<https://leetcode.com/problems/unique-paths-ii/>

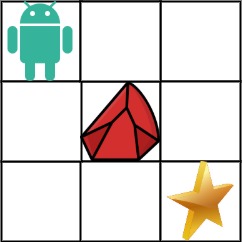
You are given an m x n integer array grid. There is a robot initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

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 \* 109.

**Example 1:**



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

**Output:** 2

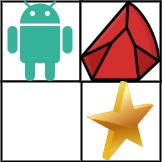
**Explanation:** There is one obstacle in the middle of the 3x3 grid above.

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.

**Attempt 1: 2023-06-11**

**Solution 1: DFS (10 min, TLE 30/41)**

**Style 1: dx[], dy[] with for loop and local variable 'result' to hold current recursion level sum up**

class Solution {

    int[] dx = {1,0};

    // Be careful, dy not {0,-1} because y's target as obstacleGrid[0].length - 1 requires increasing

    int[] dy = {0,1};

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        return helper(0, 0, obstacleGrid);

    }

    private int helper(int x, int y, int[][] obstacleGrid) {

        if(x < 0 || x >= obstacleGrid.length || y < 0 || y >= obstacleGrid[0].length || obstacleGrid[x][y] == 1) {

            return 0;

        }

        if(x == obstacleGrid.length - 1 && y == obstacleGrid[0].length - 1) {

            return 1;

        }

        int result = 0;

        for(int k = 0; k < 2; k++) {

            result += helper(x + dx[k], y + dy[k], obstacleGrid);

        }

        return result;

    }

}

**Style 2: No for loop but direct return sum up**

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        return helper(0, 0, obstacleGrid);

    }

    private int helper(int x, int y, int[][] obstacleGrid) {

        if(x < 0 || x >= obstacleGrid.length || y < 0 || y >= obstacleGrid[0].length || obstacleGrid[x][y] == 1) {

            return 0;

        }

        if(x == obstacleGrid.length - 1 && y == obstacleGrid[0].length - 1) {

            return 1;

        }

        return helper(x + 1, y, obstacleGrid) + helper(x, y + 1, obstacleGrid);

    }

}

**Refer to**

<https://leetcode.com/problems/unique-paths-ii/solutions/1180249/easy-solutions-w-explanation-comments-optimization-from-brute-force-approach/>

We start at the top-left of the **obstacleGrid** and for each cell in the grid, we can either move right or down. We can't land at an obstacle. We need to return the number of unique paths to reach the bottom-right of grid.

❌ ***Solution - I (Brute-Force) [Rejected]***

Let's build our solution starting from the brute force approach. Let's directly apply what's given in the problem statement. At each cell, we have two choices -

✦ *Go Right*

✦ *Go Down*

So, we can recursively build up our solution as -.

At each cell, explore the two choices available to us - go right & go down (**Recursive function**).

If we reach the bottom-right cell of the grid, we have found a unique path (**Base Condition - I**).

If at any time, we reach a cell with value 1, it is an **obstacle cell** and we can't move any further. So, we just stop exploring further paths from this cell (**Base Condition - II**).

We will accumulate all such unique paths.

**C++**

int m, n;

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

m = size(obstacleGrid), n = size(obstacleGrid[0]);

return solve(obstacleGrid, 0, 0);

}

// function to recursively explore all unique paths

int solve(vector<vector<int> >& grid, int i, int j){

if(i < 0 || j < 0 || i >= m || j >= n) return 0; // bounds checking

if(grid[i][j]) return 0; // if there's obstacle, just return 0 and stop further exploration

if(i == m - 1 && j == n - 1 && !grid[i][j]) return 1; // if we have reached end cell, return 1 if there's no obstacle

return solve(grid, i + 1, j) + solve(grid, i, j + 1); // explore the two choice we have at each cell

}

**Refer to**

<https://leetcode.com/problems/unique-paths-ii/solutions/2055409/beginner-friendly-recursion-to-dp-intuition-explained-python/>

Here, we have an obstacleGrid (say ***obs***) of size m\* n. We can traverse through this grid by going down and/or right only.

**Idea:** We start travering the array from start index (0, 0).

We consider a path to be valid, if it has reached the last index (m-1, n-1).

We consider a path to be invalid, if either it exceeds the boundary of the ***obs*** grid or the current position in the ***obs*** grid has an obstacle in it.

