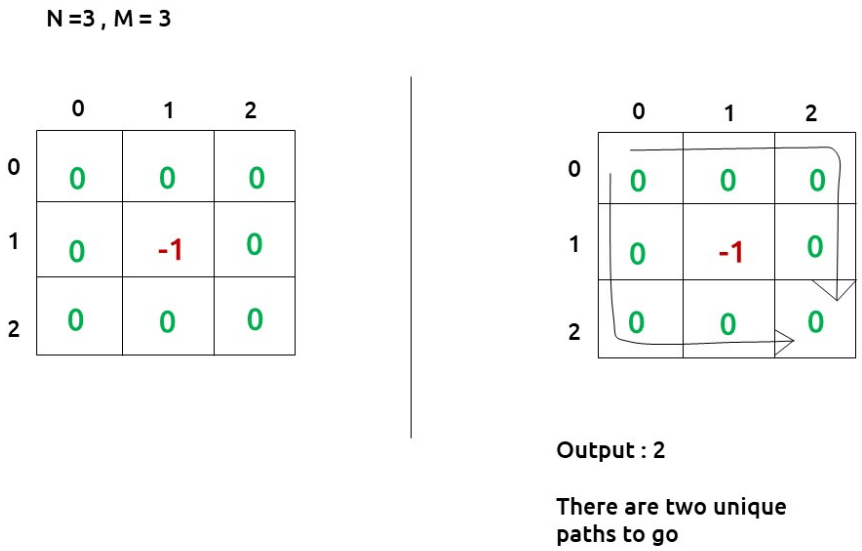
Grid Unique Paths 2 (DP 9)

Problem Description:

We are given an “N*M” Maze. The maze contains some obstacles. A cell is ‘blockage’ in the maze if its value is -1. 0 represents non-blockage. There is no path possible through a blocked cell.

We need to count the total number of unique paths from the top-left corner of the maze to the bottom-right corner. At every cell, we can move either down or towards the right.

Example:



Note: If the question asks to return the answer by performing a modulo operation, please do so.

Disclaimer: Don’t jump directly to the solution, try it out yourself first.

Pre-req: Grid Unique Paths

Solution :

This question is a slight modification of the question discussed in /*** DP 8 Link ***/. In the previous problem, there were no obstacles whereas this problem has them.

Let us look at this example:

N =3 , M = 3

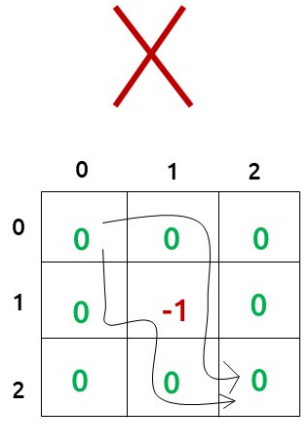
	0	1	2
0	0	0	0
1	0	-1	0
2	0	0	0

A path through the cell[1][1] is not possible as it is blocked, therefore we need to count every other legit path which doesn't include the cell[1][1].

N =3 , M = 3

	0	1	2
0	0	0	0
1	0	-1	0
2	0	0	0

These two paths doesn't pass through the blocked cell.



These paths are not allowed as they pass through the blocked cell.

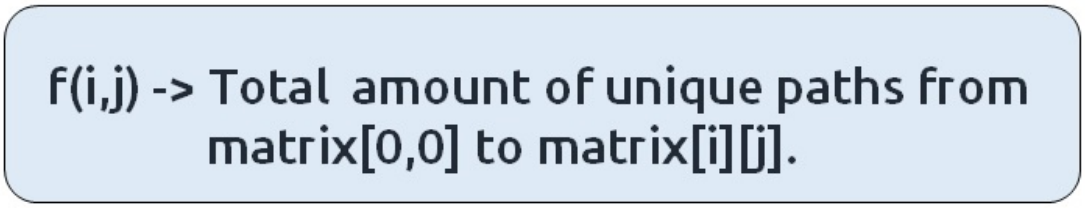
Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in Dynamic Programming Introduction.

