = 1

if obstacleGrid[i][0] == 1:

would be no paths passing through the obstacle.

unique paths in a grid with obstacles.

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.

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

These loops set the base conditions for the first row and column, stopping whenever an obstacle is encountered, since there

are not obstacles. 1 for i in range(1, m): for j in range(1, n): if obstacleGrid[i][j] == 0:

This code is the heart of our dynamic programming approach. It succinctly captures the essence that at each step, the robot

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

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

could have come from the left cell or the cell above.

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

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.

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

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:

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

[0, 0, 0],

[0, 0, 0], [0, 0, 0]

1 dp = [

[0, 0, 0]

1 grid = [

Example Walkthrough

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

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

```
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.
```

Recalculated dp after iteration: [1, 1, 1],

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.

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

 $1 \, dp = [$ [1, 1, 1], [1, 0, 1], [1, 1, 2]

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]),

for j in range(1, cols): 20 if obstacle\_grid[0][j] == 0: # If there is no obstacle, 21 dp\_table[0][j] = dp\_table[0][j-1] # Use value from cell to the left. 22 23 # Fill in the rest of the DP table. 24 for i in range(1, rows):

dp\_table[i][j] = dp\_table[i-1][j] + dp\_table[i][j-1] # Sum of top and left cells.

if obstacle\_grid[i][j] == 0: # If the current cell is not an obstacle,

# The bottom-right cell of the DP table will hold the number of unique paths.

2 [1, 1, 1], 3 [1, 0, 0], [1, 0, 0]

1 dp = [

After initialization:

```
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] \text{ is 1}), resulting in dp[2][1] = 1.
```

[1, 0, 1],

[1, 1, 0]

Final dp array:

avoid the obstacles.

Python Solution

class Solution:

15

16

17

18

19

25

26

27

28

29

30

31

Java Solution

resulting in dp[1][2] = 1.

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

def uniquePathsWithObstacles(self, obstacle\_grid: List[List[int]]) -> int:

# Get the number of rows and columns in the obstacle grid.

# Initialize the DP (Dynamic Programming) table with zeros.

if obstacle\_grid[i][0] == 0: # If there is no obstacle,

dp\_table[i][0] = dp\_table[i-1][0] # Use value from cell above.

rows, cols = len(obstacle\_grid), len(obstacle\_grid[0])

dp\_table = [[0] \* cols for \_ in range(rows)]

# Populate the first row of the DP table.

for j in range(1, cols):

return dp[numRows - 1][numCols - 1];

// Method calculates the number of unique paths from top-left to bottom-right in

int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {

// Filling in the first column, taking obstacles into account

for (let row = 0; row < rowCount; row++) {</pre>

// a grid that may have obstacles. An obstacle and space are marked as 1 and 0, respectively, in the grid.

return dp\_table[-1][-1]

```
# 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
11
12
            # Populate the first column of the DP table.
13
14
            for i in range(1, rows):
```

```
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;
6
            int numCols = obstacleGrid[0].length;
8
           // Initialize a DP table with dimensions equivalent to the obstacle grid
10
           int[][] dp = new int[numRows][numCols];
11
12
           // 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) {</pre>
14
15
                dp[row][0] = 1;
16
17
           // Set up the first row of the DP table. If there is an obstacle,
18
           // paths beyond that point are not possible, so the loop will break.
19
20
            for (int col = 0; col < numCols && obstacleGrid[0][col] == 0; ++col) {</pre>
                dp[0][col] = 1;
21
22
23
24
           // Iterate over the grid starting from cell (1, 1) to calculate the
25
           // number of unique paths to each cell, considering the obstacles.
26
            for (int row = 1; row < numRows; ++row) {</pre>
27
                for (int col = 1; col < numCols; ++col) {</pre>
                    // If the current cell is not an obstacle
28
29
                    if (obstacleGrid[row][col] == 0) {
30
                        // Number of paths to current cell is the sum of paths to the
31
                        // cell above it and the cell to the left of it.
32
                        dp[row][col] = dp[row - 1][col] + dp[row][col - 1];
33
                   // If the current cell is an obstacle, dp[row][col] remains 0
34
35
36
37
38
           // Return the number of unique paths to the bottom-right corner of the grid
```

### int numberOfRows = obstacleGrid.size(); // Get the number of rows in the grid. 6 int numberOfColumns = obstacleGrid[0].size(); // Get the number of columns in the grid. 8 // Create a 2D dp matrix with the same dimensions as obstacleGrid to store the 9 10 // number of ways to reach each cell.

