576. Out of Boundary Paths

**Dynamic Programming** 

## **Problem Description**

Medium

The problem describes a scenario where a ball is placed on a grid that is m by n in size, and the ball is initially positioned at the grid cell [startRow, startColumn]. The goal is to determine the number of distinct paths that can move the ball out of the grid boundaries, with the constraint that the ball can only be moved a maximum of maxMove times. At each move, the ball can only go to one of the four adjacent cells - up, down, left, or right - and it might move off the grid.

The problem asks for the total number of such paths where the ball exits the grid, constrained by the number of moves allowed,

and because the resulting number can be very large, it is necessary to return this number modulo 10^9 + 7.

Intuition

The intuition behind solving this problem lies in using Depth-First Search (DFS) to explore all possible paths from the start

### position until the ball moves out of the grid or until we have made maxMove moves. To find all paths efficiently, we can use

dynamic programming (specifically memoization) to store the results of sub-problems we've already solved. This will avoid redundant computations for the same position and number of moves left, which otherwise would significantly increase the runtime complexity. Specifically, we can apply memoization to remember the number of paths from a given position (i, j) with k moves remaining that lead to the ball exiting the grid.

The recursive function dfs(i, j, k) will return the number of pathways that lead out of the grid starting from cell (i, j) with k moves left. If the ball's current position is already outside the grid, the function returns 1 as it indicates a valid exit path. If no moves are left (k <= 0), the function returns 0 since the ball cannot move further. For each position (i, j) within the grid and with moves remaining, the function explores all four adjacent cells, cumulatively adding up the number of exit paths from those cells to the current count. To keep the count within the specified modulo constraint, the result is taken modulo 10^9 + 7 after each addition.

Solution Approach

The implementation of the solution takes a recursive approach with <u>dynamic programming</u> - specifically, it uses memoization to

By starting the recursion with dfs(startRow, startColumn, maxMove), we kick off this exploration process and allow the recursive

function to calculate the total number of exit paths from our start position, adhering to the move limit and memoizing along the

## the number of remaining moves. • The base case of the recursion checks if the current cell is out of bounds - that is, if i, j, k indicates a position outside the grid. If it is out of

way to ensure efficiency.

optimize the depth-first search (DFS) algorithm.

x, y are calculated by adding a, b to i, j.

Here's a step-by-step explanation of how this is done:

bounds, we have a complete path, so it returns 1.
If k is 0, meaning no more moves are left, it returns 0 since the ball cannot move out of the grid with no moves remaining.
To avoid repeating calculations for the same positions and move counts, the @cache decorator is used, which stores the results of the dfs calls

• A recursive helper function, dfs(i, j, k), takes parameters i and j representing the current cell's row and column indices, and k representing

- and returns the cached value when the same arguments are encountered again.
  The function then initializes res as 0 to accumulate the number of paths going out of the grid from the current position.
  For each allowed move (up, down, left, right), represented by a, b in directions [[-1, 0], [1, 0], [0, 1], [0, -1]], the new cell coordinates
- ensuring to apply the modulo operation % mod to keep the number under 10^9 + 7.
   Finally, after going through all possible adjacent moves, the function returns res, the total number of paths from the current cell leading out of the grid.

• It then recursively calls dfs(x, y, k - 1) to account for moving to the new cell with one less remaining move, and adds this to the res while

- By calling dfs(startRow, startColumn, maxMove), we start the recursive search from the initial ball position with the maximum allowed moves. The returned result is the total number of paths modulo 10^9 + 7. The algorithm ensures efficiency by avoiding
- recomputation and considers all possible move sequences leading to an exit path.
- Let's use a small example to illustrate the solution approach. Suppose we have a grid that is  $3\times3$  in size (m = 3, n = 3), and the ball is initially placed at the center grid cell [1, 1]. We want to find the number of distinct paths that can move the ball out of the grid

if it can only be moved a maximum of 2 times (maxMove = 2).

The grid can be visualized as:
---+---+

### Where 'S' represents the start position of the ball.

**Example Walkthrough** 

recursion continues.
 From here, dfs(-1, 1, 0) → moves the ball one cell up and out of grid (valid path).
 Other directions for this step would keep the ball in bounds (so no additional paths added).
 Moving down: Call dfs(2, 1, 1) → moves the ball one cell down. The ball is still in bounds so recursion continues.

Moving up: Call dfs(0, 1, 1) → moves the ball one cell up. Since we can still move 1 more time and the current position is within bounds,

After exploring all possible first moves and the ensuing second move from [1, 1] with 2 moves allowed, res will accumulate

By following the steps described in the solution approach and utilizing memoization, we efficiently calculate the total number of

distinct paths out of the grid using a recursive DFS algorithm. This approach can be extended to larger grids and more moves as

Since [1, 1] is within the grid bounds and we have more than 0 moves, the recursive function continues.

 $\circ$  Moving left: Call dfs(1, 0, 1)  $\rightarrow$  moves the ball one cell to the left. The ball is still in bounds so recursion continues.