The above two conditions serves as the **base condition of our recursion**,

for **valid** cases we return 1 saying that consider this path, and

for **invalid** cases we return 0 saying that do not consider this path.

Now coming to **the recursive part**: We want to traverse the matrix in both right and down direction. So, we recursively call this function for both the right (i + 1, j) and down (i, j+1) indices.

***How does this work?***

See consider you are currently at any arbitrary index in the matrix (i, j) . Your goal is to return the max number of valid paths from (i, j) to (m-1, n-1).

From (m-1, n-1) to (m-1, n-1), we have just one valid path so return 1 directly from the base condition.

...

From (i, j) to (m-1, n-1), let us assume there was **1** possible path to go from the downside and **1** possible path to go from the rightside, so total number of ways to reach (m-1, n-1) from (i, j) becomes 1 + 1 **= 2**.

In general, if there was **x** possible path to go from the downside and **y** possible path to go from the rightside, so total number of ways to reach (m-1, n-1) from (i, j) becomes **x + y**.

Similarly, If we compute it for index (0, 0) to (m-1, n-1), we will get all possible ways to reach from start index to the end index.

The recursive code (***not working, TLE***) is given below:

class Solution:

# here I just changed the formal variable name

# from obstacleGrid to obs just for convinience

def uniquePathsWithObstacles(self, obs: List[List[int]]) -> int:

m, n = len(obs), len(obs[0])

def solve(i , j):

# base condition for recursion

if i == m - 1 and j == n - 1:

return 1

if i >= m or j >= n or obs[i][j] == 1:

return 0

# the down and right recursive calls respectively

return solve(i + 1, j) + solve(i, j+1)

return solve(0,0)

No doubt, the above solution runs perfectly fine. But it doesnot match the expected time complexity, hence results in generating a TLE error.

***Why TLE?*** =>Here there are total **m \* n** grids and each having **2** possiblities : has obstacles or clear path (does not have a obstacle). We will be recursively traversing through the grid considering both the possibilites. Hence, time complexity = O(2 ^ (m \* n)) = O(2 ^ (100 \* 100)) = O(2 ^ 10000) = O(10 ^12) nearly.

Thus, **time complexity > O(10 ^ 8)**. and hence ***Time Limit Exceeded*** as Python generally caps the complexity at O(10^8), meaning you cannot exceed it.

**Solution 2:  DFS + Memoization (10 min)**

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        Integer[][] memo = new Integer[obstacleGrid.length][obstacleGrid[0].length];

        return helper(0, 0, obstacleGrid, memo);

    }

    private int helper(int x, int y, int[][] obstacleGrid, Integer[][] memo) {

        if(x < 0 || x >= obstacleGrid.length || y < 0 || y >= obstacleGrid[0].length || obstacleGrid[x][y] == 1) {

            return 0;

        }

        if(x == obstacleGrid.length - 1 && y == obstacleGrid[0].length - 1) {

            return 1;

        }

        if(memo[x][y] != null) {

            return memo[x][y];

        }

        return memo[x][y] = helper(x + 1, y, obstacleGrid, memo) + helper(x, y + 1, obstacleGrid, memo);

    }

}

Time Complexity : O(M\*N)

Space Complexity : O(M\*N)

**Refer to**

<https://leetcode.com/problems/unique-paths-ii/solutions/1180249/easy-solutions-w-explanation-comments-optimization-from-brute-force-approach/>

✔️ ***Solution - II (Dynamic Programming - Recursive version) [Accepted]***

We can observe that there are a lot of cells that are revisited in the above appraoch and the whole path is recursed till the bottom-right cell is reached for each cell again and again. We don't need to recalculate this everytime if we just store the previously calculated result for a given cell.

We can do this by maintaining a 2d DP array. Here *dp[i][j]* will denote the number of unique paths to reach the bottom-right corner of the grid starting from the cell - *obstacleGrid[i][j]*.

