2912. Number of Ways to Reach Destination in the Grid

Hard

Leetcode Link

Problem Description

In this problem, you're given a grid of size `n` by `m` and starting at a 1-indexed position called `source`, you need to find the number of distinct ways to reach another 1-indexed position called `dest`, by moving exactly `k` steps. The condition for moving is that you can only move to another cell in the same row or column, but not to the cell you are currently in. Since the number of possible ways can be quite large, you are required to return the answer modulo `10^9 + 7` to keep the number within manageable limits.

The movement rules imply you're only allowed to travel in a straight line horizontally or vertically before changing directions, and you cannot stay in the same cell if you're making a move.

The goal is to return the number of unique paths that take exactly `k` moves from the `source` to `dest`, under these constraints.

Intuition

The solution to this problem revolves around the concept of dynamic programming, which roughly involves breaking down a larger problem into smaller, similar subproblems and storing the results to avoid redundant computations.

Here, we use an array `f` which will represent the number of different ways to end at a cell, under different conditions, after taking some number of steps:

1. `f[0]`: Staying at the source cell.
2. `f[1]`: Being in another cell in the same column as the source.
3. `f[2]`: Being in another cell in the same row as the source.
4. `f[3]`: Being in any cell that is neither in the same row nor column as the source.

At the start, `f[0]` is 1 because there's exactly one way to be at the source - by not moving. The others are zero because we've taken no steps to move elsewhere.

For each step from 1 to `k`, you update these numbers based on where you can move from each of these states, which is derived from:

- From `f[0]`, you can go to any other cell in the same row (`f[2]`) or column (`f[1]`).
- From `f[1]`, you can stay in the same column (`f[1]`), move back to the source (`f[0]`), or move to a different row and column (`f[3]`).
- From `f[2]`, you can stay in the same row (`f[2]`), move back to the source (`f[0]`), or move to a different row and column (`f[3]`).
- From `f[3]`, you can move to other cells not in the same row or column (`f[3]`), or move to a cell in the same row (`f[2]`) or column (`f[1]`) as the source.

The formulae in the 'Reference Solution Approach' uses these ideas to update a new array `g` from the previous `f`, representing the state after taking another step. After `k` steps, `g` represents the possible ways to end in each of the four kinds of cell from the source.

Finally, you look at the relationship between the source and the destination: if they're in the same row or column, there are only certain ways you could have gotten there (`f[0]` and `f[2]` if in the same row, `f[0]` and `f[1]` if in the same column). If they're not, then `f[3]` is the result. You can return this final result as the answer.

Solution Approach

The solution uses dynamic programming, with the core idea being to track how many ways we can be in certain positions on the board after a given number of moves. The positions are categorized into four types, each represented by an array `f` with four elements. A secondary array `g` is introduced to calculate the next state based on the current state of `f`.

Here is the explanation for the code and how it implements the dynamic programming approach:

1. Initialize the modulo variable `mod = 10^9 + 7`, which will be used to keep results within the specified limit by taking the modulo after each computation.

2. The array `f` is initialized with `[1, 0, 0, 0]` since we start at the source and there's exactly one way to be at the source itself without making any move.

3. We loop `k` times, each time calculating a new array `g` based on the current state of `f`. The logic for calculating the next state is based on the movement rules of the problem:

   - `g[0]` updates the number of ways to stay at the source cell, which is calculated from all the ways to come from another column or row to the source cell.
   - `g[1]` computes the ways to be in a different row, same column (excluding the source cell). It includes the ways to come from the source cell, from different rows in the same column and from different columns and rows.
   - `g[2]` is for different column, same row (excluding the source), adding ways from the source cell, different columns in the same row, and different columns and rows.
   - `g[3]` adds ways to be in cells not in the same row or column as the source. This is derived from moving from cells that are in the same column but different rows, same row but different columns, and different rows and columns.
4. The calculations use `(n - 1)`, `(m - 1)`, `(n - 2)`, and `(m - 2)` because these terms represent the number of cells available to move into for each movement category described above (minus the current or source cell).

5. After computing `g`, the current state of `f` is updated to this new state, because `f` needs to represent the state of the number of ways to reach certain positions after an additional move.

6. After all `k` steps are completed, the final result depends on whether `source` and `dest` are in the same row or column. Separate checks are performed:

   - If `source` and `dest` are the exact same cell, then return `f[0]`.
   - If they're in the same row but different cells, return `f[2]`.
   - If they're in the same column but different cells, return `f[1]`.
   - Otherwise, return `f[3]`, which represents reaching a cell that is neither in the same row nor column as the source.

This approach effectively avoids repeating the calculation for each possible path by summarizing the results after each step and updating them iteratively, a hallmark of dynamic programming which makes the solution efficient, even for large `k`.

Example Walkthrough

Let's consider a small example with a 3×3 grid `n = 3`, `m = 3`, a `source` at `(1,1)`, a `dest` at `(3,3)`, and `k = 2` steps.

