Unique Paths II

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as `1` and `0` respectively in the grid.

For example,

There is one obstacle in the middle of a 3x3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is `2`.

Note: *m* and *n* will be at most 100.

Solution 1

just use dp to find the answer , if there is a obstacle at (i,j), then dp[i][j] = 0. time is O(n*m) , space is O(n*m) . here is my code:

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {
        int m = obstacleGrid.size() , n = obstacleGrid[0].size();
        vector<vector<int>> dp(m+1,vector<int>(n+1,0));
        dp[0][1] = 1;
        for(int i = 1 ; i <= m ; ++i)
            for(int j = 1 ; j <= n ; ++j)
                if(!obstacleGrid[i-1][j-1])
                    dp[i][j] = dp[i-1][j]+dp[i][j-1];
        return dp[m][n];
    }
};
```

written by kingmacrobo original link here

Solution 2

```
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];
}
```

written by tusizi original link here

Solution 3

Well, this problem is similar to **Unique Paths**. The introduction of obstacles only changes the boundary conditions and make some points unreachable (simply set to `0`).

Denote the number of paths to arrive at point `(i, j)` to be `P[i][j]`, the state equation is `P[i][j] = P[i - 1][j] + P[i][j - 1]` if `obstacleGrid[i][j] != 1` and `0` otherwise.

Now let's finish the boundary conditions. In the **Unique Paths** problem, we initialize `P[0][j] = 1, P[i][0] = 1` for all valid `i, j`. Now, due to obstacles, some boundary points are no longer reachable and need to be initialized to `0`. For example, if `obstacleGrid` is like `[0, 0, 1, 0, 0]`, then the last three points are not reachable and need to be initialized to be `0`. The result is `[1, 1, 0, 0, 0]`.

Now we can write down the following (unoptimized) code. Note that we pad the `obstacleGrid` by `1` and initialize `dp[0][1] = 1` to unify the boundary cases.

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        vector<vector<int> > dp(m + 1, vector<int> (n + 1, 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])
                    dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        return dp[m][n];
    }
};
```

Well, the code is accepted but it has some obvious redundancy. There are two major concerns:

1. Each time when we update `path[i][j]`, we only need `path[i - 1][j]` (at the same column) and `path[i][j - 1]` (at the left column), so it is unnecessary to maintain the full `m*n` matrix. Maintaining two columns is enough.
2. There are some cases that the loop can be terminated earlier. Suppose `obstacleGrid = [[0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0]]`, then we can see that it is impossible to reach the bottom-right corner after updating the second column since the number of paths to reach each element in the second column is `0`.

Taken these into considerations, we write down the following optimized code.

```cpp
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<int> pre(m, 0);
        vector<int> cur(m, 0);
        for (int i = 0; i < m; i++) {
            if (!obstacleGrid[i][0])
                pre[i] = 1;
            else break;
        }
        for (int j = 1; j < n; j++) {
            bool flag = false;
            if (!obstacleGrid[0][j]) {
                cur[0] = pre[0];
                if (cur[0]) flag = true;
            }
            else cur[0] = 0;
            for (int i = 1; i < m; i++) {
                if (!obstacleGrid[i][j]) {
                    cur[i] = cur[i - 1] + pre[i];
                    if (cur[i]) flag = true;
                }
                else cur[i] = 0;
            }
            if (!flag) return 0;
            swap(pre, cur);
        }
        return pre[m - 1];
    }
};
```

Further inspecting the above code, keeping two vectors only serve for the purpose of recovering `pre[i]`, which is simply `cur[i]` before its update. So we can use only one vector and the space is further optimized.

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<int> cur(m, 0);
        for (int i = 0; i < m; i++) {
            if (!obstacleGrid[i][0])
                cur[i] = 1;
            else break;
        }
        for (int j = 1; j < n; j++) {
            bool flag = false;
            if (obstacleGrid[0][j])
                cur[0] = 0;
            else flag = true;
            for (int i = 1; i < m; i++) {
                if (!obstacleGrid[i][j]) {
                    cur[i] += cur[i - 1];
                    if (cur[i]) flag = true;
                }
                else cur[i] = 0;
            }
            if (!flag) return 0;
        }
        return cur[m - 1];
    }
};
```

written by jianchao.li.fighter original link here