11

12

2 public:

C++ Solution

1 class Solution {

39

40

42

41 }

```
11
             vector<vector<int>> dp(number0fRows, vector<int>(number0fColumns, 0));
 12
 13
             // Initialize the first column of the dp matrix. A cell in the first column can only be reached from
 14
             // 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) {</pre>
 15
 16
                 dp[i][0] = 1;
 17
 18
 19
             // Similarly, initialize the first row of the dp matrix. A cell in the first row can only be reached from
 20
             // 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.
 21
             for (int j = 0; j < numberOfColumns && obstacleGrid[0][j] == 0; ++j) {</pre>
 22
                 dp[0][j] = 1;
 23
 24
 25
             // Start from cell (1, 1) and fill in the dp matrix until the bottom-right corner of the grid.
 26
             // The value of dp[i][j] is obtained by adding the values from the cell above (dp[i-1][j]) and
 27
             // the cell to the left (dp[i][j - 1]).
 28
             for (int i = 1; i < numberOfRows; ++i) {</pre>
 29
                 for (int j = 1; j < numberOfColumns; ++j) {</pre>
 30
                     // If there's no obstacle in the current cell, calculate the number of paths.
 31
                     if (obstacleGrid[i][j] == 0) {
 32
                         dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
 33
 34
                     // If there's an obstacle, the number of paths to the current cell will be 0.
 35
 36
 37
 38
             // The bottom-right cell of the dp matrix contains the number of unique paths
 39
             // from the top-left corner to the bottom-right corner, which we return.
             return dp[number0fRows - 1][number0fColumns - 1];
 40
 41
 42 };
 43
Typescript Solution
  1 // Calculates the number of unique paths on a grid with obstacles.
  2 // 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;
  6
      // Initializing a 2D array to store the number of ways to reach each cell
  8
       const dp = Array.from({ length: rowCount }, () => new Array(colCount).fill(0));
  9
 10
```

```
13
       if (obstacleGrid[row][0] === 1) {
14
          break; // If there's an obstacle, no path can pass through here
15
16
       dp[row][0] = 1; // Without obstacles, there's 1 way to get to each cell in the first column
17
18
     // Filling in the first row, taking obstacles into account
19
20
     for (let col = 0; col < colCount; col++) {</pre>
21
       if (obstacleGrid[0][col] === 1) {
22
          break; // If there's an obstacle, no path can pass through here
23
24
       dp[0][col] = 1; // Without obstacles, there's 1 way to get to each cell in the first row
25
26
27
     // Calculating paths for the rest of the grid
28
     for (let row = 1; row < rowCount; row++) {</pre>
        for (let col = 1; col < colCount; col++) {</pre>
29
         // If the current cell has an obstacle, skip and continue
30
31
         if (obstacleGrid[row][col] === 1) {
32
            continue;
33
34
35
          // Number of paths to current cell is the sum of paths from the cell above and to the left
36
          dp[row][col] = dp[row - 1][col] + dp[row][col - 1];
37
38
39
     // Returning the total number of ways to reach the bottom-right corner of the grid
40
     return dp[rowCount - 1][colCount - 1];
41
42
43
```

Time and Space Complexity

operations inside the loop are constant time operations.

The time complexity of the provided code is O(m\*n) where m is the number of rows and n is the number of columns in the

The space complexity of the provided code is also 0(m\*n) since it uses a 2D list dp with the same dimensions as the obstacleGrid to store the number of unique paths to each cell.

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