63. Unique Paths II

Medium 5462 396 Add to List Share

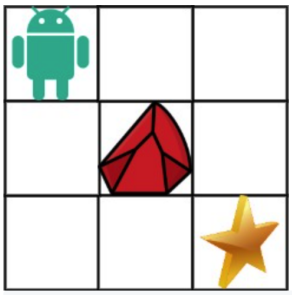
You are given an `m x n` integer array `grid`. There is a robot initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m-1][n-1]`). The robot can only move either down or right at any point in time.

An obstacle and space are marked as `1` or `0` respectively in `grid`. A path that the robot takes cannot include **any** square that is an obstacle.

Return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The testcases are generated so that the answer will be less than or equal to `2 * 10^9`.

Example 1:



```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

```cpp
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int row_num = obstacleGrid.size();
        int col_num = obstacleGrid[0].size();
        if (obstacleGrid[0][0]) return 0;
        int dp[row_num + 1][col_num + 1];
        memset(dp, 0, sizeof(dp));
        dp[1][1] = 1;
        for (int row = 1; row < row_num + 1; row++) {
            for (int col = 1; col < col_num + 1; col++) {
                if (row == 1 && col == 1) continue;
                if (obstacleGrid[row - 1][col - 1]) dp[row][col] = 0;
                else dp[row][col] = dp[row - 1][col] + dp[row][col - 1];
            }
        }
        return dp[row_num][col_num];
    }
};
```

Proof by Loop invariant

Loop invariant: `dp[i][j]` is always the unique path number from cell `(1, 1)` to `(i, j)` cell where i is the row number count from the top, and j is the column number count from left to right, i and j starts from 1

If the given Loop invariant is proof correct, we could just output `dp[row_num][col_num]` of the target cell

Initialization:

When i = 1, j = 1. if there's a obstacle in cell `(1, 1)` i.e. `obstacleGrid[0][0] == 1` our algorithm just return 0. because there's no way to move to any cell from the start cell. Otherwise, `dp[1][1] = 1` because it is obvious that only one path to the start cell. `i = 0` or `j = 0` is outer bound of the grid. They are virtual cells which can never be moved to. So `dp[i][0]` and `dp[0][j]` are all initialize to 0. which is still correct for its definition

Maintenance:

When the loop goes to calculate dp[i][j], it has 2 cases:

1. case 1: `obstacleGrid[i - 1][j - 1] == 1`. which means current cell is blocked, and there would be no path to a blocked cell, `dp[i][j] = 0`, is correct.
2. case 2: `obstacleGrid[i - 1][j - 1] == 0`. which means current cell is open. Because the robot can only move down or right, so it is either from top side or left side. Which means the number of unique path to current cell is the sum of unique path to top side cell and unique path to its left side cell. Therefore `dp[i][j] = dp[i - 1][j] + dp[i][j - 1]` still holds

Termination:

When `i = row_num, j = col_num`, the algorithem ends. And the invariant still holds for last cell (the target)

Time complexity:

`O(row_num * col_num)` because we need to traverse the whole grid once.