Step 1: Express the problem in terms of indexes.

We are given two variables N and M, representing the dimensions of the maze.

We can define the function with two parameters i and j, where i and j represent the row and column of the maze.



We will form f(i,j) in such a way that it will handle the obstacles.

We will be doing a top-down recursion, i.e we will move from the cell[M-1][N-1] and try to find our way to the cell[0][0]. Therefore at every index, we will try to move up and towards the left.

Base Case:

There will be three base cases:

- When i>0 and j>0 and mat[i][j] = -1, it means that the current cell is an obstacle, so we can't find a path from here. Therefore, we return 0.
- When i=0 and j=0, that is we have reached the destination so we can count the current path that is going on, hence we return 1.
- When i<0 and j<0, it means that we have crossed the boundary of the matrix and we couldn't find a right path, hence we return 0.

The pseudocode till this step will be:

```
f(i,j) {
      if( i>0 && j>0 && mat[i][j]==-1)
            return 0
      if( i==0 && j==0)   return 1
      if( i<0 || j<0) return 0

      }
```

Step 2: Try out all possible choices at a given index.

As we are writing a top-down recursion, at every index we have two choices, one to go up(↑) and the other to go left(←). To go upwards, we will reduce i by 1, and move towards left we will reduce j by 1.

```
f(i,j) {
      if( i>0 && j>0 && mat[i][j]==-1)
            return 0
      if( i==0 && j==0)   return 1
      if( i<0 || j<0) return 0
       up = f(i-1,j)
       left = f(i,j-1)

       }
```

Step 3: Take the maximum of all choices

As we have to count all the possible unique paths, we will return the sum of the choices(up and left)

The final pseudocode after steps 1, 2, and 3:

```
f(i,j) {
        if( i>0 && j>0 && mat[i][j]==-1)
                return 0
        if( i==0 && j==0)   return 1
        if( i<0 || j<0) return 0
         up = f(i-1,j)
         left = f(i,j-1)
         return up+left
        }
```

Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [n][m]
2. Whenever we want to find the answer of a particular row and column (say f(i,j)), we first check whether the answer is already calculated using the dp array(i.e dp[i][j]!= -1). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[i][j] to the solution we get.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int mazeObstaclesUtil(int i, int j, vector<vector<int> > &maze, vector<vector<int>
>
&dp) {
  if(i>0 && j>0 && maze[i][j]==-1) return 0;
  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 = mazeObstaclesUtil(i-1,j,maze,dp);
  int left = mazeObstaclesUtil(i,j-1,maze,dp);

  return dp[i][j] = up+left;

