2912. Number of Ways to Reach Destination in the Grid

Hard

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

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.

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

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.

- 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:

3. f[2]: Being in another cell in the same row as the source.

• 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]).

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

• From [13], 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.

cell, from different rows in the same column and from different columns and rows.

column but different rows, same row but different columns, and different rows and columns.

to move into for each movement category described above (minus the current or source cell).

4. f[3]: Being in any cell that is neither in the same row nor column as the source.

- 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: 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.

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

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:

without making any move.

different columns and rows.

Separate checks are performed:

the number of ways to reach the destination.

 \circ f is now updated to be g, so f = [0, 2, 2, 0].

Example Walkthrough

• If source and dest are the exact same cell, then return f[0].

If they're in the same column but different cells, return f[1].

• If they're in the same row but different cells, return f[2].

og[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. og[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

og[2] is for different column, same row (excluding the source), adding ways from the source cell, different columns in the same row, and

og[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

The calculations use (n - 1), (m - 1), (n - 2), and (m - 2) because these terms represent the number of cells available

After all k steps are completed, the final result depends on whether source and dest are in the same row or column.

- 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.
- 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.

Otherwise, return f[3], which represents reaching a cell that is neither in the same row nor column as the source.

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.

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

 \circ 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.

 \circ g[3]: We cannot reach cells in a different row and column in just one move, so g[3] = 0.

First iteration: We create a new array g to represent the ways to be at different types of cells after one move. og[0]: This is still 0 because there's no way to leave the source cell and return to it in one move.

og[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 =

as source. Therefore, there are 8 distinct ways to reach the destination in exactly 2 steps. This makes the final answer f[3] = 8.

∘ 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

Let's walk through the two iterations of the loop (since k = 2) to understand how the dynamic programming approach calculates

 \circ g[2]: For different columns, same row, g[2] = (f[0] + (f[2]*(m - 2)) + f[3]) % mod = (0 + (2 * 1) + 0) % mod = 2. \circ 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.

Python

class Solution:

mod = 10**9 + 7

counts = [1, 0, 0, 0]

counts = new_counts

if source[0] == destination[0]:

Second iteration:

different row and column).

We again calculate a new g.

(0 + (2 * 1) + 0) % mod = 2.

 \circ 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
- Solution Implementation

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

new counts[0] = ((rows - 1) * counts[1] + (cols - 1) * counts[2]) % mod

Based on the position of source and destination, return the appropriate count

If not on the same row, return column move count or diagonal move count

// Method to calculate the number of wavs to reach from source to destination within k steps

int numberOfWays(int numRows, int numCols, int maxMoves, vector<int>& start, vector<int>& end) {

// 'dp' holds the current number of ways to reach points with various starting and ending constraints

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

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

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

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

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

dp[0] = 1; // Initialize the first element representing no constraints

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

nextDp[0] = (((numRows - 1n) * dp[1] + (numCols - 1n) * dp[2]) % BigInt(MOD));

dp[0] = 1n; // Initialize the first element representing no constraints

// Check if the starting and ending positions are the same (row and column)

Define modulo as per the problem statement to handle large numbers

Create a new list to store updated counts after each step

Update new counts based on the previous counts

Loop over the number of steps to compute the number of ways dynamically

new counts[0] = ((rows - 1) * counts[1] + (cols - 1) * counts[2]) % mod

// If in the same row, check if they are also in the same column

// Iterate 'maxMoves' times applying the transition between states

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));

dp = updateDp(BigInt(numRows), BigInt(numCols)); // Move to the next state

nextDp[3] = ((dp[1] + dp[2] + (numRows - 2n) * dp[3] + (numCols - 2n) * dp[3]) % BigInt(MOD));

function numberOfWays(numRows: number, numCols: number, maxMoves: number, start: number[], end: number[]): bigint {

// If they are in the same row, check if they are also in the same column

// If they are not in the same row, they must be in the same column or different column

// Iterate 'maxMoves' times applying the transition between states

// 'nextDp' will hold the next state of our dp array

// Check if the starting and ending rows are the same

return start[1] == end[1] ? dp[0] : dp[2];

return start[1] == end[1] ? dp[1] : dp[3];

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

final int mod = 1000000007; // Define the modulo value for large number handling

public int numberOfWays(int rows, int cols, int steps, int[] source, int[] dest) {

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

 $new_counts[3] = (counts[1] + counts[2] + (rows - 2) * counts[3] + (cols - 2) * counts[3]) % mod$

Update new counts based on the previous counts

Overwrite previous counts with the new computed counts

If on the same row, return staying count or row move count

return counts[0] if source[1] == destination[1] else counts[2]

return counts[1] if source[1] == destination[1] else counts[3]

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 $new_counts = [0] * 4$

Initialization of counts for different scenarios — staying, moving in rows, moving in cols, and moving diagonally

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

else:

class Solution {

class Solution {

vector<long long> dp(4);

if (start[0] == end[0]) {

let nextDp = new Array<bigint>(4);

dp = new Array<bigint>(4).fill(0n);

if (start[0] === end[0]) {

mod = 10**9 + 7

counts = [1, 0, 0, 0]

for in range(steps):

new counts = [0] * 4

class Solution:

Time Complexity

return nextDp; // Return the updated dp array

for (let move = 0; move < maxMoves; move++) {</pre>

// Function to calculate the number of ways to move on the grid

vector<long long> nextDp(4);

while (maxMoves--) {

public:

```
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 wavs 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) * waysCountByPositic
         // Calculate number of ways for a source placed on the column border, except corners
         newWaysCount[2] = (waysCountByPositionType[0] + (cols - 2) * waysCountByPositionType[2] + (rows - 1) * waysCountByPositic
         // Calculate number of ways for a source placed at the corners
         newWaysCount[3] = (waysCountByPositionType[1] + waysCountByPositionType[2] + (rows - 2) * waysCountByPositionType[3] + (countByPositionType[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];
```

long[] waysCountByPositionType = new long[4]; // Create an array to hold numbers of ways for the 4 types of positions

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[] {

};

TypeScript

```
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];
```

def number of ways(self, rows: int, cols: int, steps: int, source: List[int], destination: List[int]) -> int:

```
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 staving 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 file
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

Initialization of counts for different scenarios — staying, moving in rows, moving in cols, and moving diagonally

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 0(1) operations, making the overall time complexity O(k).

can be visited any number of times, and we can move in four directions: up, down, left, and right.