**C++**

int m, n;

vector<vector<int> > dp;

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

m = size(obstacleGrid), n = size(obstacleGrid[0]);

dp.resize(m, vector<int>(n));

return solve(obstacleGrid, 0, 0);

}

// function to recursively explore all unique paths and store the results once calculated

int solve(vector<vector<int>>& grid, int i, int j) {

if(i < 0 || j < 0 || i >= m || j >= n) return 0; // bounds checking

if(grid[i][j]) return dp[i][j] = 0; // obstacle found at current cell

if(i == m - 1 && j == n - 1) return 1; // reached bottom-right of grid ? return 1

if(dp[i][j]) return dp[i][j]; // if already computed for current cell, just return the stored results

return dp[i][j] = solve(grid, i + 1, j) + solve(grid, i, j + 1); // recursively explore the two options available with us

}

***Time Complexity :*** **O(M\*N)**

***Space Complexity :*** **O(M\*N)**

**Refer to**

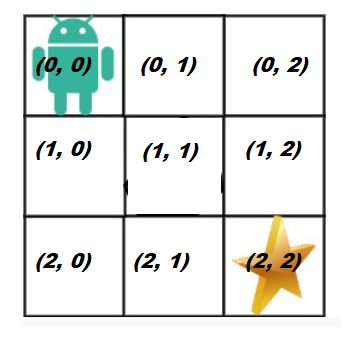
<https://leetcode.com/problems/unique-paths-ii/solutions/2055409/beginner-friendly-recursion-to-dp-intuition-explained-python/>

**Can we do any better?**

See generally when you have TLE for recursive solutions, you can always memoize it (it's just a random observation). But this is not a good reason to say to the interviewers. xD

**How to identify a DP problem?** Analyze the problem and Check for overlapping subproblems, so that we can memoize it.

Consider the first example as shown below: (indexes of each element is given in *(i, j)* format).



Let's say we are at (0, 1), so by our recursive apporach we would calculate the value of (1,1) , i.e., down and (0, 2), i.e., right indices.

Now, let's say we reached (1, 0), here also we will need to calulate (2, 0) and (1, 1).

But haven't we already calulated the number of paths for (1, 1) earlier? **Yes, we did** right (when we were traversing down from (0, 1) index).

So, why not **store it some where** so that we can use it later without the need of recomputing it.

Just store the previously computed value and return it whenever the (already computed) index (1, 0) is called. Similarly, do the same for all indices that requires recomputation.

This process of finding overlapping subproblems, storing the computed value, reusing the previously computed value and reducing the recomputation complexity is called **Memoization** or **Top Down Dynamic Programming Approach** (this is specific to recursion, it's nearly equivalent *Bottom up approach or Tabulation* is achieved using iteration).

The Memoized Code: (**Top Down DP**) => **ACCEPTED**.

class Solution:

# here I just changed the formal variable name

# from obstacleGrid to obs just for convinience

def uniquePathsWithObstacles(self, obs: List[List[int]]) -> int:

m, n = len(obs), len(obs[0])

# create a dp array of size m \* n to store already computed number of paths for index (i, j) to end

# where 0 <= i < m and 0 <= j < n

# initialize the dp array by -1 as number of paths can only be a whole number.

dp = [[-1]\*n for \_ in range(m)]

def solve(i , j):

if i >= m or j >= n or obs[i][j] == 1:

return 0

if i == m - 1 and j == n - 1:

return 1

# if the value of dp[i][j] is updated then directly return the updated value

if dp[i][j] != -1:

return dp[i][j]

# else compute the value of dp[i][j] for the first time ever

# and return it saying that the returned number (dp[i][j]) of ways are possible from

# (i, j) to the end index.

dp[i][j] = solve(i + 1, j) + solve(i, j+1)

return dp[i][j]

return solve(0,0)

**Time and Space Complexity Analysis:** (for Top Down DP)

Here, each of the successful recursive calls (whose value will be inserted in the dp matrix) will exactly be called once, because later it will take it directly from the dp array in constant time. And the unsuccessful recursive calls will return from the base condition in constant time. But we are traversing through the matrix of size 'm \* n' through nested loops (simultaneously). Thus, **optimized time complexity = O(m \* n)** owing to the traversal of matrix.

Now, we have used only one variable sized space that is dp array of size 'm \* n'. Apart from that all other variables are constant sized like n, m, ans, etc.. are fixed sized 32 - bit integer variables. So, no need to consider that, as their size will always be fixed and will not change with respect to change in input variable. Hence, **Space complexity = O(m \* n)** owing to the dp array.

**Solution 3: DP (10 min)**

**Style 1: Paddle with additional one column and one row**

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        int m = obstacleGrid.length;

        int n = obstacleGrid[0].length;

        // Using padded row and column in dp to simplify the code.