Moving right: Call dfs(1, 2, 1) → moves the ball one cell to the right. The ball is still in bounds so recursion continues.

the total paths leading out of the grid. Here, we find that there are a total of 4 paths (one for each direction).

6. Finally, since we do not exceed the modulo 10^9 + 7 with such small numbers, res equals 4. So, dfs(1, 1, 2) returns 4.

def findPaths(self, m: int, n: int, maxMove: int, startRow: int, startColumn: int) -> int:

# Using the Least Recently Used (LRU) cache decorator for memoization to optimize the solution

We begin by calling the recursive helper function dfs(1, 1, 2).

We check all four possible directions the ball can move: up, down, left, and right.

From here, dfs(3, 1, 0) → moves the ball one cell down and out of grid (valid path).

Other directions for this step would keep the ball in bounds (so no additional paths added).

■ From here,  $dfs(1, -1, 0) \rightarrow$  moves the ball one cell to the left and out of grid (valid path).

■ From here,  $dfs(1, 3, 0) \rightarrow$  moves the ball one cell to the right and out of grid (valid path).

Each time the ball moves out of grid, we return 1 and add that to our total res.

from functools import lru\_cache
class Solution:

# Base case: if the current cell (row,col) is out of bounds, return 1 since we've found a way out

if remaining\_moves <= 0:
 return 0

# Accumulate paths' counts from the current cell
paths\_count = 0
# Possible movement directions: up, down, right, left
directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]</pre>

private static final int[] DIRECTIONS =  $\{-1, 0, 1, 0, -1\}$ ; // Direction array to facilitate movement to adjacent cells

# # Modulo constant for the problem MOD = 10\*\*9 + 7 # Initial call to depth-first search return dfs(startRow, startColumn, maxMove)

rowCount = m;

colCount = n;

Java

class Solution {

required by the problem.

**Python** 

Solution Implementation

@lru\_cache(maxsize=None)

return 1

def dfs(row, col, remaining\_moves):

for direction in directions:

paths\_count %= MOD

for (int[][] grid : memoization) {

for (int[] row : grid) {

return paths\_count

if row < 0 or col < 0 or row >= m or col >= n:

# Define the new cell to be explored

# Recursive call for the next move

private int rowCount; // Total number of rows in the grid

private int colCount; // Total number of columns in the grid

memoization = new int[rowCount][colCount][maxMove + 1];

// Take modulo after each addition to prevent overflow.

// Cache the result in the memoization table before returning.

memo[row][col][remainingMoves] = pathCount;

pathCount %= MOD;

# Initial call to depth-first search

Time and Space Complexity

k) triplet.

return dfs(startRow, startColumn, maxMove)

multiplied by the constant gives us the time complexity.

return pathCount;

**}**;

**TypeScript** 

# If no moves left, return 0 since we cannot move further

new\_row, new\_col = row + direction[0], col + direction[1]

paths\_count += dfs(new\_row, new\_col, remaining\_moves - 1)

private int[][][] memoization; // Memoization array to store results of sub-problems

// Function to calculate the number of paths to move the ball out of the grid boundary

public int findPaths(int m, int n, int maxMove, int startRow, int startColumn) {

// Initialize the memoization array with -1 to indicate uncomputed states

private static final int MOD = (int) 1e9 + 7; // Modulus value for the result to prevent overflow

