



## Unique Paths II

Solution

★★★★★

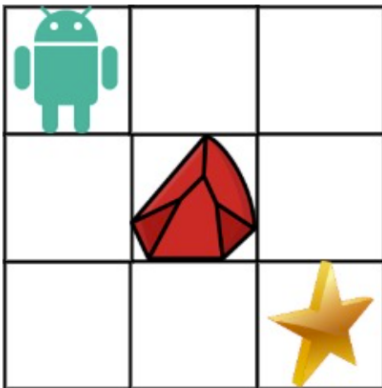
You are given an  $m \times n$  integer array `grid`. There is a robot initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m-1][n-1]`). The robot can only move either down or right at any point in time.

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:



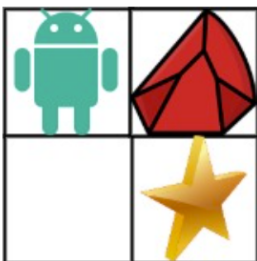
Input: `obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]`

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

### Example 2:



```
Input: obstacleGrid = [[0,1],[0,0]]
Output: 1
```

#### Constraints:

- `m == obstacleGrid.length`
- `n == obstacleGrid[i].length`
- `1 <= m, n <= 100`
- `obstacleGrid[i][j]` is `0` or `1`.

#### 💡 Hide Hint #1 ▲

The robot can only move either down or right. Hence any cell in the first row can only be reached from the cell left to it. However, if any cell has an obstacle, you don't let that cell contribute to any path. So, for the first row, the number of ways will simply be

```
if obstacleGrid[i][j] is not an obstacle
    obstacleGrid[i][j] = obstacleGrid[i][j - 1]
else
    obstacleGrid[i][j] = 0
```

You can do a similar processing for finding out the number of ways of reaching the cells in the first column.

#### 💡 Hide Hint #2 ▲

For any other cell, we can find out the number of ways of reaching it, by making use of the number of ways of reaching the cell directly above it and the cell to the left of it in the grid. This is because these are the only two directions from which the robot can come to the current cell.

#### 💡 Hide Hint #3 ▲

Since we are making use of pre-computed values along the iteration, this becomes a dynamic programming problem.

```
if obstacleGrid[i][j] is not an obstacle
    obstacleGrid[i][j] = obstacleGrid[i][j - 1] + obstacleGrid[i - 1][j]
else
    obstacleGrid[i][j] = 0
```

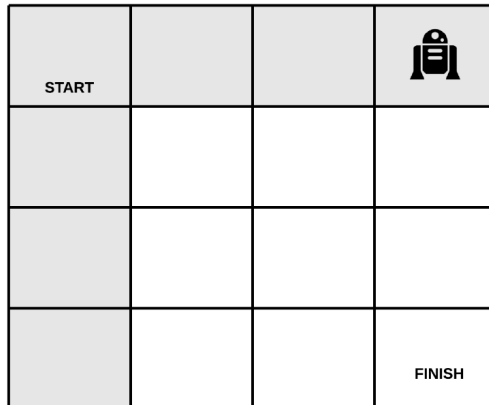
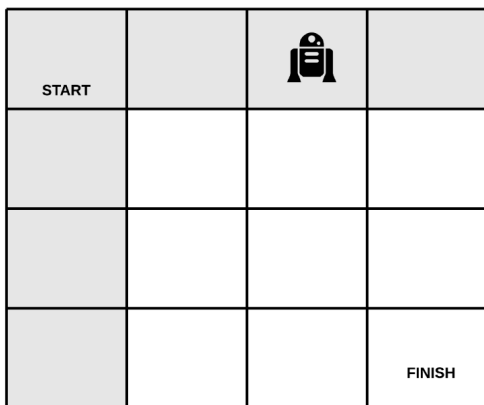
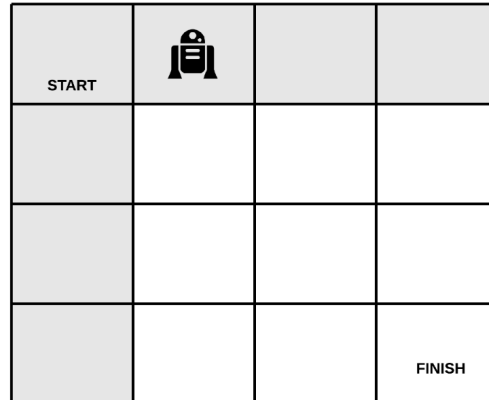
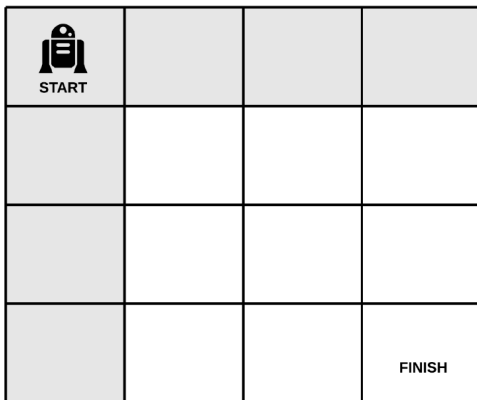
## Solution

---


## Approach 1: Dynamic Programming


### Intuition


The robot can only move either down or right. Hence any cell in the first row can only be reached from the cell left to it.




And, any cell in the first column can only be reached from the cell above it.