        // dp[i][j], will denote the number of unique paths to reach

        // the cell grid[i - 1][j - 1] (since padded row and column

        // used in dp at start) from the start.

        int[][] dp = new int[m + 1][n + 1];

        // dp[0][1] (or dp[1][0]) needs to be set to 1 at the start,

        // so that dp[1][1] will become 1 in our loop (denoting we

        // have one way to reach the starting cell grid[0][0]).

        dp[0][1] = 1;

        for(int i = 1; i <= m; i++) {

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

                if(obstacleGrid[i - 1][j - 1] == 1) {

                    dp[i][j] = 0;

                } else {

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

                }

            }

        }

        // e.g for input {{0,0,0},{0,1,0},{0,0,0}}, dp[3 + 1][3 + 1] as below

        // [0, 1, 0, 0]

        // [0, 1, 1, 1]

        // [0, 1, 0, 1]

        // [0, 1, 1, 2]

        return dp[m][n];

    }

}

Time Complexity : O(M\*N)

Space Complexity : O(M\*N)

**Refer to**

<https://leetcode.com/problems/unique-paths-ii/solutions/1180249/easy-solutions-w-explanation-comments-optimization-from-brute-force-approach/>

✔️ ***Solution - III (Dynamic Programming - Iterative version) [Accepted]***

We can also solve it iteratively. We have 1 way in which we can start from *grid[0][0]* and for rest of the cells, we could have reached here from the top cell or the left cell. So, we can maintain *dp[i][j]* and iteratively accumulate unique paths for current cell by adding *dp[i - 1][j]* (number of ways we reached top cell) and *dp[i][j - 1]* (number of ways we reached left cell).

Here, I am using padded row and column in *dp* to simplify the code. In this solution, *dp[i][j]*, will denote the number of unique paths to reach the cell *grid[i-1][j-1]* (since padded row and column used in dp at start) from the start.

Thus, we can iterate over the whole grid and at last return **dp[m][n]** which will be the number of unique paths to reach bottom-right of grid from the start.

Here, *dp[0][1]* (or *dp[1][0]*) needs to be set to 1 at the start, so that *dp[1][1]* will become 1 in our loop (denoting we have one way to reach the starting cell grid[0][0]).

**C++**

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

int m = size(grid), n = size(grid[0]);

vector<vector<int> > dp (m + 1, vector<int>(n + 1));

dp[0][1] = 1;

for(int i = 1; i <= m; i++)

for(int j = 1; j <= n; j++)

// dp[i][j] = sum of unique paths for top and left cell (cells from which we reach current one)

dp[i][j] = !grid[i - 1][j - 1] ? dp[i - 1][j] + dp[i][j - 1] : 0;

return dp[m][n];

}

***Time Complexity :*** **O(M\*N)**

***Space Complexity :*** **O(M\*N)**

**Style 2: No paddle with additional one column and one row**

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        // Handle special case as obstacleGrid = {{0}}

        if(obstacleGrid[0][0] == 1) {

            return 0;

        }

        int m = obstacleGrid.length;

        int n = obstacleGrid[0].length;

        int[][] dp = new int[m][n];

        dp[0][0] = 1;

        // 最上面一行和最左边一列，一旦遇到障碍就不再赋1了，因为沿着边走的那条路径被封死了

        // another style:

        /\*\*

        for(int i = 0; i < m; i++) {

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

                break;

            }

            dp[i][0] = 1;

        }

        for(int j = 0; j < n; j++) {

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

                break;

            }

            dp[0][j] = 1;

        }

        \*/

        for(int i = 1; i < m; i++) {

            if(dp[i - 1][0] == 1 && obstacleGrid[i][0] == 0) {

                dp[i][0] = 1;

            }

        }

        for(int i = 1; i < n; i++) {

            if(dp[0][i - 1] == 1 && obstacleGrid[0][i] == 0) {

                dp[0][i] = 1;

            }

        }

        for(int i = 1; i < m; i++) {

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

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

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

                }

            }

        }

        return dp[m - 1][n - 1];

    }

}

**Refer to**

<https://leetcode.com/problems/unique-paths-ii/solutions/2055409/beginner-friendly-recursion-to-dp-intuition-explained-python/>

Now, we know that, any code that is written in **Top Down DP can be easlily converted to Bottom-up DP**.