Let's walk through the two iterations of the loop (since `k = 2`) to understand how the dynamic programming approach calculates the number of ways to reach the destination.

1. **Initialization:** Our initial state `f` is `[1, 0, 0, 0]`, indicating there is 1 way to be in the source without moving.

2. **First iteration:**

   - We create a new array `g` to represent the ways to be at different types of cells after one move.
   - `g[0]`: This is still 0 because there's no way to leave the source cell and return to it in one move.
   - `g[1]`: We can move to any of the two other cells in the same column, so `g[1] = 2*(f[0] + f[3]) = 2` (from the source and from any other different row and column).
   - `g[2]`: Similarly, we can move to any of the two cells in the same row, so `g[2] = 2*(f[0] + f[3]) = 2`.
   - `g[3]`: We cannot reach cells in a different row and column in just one move, so `g[3] = 0`.
   - `f` is now updated to be `g`, so `f = [0, 2, 2, 0]`.
3. **Second iteration:**

   - We again calculate a new `g`.
   - `g[0]`: We can move back to the source cell only from cells in the same row or the same column, `g[0] = f[1] + f[2] = 2 + 2 = 4`.
   - `g[1]`: This includes ways from the source cell, same column, and different row/column, `g[1] = (f[0] + (f[1]*(n - 2)) + f[3]) % mod = (0 + (2 * 1) + 0) % mod = 2`.
   - `g[2]`: For different columns, same row, `g[2] = (f[0] + (f[2]*(m - 2)) + f[3]) % mod = (0 + (2 * 1) + 0) % mod = 2`.
   - `g[3]`: Ways to be not in the same row/column, `g[3] = ((f[1] + f[2])*(n + m - 4) + f[3]*(n * m - n - m + 1)) % mod = (4 * 2 + 0) % mod = 8`.
   - We update `f = [4, 2, 2, 8]`.

At the end of `k` moves, to reach `(3,3)` from `(1,1)`, we need to consider `f[3]` since `dest` is neither in the same row nor column as `source`. Therefore, there are 8 distinct ways to reach the destination in exactly 2 steps. This makes the final answer `f[3] = 8`.

Python Solution

```python
class Solution:
    def number_of_ways(self, rows: int, cols: int, steps: int, source: List[int], destination: List[int]) -> int:
        # Define modulo as per the problem statement to handle large numbers
        mod = 10**9 + 7

        # Initialization of counts for different scenarios - staying, moving in rows, moving in cols, and moving diagonally
        counts = [1, 0, 0, 0]

        # Loop over the number of steps to compute the number of ways dynamically
        for _ in range(steps):
            # Create a new list to store updated counts after each step
            new_counts = [0] * 4

            # Update new counts based on the previous counts
            new_counts[0] = ((rows - 1) * counts[1] + (cols - 1) * counts[2]) % mod
            new_counts[1] = (counts[0] + (rows - 2) * counts[1] + (cols - 1) * counts[3]) % mod
            new_counts[2] = (counts[0] + (cols - 2) * counts[2] + (rows - 1) * counts[3]) % mod
            new_counts[3] = (counts[1] + counts[2] + (rows - 2) * counts[3] + (cols - 2) * counts[3]) % mod

            # Overwrite previous counts with the new computed counts
            counts = new_counts

        # Based on the position of source and destination, return the appropriate count
        if source[0] == destination[0]:
            # If on the same row, return staying count or row move count
            return counts[0] if source[1] == destination[1] else counts[2]
        else:
            # If not on the same row, return column move count or diagonal move count
            return counts[1] if source[1] == destination[1] else counts[3]

# Note: The class expects the `List` to be imported from `typing`, so you should add `from typing import List` at the top of the fi

