Grid Unique paths

Problem Statement

Given two values M and N, which represent a matrix[M][N]. We need to find the total unique paths from the top-left cell (matrix[0][0]) to the rightmost cell (matrix[M-1][N-1]).

At any cell we are allowed to move in only two directions, bottom and right.

Editorial

Recursion

Approach

We have to count all possible ways to go from [0,0] to [m-1,n-1].

We need to write a function that calls itself(Recursion) and at one point gives the final answer.

Define the function statement: `f(int i, int j)` : The number of unique ways to go from (0, 0) to (i, j).

Fix one direction: We chose to start from last row and column. Consider all changes: Travel Up and Left instead of Right and Bottom as mentioned in the question.

Fix answer output: Since we started from the last element, path ends if we reach the first element (0, 0).

- Hence, base case:

  ```
  if(i==0 && j==0) return 1;
  ```

- Returning 1 means counting that way.

Define all Corner cases: If index crosses the boundary, do not consider it, return 0.

```
if(i < 0 || j < 0) return 0;
```

Define action at every step: We either go Up or Left.

- If we go Up: f(i-1, j); If we go Left: f(i, j-1);

```
up = f(i-1, j);
left: f(i, j-1);
```

Define the output: We need total unique ways.

```
return up + left
```

Code

```
int uniquePaths(int i, int j) {
    if (i == 0 && j == 0) return 1;
    if (i < 0 || j < 0) return 0;
    int up = uniquePaths(i - 1, j);
    int left = uniquePaths(i, j - 1);
```

```
    return up + left;
}
```

Complexity Analysis

Time complexity: `2^(MxN)`

- For each element, we are considering 2 ways, left and top.
- Exponential time complexity.

Space complexity: The stack space: Path length

Memoization

Approach

Draw the recursion tree, you will find overlapping subproblems.

- Figure out changing parameters: `i, j`
- Define the size of dp array: MxN : `vector<vector<int>> dp(m, vector<int>(N, -1));`
- Store the answer we are returning: dp[i][j] = up + left;
- Check if answer is answer is stored or not. If yes, return the answer: if(dp[i][j] ≠ -1) return dp[i][j]

Code

```
int countPaths(int i, int j, vector<vector<int>>& dp) {
    if (i == 0 && j == 0) return 1;
    if (i < 0 || j < 0) return 0;
    if (dp[i][j] != -1) return dp[i][j];
    int up = countPaths(i - 1, j, dp);
    int left = countPaths(i, j - 1, dp);
    return dp[i][j] = up + left;
}
```

Complexity Analysis:

Time complexity: `O(MxN)` - The total number of recursion calls will only be MxN.

Space complexity: Path length + dp array: `O((m-1)+(n-1)) + O(MxN)`

Tabulation

Approach

Declare the dp array of same size MxN.

Identify the problem type:

- From cell `(i, j)`, you can reach it either from `(i-1, j)` or `(i, j-1)`.
- The number of unique paths to reach a cell depends on paths to adjacent sub-cells.

Decide DP table structure:

- `dp[i][j]` represents **the number of unique paths to reach cell `(i, j)`**.

Define the base case(s):

- `dp[0][0] = 1`: there's **only one way** to be at the start - by being there.

Decide the state transition:

What relation allows building up the solution using previously solved subproblems?

For cell `(i, j)`:

You can only come from **top** or **left**:

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

Choose iteration order:

How should the loops be arranged to fill the table correctly?

- Outer loop: `for i = 0 to m-1` (rows)
- Inner loop: `for j = 0 to n-1` (columns)
- This ensures we always have `dp[i-1][j]` and `dp[i][j-1]` already computed.

Case 1: `i == 0` (first row)

- No cell **above** exists → `dp[i-1][j]` is **invalid**.
- So `up = 0`

Case 2: `j == 0` (first column)

- No cell **left** exists → `dp[i][j-1]` is **invalid**.
- So `left = 0`

Final result

The answer to the full problem is stored in `dp[m-1][n-1]`.

Code

```
int countWays(int m, int n) {
    vector<vector<int>> dp(m, vector<int>(n, 0));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (i == 0 && j == 0)
                dp[i][j] = 1;
            else {
                int up = (i > 0) ? dp[i - 1][j] : 0;
                int left = (j > 0) ? dp[i][j - 1] : 0;
                dp[i][j] = up + left;
            }
        }
    }
    return dp[m - 1][n - 1];
}
```

Complexity Analysis

Time Complexity: `O(MxN)` - 2 nested loops.

Space Complexity: `O(MxN)` - 2D DP Array.

Space optimization

Approach

Identify what you actually use at each step

In most 2D DP tabulation problems (especially grid problems), notice this:

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

At any point, to compute `dp[i][j]`, you only need:

- `dp[i-1][j]` → value **above**
- `dp[i][j-1]` → value **to the left**

So you only need **previous row(for up)** and **current row(for left).**

Look how the loop is running. First one row will be started, after completion of all comumns, it will move to the next row.

For each element: We need one up value and one left value.

But, on the first row, we won't be having up values.

Based on this so far, we need a linear array that stores the prev row data, all zeroes for the first time or first row.

To make sure every element gets its own corresponding up values, we need to update the rows everytime.

Hence, create a temporary array for current row, compute the values and before moving to the next row we need to update the prev row to the current row so that new rows values can be calculated correctly.

```
prev = curr;
```

Code

```
int f(int m, int n) {
    vector<int> prev(n, 0);
    for (int i = 0; i < m; i++) {
        vector<int> curr(n, 0);
        for (int j = 0; j < n; j++) {
            if (i == 0 && j == 0) {
                curr[j] = 1;
            } else {
                int up = (i > 0) ? prev[j] : 0;
                int left = (j > 0) ? curr[j - 1] : 0;
                curr[j] = up + left;
            }
        }
        prev = curr;
    }
    return prev[n - 1];
}
```

In the last iteration, the final row will be stored in prev and comes out. Hence the final answer will be in prev[n-1].

Complexity Analysis:

Time Complexity: `O(MxN)` - 2 nested loops.

Space Complexity: `O(N)` - 1D DP Array.

THE END