**Is there any advantage of doing so?** or Is it just a waste of time to convert the recursive code to Iterative code?

See, whenever there is recursion involved we use some extra memory in the stack region of the main memory (RAM) to store the functions that are being called recursively one after another.

This stack space is not considered while calculating the output space complexity. So, **asymptotically, it does not** matter whether you convert recursion into iteration or not. But **practically it does matter**.

***Let's see how*** => The max depth of recursion will be O(m + n) as in the worst case **either** we will go fully towards right then fully towards down **or** we will first go fully towards down then move straight fully to the right, till we reach the end index. (It will not be anything more than O(m + n)) as we do not traverse back (i.e., top or left) ever.

Hence **Stack Space =** max number of recursive functions called at a stretch = max depth of recursion **= O(m + n)**.

Thus, by coverting the Top Down DP to Bottom Up DP, we can further reduce this practical execution space. Thus, keeping our main memory little bit free for storing other data, hence reduing page faults (ignore it for now, if you don't know).

The Tabulation Method (**Bottom Up Dp**): => Definitely **accepted**..... xD

class Solution:

# here I just changed the formal variable name

# from obstacleGrid to obs just for convinience

def uniquePathsWithObstacles(self, obs: List[List[int]]) -> int:

m, n = len(obs), len(obs[0])

# create a dp array of size m \* n to store already computed number of paths for index (i, j) to end

# where 0 <= i < m and 0 <= j < n

# initialize the dp array by -1 as number of paths can only be a whole number.

dp = [[0]\*n for \_ in range(m)]

if obs[0][0] == 1 or obs[-1][-1] == 1:

return 0

# initialize the first row first column of dp

# assign the clear path = 1 and obstacles = 0 value in dp based in obstacle\_grid

# we did this because we want to add clear paths and not add obstacles path.

dp[0][0] = 1

for i in range(1,m):

dp[i][0] = 1 if obs[i][0]==0 and dp[i-1][0]==1 else 0

for j in range(1,n):

dp[0][j] = 1 if obs[0][j]==0 and dp[0][j-1]==1 else 0

# add clear paths if no obstacles are found.

for i in range(1, m):

for j in range(1, n):

if obs[i][j] == 0:

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

print(dp)

return dp[m-1][n-1]

I am leaving the above code for you to understand yourself as it will help you analyze deeper. For hints, I have given a few comments in the code.

Also, **note that** in Top Down DP we returned dp[0][0] but in Bottom Up DP Approach we returned dp[-1][-1]. Why??? - Brainstorm a little first, and do comment down below, your findings regrading this.

Time Complexity = Space Complexity = O(m \* n) for Bottom Up DP (in this case).

***Can we further reduce the space complexity?*** Yes, we can use the obstacle grid itself inplace of DP array to reduce the space complexity to O(1). But generally, modfying the input itself is not considered a wise choice, until and unless an inplace solution is required, where you do not need to return anything, just modify the input array as per conditions and the main function will access it using reference of the input variable. So, no need to return. Hence, inplace. But when nothing such is mentioned and, in real life scenerio if we modify our input array, there might be a possiblilty that this array may be used further somewhere in the program. But by modifying it, we lost the original array, which may not be a good practice.

If this helps you, then **Please UPVOTE**.

***Similar Problems with detailed intuitions explained:***

*Leetcode 329. Longest Increasing Path in a Matrix:*[*https://leetcode.com/problems/longest-increasing-path-in-a-matrix/discuss/2052360/python-beginner-friendly-recursion-to-dp-intuition-explained*](https://leetcode.com/problems/longest-increasing-path-in-a-matrix/discuss/2052360/python-beginner-friendly-recursion-to-dp-intuition-explained)

*Leetcode 576. Out of Boundary Paths:*[*https://leetcode.com/problems/out-of-boundary-paths/discuss/1293697/python-easy-to-understand-explanation-recursion-and-memoization-with-time-and-space-complexity*](https://leetcode.com/problems/out-of-boundary-paths/discuss/1293697/python-easy-to-understand-explanation-recursion-and-memoization-with-time-and-space-complexity)