```

Java Solution

```java
class Solution {
    // Method to calculate the number of ways to reach from source to destination within k steps
    public int numberOfWays(int rows, int cols, int steps, int[] source, int[] dest) {
        final int mod = 1000000007; // Define the modulo value for large number handling
        long[] waysCountByPositionType = new long[4]; // Create an array to hold numbers of ways for the 4 types of positions
        waysCountByPositionType[0] = 1; // Initialize with 1 way to stand still (i.e., 0 step)

        // Loop through the number of steps
        while (steps-- > 0) {
            long[] newWaysCount = new long[4]; // Temp array to hold the new count of ways after each step

            // Calculate number of ways to stand still
            newWaysCount[0] = ((rows - 1) * waysCountByPositionType[1] + (cols - 1) * waysCountByPositionType[2]) % mod;
            // Calculate number of ways for a source placed on the row border, except corners
            newWaysCount[1] = (waysCountByPositionType[0] + (rows - 2) * waysCountByPositionType[1] + (cols - 1) * waysCountByPosit
            // Calculate number of ways for a source placed on the column border, except corners
            newWaysCount[2] = (waysCountByPositionType[0] + (cols - 2) * waysCountByPositionType[2] + (rows - 1) * waysCountByPosit
            // Calculate number of ways for a source placed at the corners
            newWaysCount[3] = (waysCountByPositionType[1] + waysCountByPositionType[2] + (rows - 2) * waysCountByPositionType[3] +

            // After each step, update the count of ways
            waysCountByPositionType = newWaysCount;
        }

        // If source and destination are on the same row
        if (source[0] == dest[0]) {
            // If they are also on the same column, return the count of standing still
            if (source[1] == dest[1]) {
                return (int) waysCountByPositionType[0];
            } else {
                // Otherwise, return the count for column border
                return (int) waysCountByPositionType[2];
            }
        } else {
            // If source and destination are on the same column
            if (source[1] == dest[1]) {
                // Return the count for row border
                return (int) waysCountByPositionType[1];
            } else {
                // Otherwise, return count for corners
                return (int) waysCountByPositionType[3];
            }
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int numberOfWays(int numRows, int numCols, int maxMoves, vector<int>& start, vector<int>& end) {
        const int MOD = 1e9 + 7; // Define the modulus for large numbers

        // 'dp' holds the current number of ways to reach points with various starting and ending constraints
        vector<long long> dp(4);
        dp[0] = 1; // Initialize the first element representing no constraints

        // Iterate 'maxMoves' times applying the transition between states
        while (maxMoves--) {
            // 'nextDp' will hold the next state of our dp array
            vector<long long> nextDp(4);

            nextDp[0] = ((numRows - 1) * dp[1] + (numCols - 1) * dp[2]) % MOD; // Updating with constraints on rows and columns
            nextDp[1] = (dp[0] + (numRows - 2) * dp[1] + (numCols - 1) * dp[3]) % MOD; // Updating with constraint on rows
            nextDp[2] = (dp[0] + (numCols - 2) * dp[2] + (numRows - 1) * dp[3]) % MOD; // Updating with constraint on columns
            nextDp[3] = (dp[1] + dp[2] + (numRows - 2) * dp[3] + (numCols - 2) * dp[3]) % MOD; // Updating with constraints on both

            dp = move(nextDp); // Move to the next state, avoiding copying
        }

        // Check if the starting and ending rows are the same
        if (start[0] == end[0]) {
            // If they are in the same row, check if they are also in the same column
            return start[1] == end[1] ? dp[0] : dp[2];
        }

        // If they are not in the same row, they must be in the same column or different column
        return start[1] == end[1] ? dp[1] : dp[3];
    }
};

```

Typescript Solution

```typescript
const MOD = 1e9 + 7; // Define the modulus for large numbers

let dp: bigint[]; // 'dp' will hold the current number of ways to reach points with various constraints

// Function to update the number of ways to reach given points
function updateDp(numRows: number, numCols: number): bigint[] {
    let nextDp = new Array<bigint>(4);
    nextDp[0] = (((numRows - 1n) * dp[1] + (numCols - 1n) * dp[2]) % BigInt(MOD));
    nextDp[1] = ((dp[0] + (numRows - 2n) * dp[1] + (numCols - 1n) * dp[3]) % BigInt(MOD));
    nextDp[2] = ((dp[0] + (numCols - 2n) * dp[2] + (numRows - 1n) * dp[3]) % BigInt(MOD));
    nextDp[3] = ((dp[1] + dp[2] + (numRows - 2n) * dp[3] + (numCols - 2n) * dp[3]) % BigInt(MOD));
    return nextDp; // Return the updated dp array
}

// Function to calculate the number of ways to move on the grid
function numberOfWays(numRows: number, numCols: number, maxMoves: number, start: number[], end: number[]): bigint {
    dp = new Array<bigint>(4).fill(0n);
    dp[0] = 1n; // Initialize the first element representing no constraints

    // Iterate 'maxMoves' times applying the transition between states
    for (let move = 0; move < maxMoves; move++) {
        dp = updateDp(BigInt(numRows), BigInt(numCols)); // Move to the next state
    }

    // Check if the starting and ending positions are the same (row and column)
    if (start[0] === end[0]) {
        // If in the same row, check if they are also in the same column
        return start[1] === end[1] ? dp[0] : dp[2];
    }

    // If not in the same row, they must be in the same column or a different column
    return start[1] === end[1] ? dp[1] : dp[3];
}

```

Time and Space Complexity

The given Python code calculates the number of ways to reach from `source` to `dest` within `k` moves on an `n * m` grid. Each cell can be visited any number of times, and we can move in four directions: up, down, left, and right.

Time Complexity

The main operation that contributes to the time complexity is the `for` loop, which iterates exactly `k` times, where `k` is the number of moves. The operations inside the loop are constant-time operations because they involve arithmetic operations and assignment, which do not depend on the size of the grid. Therefore, the loop runs `k` times, each with `O(1)` operations, making the overall time complexity `O(k)`.

Space Complexity

Regarding space complexity, we have constant space usage. The function uses a fixed amount of extra space for the variables `f`, `g`, and `mod`, and these do not scale with the input size `n`, `m`, or `k`. Even though `f` and `g` are lists, they always contain exactly four elements. Thus, the space complexity is `O(1)`, which is independent of the input size.