 START			
			FINISH

START			
			
			FINISH

START			
			
			FINISH

START			
			FINISH

---

For any other cell in the grid, we can reach it either from the cell to left of it or the cell above it.

If any cell has an obstacle, we won't let that cell contribute to any path.

We will be iterating the array from left-to-right and top-to-bottom. Thus, before reaching any cell we would have the number of ways of reaching the predecessor cells. This is what makes it a **Dynamic Programming** problem. We will be using the **obstacleGrid** array as the DP array thus not utilizing any additional space.

**Note:** As per the question, cell with an obstacle has a value **1**. We would use this value to make sure if a cell needs to be included in the path or not. After that we can use the same cell to store the number of ways to reach that cell.

### Algorithm

1. If the first cell i.e. `obstacleGrid[0,0]` contains **1**, this means there is an obstacle in the first cell. Hence the robot won't be able to make any move and we would return the number of ways as **0**.
2. Otherwise, if `obstacleGrid[0,0]` has a **0** originally we set it to **1** and move ahead.
3. Iterate the first row. If a cell originally contains a **1**, this means the current cell has an obstacle and shouldn't contribute to any path. Hence, set the value of that cell to **0**. Otherwise, set it to the value of previous cell i.e.  
`obstacleGrid[i,j] = obstacleGrid[i,j-1]`
4. Iterate the first column. If a cell originally contains a **1**, this means the current cell has an obstacle and shouldn't contribute to any path. Hence, set the value of that cell to **0**. Otherwise, set it to the value of previous cell i.e.  
`obstacleGrid[i,j] = obstacleGrid[i-1,j]`
5. Now, iterate through the array starting from cell `obstacleGrid[1,1]`. If a cell originally doesn't contain any obstacle then the number of ways of reaching that cell would be the sum of number of ways of reaching the cell above it and number of ways of reaching the cell to the left of it.

```
obstacleGrid[i,j] = obstacleGrid[i-1,j] + obstacleGrid[i,j-1]
```

6. If a cell contains an obstacle set it to **0** and continue. This is done to make sure it doesn't contribute to any other path.

Following is the animation to explain the algorithm's steps:

### Video

1/18

Code:

```
class Solution {  
  
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {  
  
        int R = obstacleGrid.length;  
  
        int C = obstacleGrid[0].length;
```



```
// If the starting cell has an obstacle, then simply return as there would be
```

```
// no paths to the destination.
```

```
if (obstacleGrid[0][0] == 1) {
```

```
    return 0;
```

```
}
```

```
// Number of ways of reaching the starting cell = 1.
```

```
obstacleGrid[0][0] = 1;
```

```
// Filling the values for the first column
```

```
for (int i = 1; i < R; i++) {
```

```
    obstacleGrid[i][0] = (obstacleGrid[i][0] == 0 && obstacleGrid[i - 1][0] == 1) ? 1 : 0;
```

```
}
```

```
// Filling the values for the first row
```

```
for (int i = 1; i < C; i++) {
```

```
    obstacleGrid[0][i] = (obstacleGrid[0][i] == 0 && obstacleGrid[0][i - 1] == 1) ? 1 : 0;
```

```
}
```

```

// Starting from cell(1,1) fill up the values

// No. of ways of reaching cell[i][j] = cell[i - 1][j] + cell[i][j - 1]

// i.e. From above and left.

for (int i = 1; i < R; i++) {

    for (int j = 1; j < C; j++) {

        if (obstacleGrid[i][j] == 0) {

            obstacleGrid[i][j] = obstacleGrid[i - 1][j] + obstacleGrid[i][j - 1];

        } else {

            obstacleGrid[i][j] = 0;

        }

    }

}

// Return value stored in rightmost bottommost cell. That is the destination.

return obstacleGrid[R - 1][C - 1];

}

}

```

#### Complexity Analysis

- Time Complexity:  $O(M \times N)$ . The rectangular grid given to us is of size  $M \times N$  and we process each cell just once.
- Space Complexity:  $O(1)$ . We are utilizing the `obstacleGrid` as the DP array. Hence, no extra space.

Comment:

Another approach(with recursion):

```
class Solution(object):

    def uniquePathsWithObstacles(self, obstacleGrid):

        """
        brute force, bottom-up recursively with memorization

        - intuitively go through all the path with i+1 OR j+1

        - count the path which reaches to the destination coordinate (m, n)

        - cache the count of the coordinates which we have calculated before

        - if the current grid, grid[i][j], is blocked, tell its parent that
        this way is blocked by return 0

        - sum up all the coordinates' count

        Time    O(row*col) since we cache the intermediate coordinates, we wont
        go through the visited coordinates again

        Space   O(row*col) depth of recursions

        """

        if len(obstacleGrid) == 0 or len(obstacleGrid[0]) == 0:

            return 0

        seen = {}

        return self.dfs(obstacleGrid, 0, 0, len(obstacleGrid)-1,
            len(obstacleGrid[0])-1, seen)

    def dfs(self, grid, i, j, m, n, seen):

        key = str(i)+"-"+str(j)

        if key in seen:

            return seen[key]

        if i == m and j == n:

            if grid[i][j] == 1:
```

```
        return 0

    return 1

    elif i > m or j > n:

        return 0

    if grid[i][j] == 1:

        seen[key] = 0

        return 0

    left = self.dfs(grid, i+1, j, m, n, seen)

    right = self.dfs(grid, i, j+1, m, n, seen)

    seen[key] = left + right

    return left + right
```