*Leetcode 792. Number of Matching Subsequences:*[*https://leetcode.com/problems/number-of-matching-subsequences/discuss/1289549/python-explained-all-possible-solutions-with-time-and-space-complexity*](https://leetcode.com/problems/number-of-matching-subsequences/discuss/1289549/python-explained-all-possible-solutions-with-time-and-space-complexity)

**Solution 4:  DP + Space Optimized (10 min)**

**Style 1: Have dpPrev, more clear to explain the relation between current row and previous row**

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        // Handle special case as obstacleGrid = {{0}}

        if(obstacleGrid[0][0] == 1) {

            return 0;

        }

        int m = obstacleGrid.length;

        int n = obstacleGrid[0].length;

        int[] dp = new int[n];

        int[] dpPrev = new int[n];

        dp[0] = 1;

        dpPrev[0] = 1;

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

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

                    dp[j] = dpPrev[j] + dp[j - 1];

                } else {

                    // If not adding condition as (obstacleGrid[i][j] == 1)

                    // the original condition will be (j == 0 || obstacleGrid[i][j] == 1)

                    // and the j == 0 -> dp[j] = 0 is wrong, which means dp[0] = 1 will

                    // reset to dp[0] = 0 when i = 0, j = 0

                    if(obstacleGrid[i][j] == 1) {

                        dp[j] = 0;

                    }

                }

            }

            dpPrev = dp;

        }

        return dpPrev[n - 1];

    }

}

==================================================================

We can switch the if else condition as below

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        // Handle special case as obstacleGrid = {{0}}

        if(obstacleGrid[0][0] == 1) {

            return 0;

        }

        int m = obstacleGrid.length;

        int n = obstacleGrid[0].length;

        int[] dp = new int[n];

        int[] dpPrev = new int[n];

        dp[0] = 1;

        dpPrev[0] = 1;

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                if(obstacleGrid[i][j] == 1) {

                    dp[j] = 0;

                } else {

                    if(j > 0) {

                        dp[j] = dpPrev[j] + dp[j - 1];

                    }

                }

            }

            dpPrev = dp;

        }

        return dpPrev[n - 1];

    }

}

**Refer to**

<https://leetcode.com/problems/unique-paths-ii/solutions/1180249/easy-solutions-w-explanation-comments-optimization-from-brute-force-approach/>

✔️ ***Solution - IV (Dynamic Programming - Space Optimized) [Accepted]***

We can see that in the above solution, we are only ever accessing the current and previous rows of the dp array. So, we don't need to maintain the whole *M\*N* DP array and the space usage can be optimized by maintaining just 2 rows.

A common way of doing this with most dp problems is to declare a two rows dp matrix and just alternate between the rows at each iteration. We can alternate between the rows by doing a parity check while indexing a row of dp. Thus, we can use dp[0] at even indices and dp[1] and odd indices of iteration.

**C++**

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

int m = size(grid), n = size(grid[0]);

vector<vector<int> > dp (2, vector<int>(n + 1));

dp[0][1] = 1;

for(int i = 1; i <= m; i++)

for(int j = 1; j <= n; j++)

dp[i & 1][j] = !grid[i - 1][j - 1] ? dp[(i - 1) & 1][j] + dp[i & 1][j - 1] : 0;

return dp[m & 1][n];

}

***Time Complexity :*** **O(M\*N)**

***Space Complexity :*** **O(N)**

**Style 2: Don't have dpPrev, only dp**

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        // Handle special case as obstacleGrid = {{0}}

        if(obstacleGrid[0][0] == 1) {

            return 0;

        }

        int m = obstacleGrid.length;

        int n = obstacleGrid[0].length;

        int[] dp = new int[n];

        dp[0] = 1;

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

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

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

                } else {

                    // If not adding condition as (obstacleGrid[i][j] == 1)

                    // the original condition will be (j == 0 || obstacleGrid[i][j] == 1)

                    // and the j == 0 -> dp[j] = 0 is wrong, which means dp[0] = 1 will

                    // reset to dp[0] = 0 when i = 0, j = 0

                    if(obstacleGrid[i][j] == 1) {

                        dp[j] = 0;

                    }

                }

            }

        }

        return dp[n - 1];

    }

}

=========================================================================

We can switch the if else condition as below

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        // Handle special case as obstacleGrid = {{0}}

        if(obstacleGrid[0][0] == 1) {

            return 0;

        }

        int m = obstacleGrid.length;

        int n = obstacleGrid[0].length;

        int[] dp = new int[n];

        dp[0] = 1;