# Modulo operation for the final result to prevent integer overflow

```
Arrays.fill(row, -1);
       // Run DFS from the starting cell
       return dfs(startRow, startColumn, maxMove);
    // Helper function using DFS to find paths, with memoization to optimize overlapping subproblems
    private int dfs(int row, int col, int remainingMoves) {
       // Base case: if the current position is out of the grid, return 1
       if (row < 0 || row >= rowCount || col < 0 || col >= colCount) {
            return 1;
       // Check if the current state is already computed
       if (memoization[row][col][remainingMoves] != -1) {
            return memoization[row][col][remainingMoves];
       // Base case: if no moves left, return 0 as we can't move further
        if (remainingMoves == 0) {
            return 0;
       // Variable to hold the result
       int pathCount = 0;
       // Iterate through all possible directions and explore further moves
        for (int index = 0; index < 4; ++index) {</pre>
            int nextRow = row + DIRECTIONS[index];
            int nextCol = col + DIRECTIONS[index + 1];
           // Recur for the next state with one less move
            pathCount += dfs(nextRow, nextCol, remainingMoves - 1);
            // Apply modulo operation to prevent overflow
            pathCount %= MOD;
       // Store the computed result in the memoization array
       memoization[row][col][remainingMoves] = pathCount;
       // Return the total number of paths for the current state
       return pathCount;
C++
class Solution {
public:
    int maxRowCount;
    int maxColCount;
    const int MOD = 1e9 + 7; // A constant to take modulo after additions.
    int memo[51][51][51]; // Memoization table to store intermediate results.
    int directions [5] = \{-1, 0, 1, 0, -1\}; // Directions array to perform DFS.
    // Public method to find the number of possible paths which move out of boundary
    int findPaths(int m, int n, int maxMove, int startRow, int startColumn) {
       memset(memo, -1, sizeof(memo)); // Initialize memoization table with -1.
        this->maxRowCount = m; // Initialize number of rows in the grid.
        this->maxColCount = n; // Initialize number of columns in the grid.
       return dfs(startRow, startColumn, maxMove);
   // Private helper method for performing DFS traversal.
    int dfs(int row, int col, int remainingMoves) {
       // Base condition: if out of boundary, return 1 as one path is found.
       if (row < 0 || row >= maxRowCount || col < 0 || col >= maxColCount) return 1;
       // Return cached result if already computed.
       if (memo[row][col][remainingMoves] != -1) return memo[row][col][remainingMoves];
       // Base condition: if no more moves left, return 0 as no path can be made.
       if (remainingMoves == 0) return 0;
       // Variable to store the result of paths from the current cell.
       int pathCount = 0;
       // Explore all adjacent cells in order up, right, down, and left.
        for (int t = 0; t < 4; ++t) {
            int nextRow = row + directions[t];
            int nextCol = col + directions[t + 1];
           // Perform DFS for the next cell with one less remaining move.
            pathCount += dfs(nextRow, nextCol, remainingMoves - 1);
```

```
// Define maxRowCount and maxColCount to track the grid dimensions.
  let maxRowCount: number;
  let maxColCount: number;
  // Define MOD as a constant to take modulo after additions to prevent overflow.
  const MOD: number = 1e9 + 7;
  // Initialize memoization table to store intermediate results.
  let memo: number[][][] = [];
  // Directions array to facilitate DFS moves: Up, Right, Down, Left.
  const directions: number[] = [-1, 0, 1, 0, -1];
  // Function to find the number of possible paths which move out of boundary.
  function findPaths(m: number, n: number, maxMove: number, startRow: number, startColumn: number): number {
      // Initialize the memoization table with '-1' to indicate uncomputed states.
      memo = [...Array(51)].map(() => [...Array(51)].map(() => Array(51).fill(-1)));
      // Set up the grid dimensions.
      maxRowCount = m;
      maxColCount = n;
      // Initiate DFS and return the resultant path count.
      return dfs(startRow, startColumn, maxMove);
  // Helper method for performing DFS traversal to compute path counts.
  function dfs(row: number, col: number, remainingMoves: number): number {
      // If the cell is out of boundary, return 1 as we found a path out.
      if (row < 0 || row >= maxRowCount || col < 0 || col >= maxColCount) return 1;
      // Return cached result if this state has already been computed.
      if (memo[row][col][remainingMoves] !== −1) return memo[row][col][remainingMoves];
      // If no more moves are left, return 0 as no further path can be made.
      if (remainingMoves === 0) return 0;
      // Variable to keep track of the number of paths from the current cell.
      let pathCount: number = 0;
      // Iterate over all 4 potential moves (Up, Right, Down, Left).
      for (let i = 0; i < 4; i++) {
          let nextRow = row + directions[i];
          let nextCol = col + directions[i + 1];
          // Recursive DFS call for the adjacent cell with one less remaining move.
          pathCount += dfs(nextRow, nextCol, remainingMoves - 1);
          // Taking modulo to prevent integer overflow.
          pathCount %= MOD;
      // Cache the computed result in the memo table before returning.
      memo[row][col][remainingMoves] = pathCount;
      return pathCount;
from functools import lru_cache
class Solution:
   def findPaths(self, m: int, n: int, maxMove: int, startRow: int, startColumn: int) -> int:
       # Using the Least Recently Used (LRU) cache decorator for memoization to optimize the solution
       @lru_cache(maxsize=None)
       def dfs(row, col, remaining_moves):
           # Base case: if the current cell (row,col) is out of bounds, return 1 since we've found a way out
           if row < 0 or col < 0 or row >= m or col >= n:
            # If no moves left, return 0 since we cannot move further
            if remaining_moves <= 0:</pre>
                return 0
            # Accumulate paths' counts from the current cell
            paths_count = 0
            # Possible movement directions: up, down, right, left
            directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]
            for direction in directions:
               # Define the new cell to be explored
                new_row, new_col = row + direction[0], col + direction[1]
                # Recursive call for the next move
                paths_count += dfs(new_row, new_col, remaining_moves - 1)
                # Modulo operation for the final result to prevent integer overflow
                paths_count %= MOD
            return paths_count
       # Modulo constant for the problem
       MOD = 10**9 + 7
```

The time complexity of the given code is 0(m \* n \* maxMove). Each state in the dynamic programming (the dfs function is

i, the current column j, and the remaining number of moves k. There are m rows, n columns, and maxMove possible remaining

moves. For each state, we perform a constant amount of work (exploring 4 adjacent cells), hence the total number of states

essentially dynamic programming with memoization due to the @cache decorator) is defined by three parameters: the current row

The space complexity is also 0(m \* n \* maxMove) because we need to store the result for each state to avoid recomputation. The

space is used by the cache that stores intermediate results of the dfs function for each of the states characterized by the (i, j,