}
int mazeObstacles(int n, int m, vector<vector<int> > &maze){
    vector<vector<int> > dp(n,vector<int>(m,-1));
    return mazeObstaclesUtil(n-1,m-1,maze,dp);
}
int main() {
  vector<vector<int> > maze{{0,0,0},
                            {0,-1,0},
                            {0,0,0}};
  int n = maze.size();
  int m = maze[0].size();
  cout<<mazeObstacles(n,m,maze);
}
```

Output:

2

Time Complexity: O(N*M)

Reason: At max, there will be N*M calls of recursion.

Space Complexity: O((M-1)+(N-1)) + O(N*M)

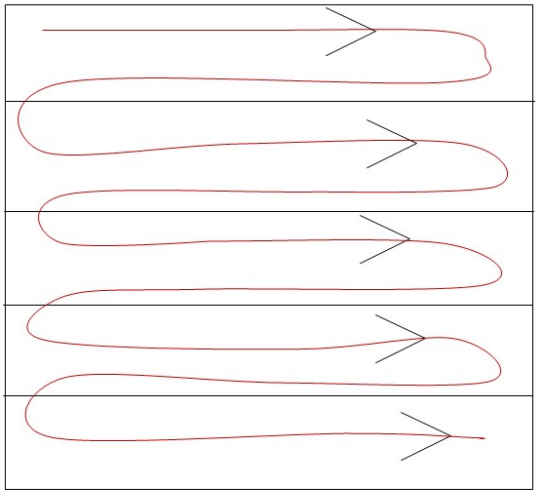
Reason: We are using a recursion stack space:O((M-1)+(N-1)), here (M-1)+(N-1) is the path length and an external DP Array of size 'N*M'.

Steps to convert Recursive Solution to Tabulation one.

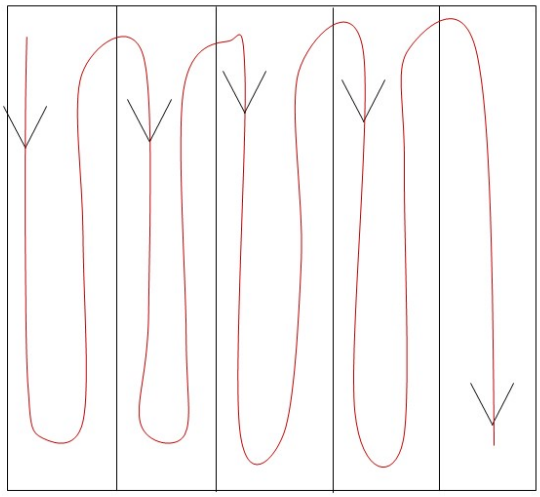
Tabulation is the bottom-up approach, which means we will go from the base case to the main problem.

The steps to convert to the tabular solution are given below:

- Declare a dp[] array of size [n][m].
- First initialize the base condition values, i.e dp[0][0] = 1
- Our answer should get stored in dp[n-1][m-1]. We want to move from (0,0) to (n-1,m-1). But we can't move arbitrarily, we should move such that at a particular i and j, we have all the values required to compute dp[i][j].
- If we see the memoized code, values required for dp[i][j] are: dp[i-1][j] and dp[i][j-1]. So we only use the previous row and column value.
- We have already filled the top-left corner (i=0 and j=0), if we move in any of the two following ways(given below), at every cell we do have all the previous values required to compute its value.



from (0,0) to (n-1,m-1)



from (0,0) to (n-1,m-1)

- We can use two nested loops to have this traversal
- Whenever i>0 , j>0 and mat[i][j]==-1, we will simply mark dp[i][j] = 0, it means that this cell is a blocked one and no path is possible through it.
- At every cell we calculate up and left as we had done in the recursive solution and then assign the cell's value as (up+left)

Note: For the first row and first column (except for the top-left cell), then up and left values will be zero respectively.

Code:

```
#include <bits/stdc++.h>

using namespace std;

int mazeObstaclesUtil(int n, int m, vector<vector<int>>
&maze,vector<vector<int>>
 &dp)
{
  for(int i=0; i<n ;i++){
     for(int j=0; j<m; j++){

        //base conditions
        if(i>0 && j>0 && maze[i][j]==-1){
          dp[i][j]=0;
          continue;
        }
        if(i==0 && j==0){
           dp[i][j]=1;
           continue;
        }

        int up=0;
        int left = 0;

        if(i>0)
          up = dp[i-1][j];
        if(j>0)
          left = dp[i][j-1];

        dp[i][j] = up+left;
     }
```

```
  }

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


}

int mazeObstacles(int n, int m, vector<vector<int> > &maze){
    vector<vector<int> > dp(n,vector<int>(m,-1));
    return mazeObstaclesUtil(n,m,maze,dp);

}

int main() {

  vector<vector<int> > maze{{0,0,0},
                            {0,-1,0},
                            {0,0,0}};

  int n = maze.size();
  int m = maze[0].size();

  cout<<mazeObstacles(n,m,maze);
}
```

Output:

2

Time Complexity: O(N*M)

Reason: There are two nested loops

Space Complexity: O(N*M)

Reason: We are using an external array of size ‘N*M’’.