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                if(obstacleGrid[i][j] == 1) {

                    dp[j] = 0;

                } else {

                    if(j > 0) {

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

                    }

                }

            }

        }

        return dp[n - 1];

    }

}

**Refer to**

<https://leetcode.com/problems/unique-paths-ii/solutions/23250/short-java-solution/>

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

    int width = obstacleGrid[0].length;

    int[] dp = new int[width];

    dp[0] = 1;

    for (int[] row : obstacleGrid) {

        for (int j = 0; j < width; j++) {

            if (row[j] == 1)

                dp[j] = 0;

            else if (j > 0)

                dp[j] += dp[j - 1];

        }

    }

    return dp[width - 1];

}

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

is

dp[j] = dp[j] + dp[j - 1];which is

new dp[j] = old dp[j] + dp[j-1]which is

current cell = top cell + left cell

**Attempt 2: 2025-06-11**

**Refer to Deepseek**

**2D DP solution match native DFS solution**

class Solution {

    public int uniquePathsWithObstacles(int[][] obstacleGrid) {

        return helper(0, 0, obstacleGrid);

    }

    private int helper(int x, int y, int[][] obstacleGrid) {

        if(x < 0 || x >= obstacleGrid.length || y < 0 || y >= obstacleGrid[0].length || obstacleGrid[x][y] == 1) {

            return 0;

        }

        if(x == obstacleGrid.length - 1 && y == obstacleGrid[0].length - 1) {

            return 1;

        }

        return helper(x + 1, y, obstacleGrid) + helper(x, y + 1, obstacleGrid);

    }

}

Here's a 2D dynamic programming solution that exactly matches the logic of your native DFS solution for the obstacle grid problem:

class Solution {

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

int m = obstacleGrid.length;

int n = obstacleGrid[0].length;

// If start or end is blocked, return 0 immediately

if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) {

return 0;

}

int[][] dp = new int[m][n];

// Initialize destination cell

dp[m-1][n-1] = 1;

// Fill DP table from bottom-right to top-left

for (int i = m-1; i >= 0; i--) {

for (int j = n-1; j >= 0; j--) {

// Skip destination cell (already initialized)

if (i == m-1 && j == n-1) continue;

// If current cell is obstacle, no paths through it

if (obstacleGrid[i][j] == 1) {

dp[i][j] = 0;

continue;

}

// Number of ways equals sum of ways from below and right

int ways = 0;

if (i + 1 < m) ways += dp[i+1][j]; // down move

if (j + 1 < n) ways += dp[i][j+1]; // right move

dp[i][j] = ways;

}

}

return dp[0][0];

}

}

**Key Features Matching DFS Logic:**

**Base Cases**:

Returns 0 immediately if start or end position is blocked

Destination cell initialized to 1 (same as DFS base case)

**Obstacle Handling**:

Sets dp[i][j] = 0 for obstacle cells (matches DFS return 0)

Exactly replicates the obstacle check from DFS

**Recursive Relation**:

Sums paths from below and right (same as DFS recursive calls)

Uses bounds checking (same as DFS boundary checks)

**Processing Order**:

Processes grid from bottom-right to top-left

This matches how DFS would unwind its call stack

**Time and Space Complexity:**

**Time Complexity**: O(m×n) - processes each cell exactly once

**Space Complexity**: O(m×n) - for the DP table

This solution provides identical results to your DFS approach while being more efficient through dynamic programming.

**How to optimize above 2D DP into 2 rows DP ?**

Here's the space-optimized version using only 2 rows that matches the logic of your 2D DP solution:

class Solution {

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

int m = obstacleGrid.length;

int n = obstacleGrid[0].length;

// Early exit if start or end is blocked

if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) {

return 0;

}

int[] currentRow = new int[n];

int[] nextRow = new int[n];

// Initialize bottom row (last row)

nextRow[n-1] = 1;

for (int j = n-2; j >= 0; j--) {

nextRow[j] = obstacleGrid[m-1][j] == 1 ? 0 : nextRow[j+1];

}

// Process rows from bottom to top

for (int i = m-2; i >= 0; i--) {

// Initialize current row

currentRow[n-1] = obstacleGrid[i][n-1] == 1 ? 0 : nextRow[n-1];

// Process columns from right to left

for (int j = n-2; j >= 0; j--) {

if (obstacleGrid[i][j] == 1) {

currentRow[j] = 0;

} else {

currentRow[j] = currentRow[j+1] + nextRow[j];

}

}

// Swap rows for next iteration

int[] temp = nextRow;

nextRow = currentRow;

currentRow = temp;

}

return nextRow[0];

}

}

**Key Optimizations:**

**Space Reduction**:

Uses only two 1D arrays (currentRow and nextRow) instead of full 2D grid

Reduces space from O(m×n) to O(n)

**Matching 2D DP Logic**:

Still processes grid from bottom-right to top-left

Maintains same obstacle checking logic

Preserves the exact same computational dependencies

**Efficient Updates**:

currentRow[j] = currentRow[j+1] + nextRow[j] corresponds to:

currentRow[j+1] = paths from right (same row)

nextRow[j] = paths from below (next row)

**Obstacle Handling**:

Directly checks obstacleGrid[i][j] just like original

Sets to 0 immediately when obstacle found

**Why This Works:**

**Dependency Preservation**:

Each cell still only depends on its right and bottom neighbors

The right-to-left processing ensures these are available when needed

**Correct State Propagation**:

nextRow always contains the previous row's values

currentRow builds up the current row's values

Swapping ensures proper state progression

**Edge Case Handling**:

Still properly handles obstacles in first/last rows/columns

Maintains the early exit for blocked start/end positions

This solution provides identical results to your 2D DP version while using significantly less memory, and maintains the same right-to-left, bottom-to-top processing order.

**Time and Space Complexity:**

**Time Complexity**: O(m×n) - same as 2D version

**Space Complexity**: O(n) - only two 1D arrays used

**How to optimize above 2 rows DP into 1D DP ?**

Here's the most space-efficient version using just a single 1D array, while maintaining the same logic as your 2-row solution:

class Solution {

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

int m = obstacleGrid.length;

int n = obstacleGrid[0].length;

// Early exit if start or end is blocked

if (obstacleGrid[0][0] == 1 || obstacleGrid[m-1][n-1] == 1) {

return 0;

}

int[] dp = new int[n];

// Initialize last row

dp[n-1] = 1;

for (int j = n-2; j >= 0; j--) {

dp[j] = obstacleGrid[m-1][j] == 1 ? 0 : dp[j+1];

}

// Process rows from bottom to top

for (int i = m-2; i >= 0; i--) {

// Handle last column separately

dp[n-1] = obstacleGrid[i][n-1] == 1 ? 0 : dp[n-1];

// Process columns from right to left

for (int j = n-2; j >= 0; j--) {

if (obstacleGrid[i][j] == 1) {

dp[j] = 0;

} else {

dp[j] = dp[j] + dp[j+1];

}

}

}

return dp[0];

}

}

**Key Insights:**

**Single Array Usage**:

We use just one array (dp) instead of two

dp[j] serves dual purpose:

Before update: stores value from below (previous row)

After update: stores value for current cell

**Right-to-Left Processing**:

Essential for correct dependency chain

dp[j+1] is already updated for current row

dp[j] (before update) contains value from row below

**Obstacle Handling**:

Directly checks obstacle grid

Sets dp[j] = 0 immediately when obstacle found

**Edge Cases**:

Last column handled separately (only depends on cell below)

First/last row initialization preserved

**Why This Works:**

**State Transition**:

dp[j] = dp[j] + dp[j+1] breaks down to:

First dp[j]: paths from below (previous row)

dp[j+1]: paths from right (current row)

Sum gives total paths to current cell

**Memory Efficiency**:

Reduces space from O(n) to O(n)

While the complexity class is the same, we've halved the actual memory usage

**Correctness**:

Mathematically equivalent to the 2D and 2-row versions

Maintains same computation order (bottom-right to top-left)

**Time and Space Complexity:**

**Time Complexity**: O(m×n) - processes each cell exactly once

**Space Complexity**: O(n) - single array of size n

This is the most space-optimized version possible while maintaining the same computational logic as your original 2D DP solution. The right-to-left processing is crucial for the single array optimization to work correctly.

**Refer to**

[L62.Unique Paths (Ref.L63,L64,L174,L2087,L2304,L2400)](note://E33F0B728ABE4CCF8B7E554A219CF1EA)