2328. Number of Increasing Paths in a Grid

Hard Depth-First Search Breadth-First Search Graph Topological Sort Memoization Array Dynamic Programming Matrix

Leetcode Link

Problem Description

In this problem, we're presented with a 2D matrix `grid` consisting of integers, with dimensions `m` x `n`. The task is to find the total count of strictly increasing paths through the grid. A path is considered strictly increasing if every consecutive pair of numbers in the path has the latter number greater than the former.

We can start from any cell in the grid and move to any adjacent cell (either up, down, left, or right) as long as we respect the strictly increasing condition. Our goal is to determine all such possible paths in the grid.

A few additional points to consider:

- You can move in one of the four cardinal directions: up, down, left, or right.
- Two paths are unique if they have a different sequence of visited cells.
- Because the number of paths might be quite large, the result should be returned modulo `10**9 + 7`.

Intuition

When faced with problems involving paths, grids, and conditions on movements, it's common to consider depth-first search (DFS) as it allows us to explore all possible paths from a given starting point. DFS is a recursive algorithm that dives deep into a graph (or grid, in this case) to visit nodes and check for the satisfaction of the given condition before backtracking.

The approach hinges on two key insights:

1. **Start from anywhere, end anywhere:** Since we can start and end at any cell, every cell must be considered as a possible start point.
2. **Count each valid path once:** Since moving to an adjacent greater cell constitutes a valid path, from any cell `(i, j)`, we should count all paths starting from it.

To implement this, we arrive at a solution that involves recursive DFS, where for each cell `(i, j)`, the function `dfs(i, j)` calculates the number of strictly increasing paths starting from that cell.

Rather than recalculating paths from each cell multiple times, we employ **memoization** — a technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again.

The core idea behind the given solution is:

- Use DFS to explore all adjacent cells.
- Cache the results to prevent redundant calculations.
- Sum up the counts of increasing paths starting from all cells.
- Return the summed count modulo `10**9 + 7` as per the problem's requirement to deal with large numbers.

In essence, for each cell `(i, j)`, `dfs` is called and checks its 4 neighbors (up, down, left, right). If a neighbor `(x, y)` is in the bounds of the grid and the value at `grid[x][y]` is larger than `grid[i][j]`, it is considered part of a strictly increasing path. The counts from all such neighbors are summed up to give the total count of paths starting from `(i, j)`.

The solution sums the counts from all possible starting points and applies the modulus at each step to keep the numbers in the range, finally giving the aggregated result modulo `10**9 + 7`.

Solution Approach

As outlined previously, the solution uses DFS to explore paths starting from each cell, keeping track of the strictly increasing sequences. In addition, memoization ensures that the same computation is not repeated for a cell, thereby optimizing the process significantly. Let's walk through the implementation:

The `dfs` function is the heart of our solution. It recursively computes the number of increasing paths starting from a cell `(i, j)`. We first check if the answer for this cell is already calculated and stored (memoized). If so, we directly return the stored answer, circumventing repetitive work. This memoization is achieved using the `@cache` decorator, which stores the result of each unique call based on the arguments `(i, j)`.

If the result for a cell `(i, j)` is not memoized, we calculate it by initializing `ans = 1`, accounting for the path that consists only of the starting cell. Then, we move to adjacent cells using computed offsets within the `pairwise` list `(-1, 0, 1, 0, -1)`, ensuring we stay inside the grid bounds. If an adjacent cell `(x, y)` contains a greater integer value than current `(i, j)`, it forms a valid strictly increasing path, so we recursively call `dfs(x, y)` and add the result to `ans`.

To handle the large counts, every addition operation is done modulo `10**9 + 7`. This ensures that intermediate results do not overflow integer limits, while maintaining the final count's accuracy in modular arithmetic.

Finally, to obtain the sum of the paths starting from all cells, we iterate over each cell `(i, j)` of the grid, calling `dfs(i, j)`, and sum the results, applying the modulo operation one last time to get the total count of strictly increasing paths.

The `mod = 10**9 + 7` variable defines the modulo value which is used to keep the results within the specified range throughout the calculations. The `m` and `n` variables store the dimensions of the grid.

Data structures and patterns used in the implementation:

- A 2D list (`grid`) to store the matrix.
- Recursion for DFS.
- A cache (implicit from the `@cache` decorator) to memoize the results for each cell.
- Pairwise iteration (using a pattern with `(-1, 0, 1, 0, -1)`) to get the adjacent cells in the grid.

By leveraging recursion, memoization, and modular arithmetic, the provided solution approach efficiently computes the number of strictly increasing paths in a grid.

Example Walkthrough

Let's take a small 3×3 grid as our example to illustrate the solution approach:

```
grid:
1 2 3
2 1 2
3 1 4
```

We will go through the major steps of DFS and memoization to explain how we determine the total count of strictly increasing paths for each cell in the grid.

1. **Start with the top-left cell (1,1)**:
   - The starting value is 1.
   - We look for strictly increasing numbers among its neighbors `grid[1][2]` (which is 2), and `grid[2][1]` (which is 3).
   - We call `dfs(1,2)` and `dfs(2,1)` to continue the path.
2. **Exploring paths from cell (1,2)** with value 2:
   - Its neighbors are `grid[1][3]` (5), and `grid[2][2]` (2).
   - Since `grid[2][2]` is not greater than 2, it is not considered.
   - We call `dfs(1,3)` to continue the path.
3. **Exploring paths from cell (2,1)** with value 3:
   - The only neighbor greater than 3 is `grid[2][2]` (value 2), which is not strictly increasing and thus this path ends here.
4. **Exploring paths from cell (1,3)** with value 5:
   - There are no neighbors with a greater value, so this path ends here.
5. **Continue the process for each cell**:
   - For this 3×3 grid, we will end up calling the `dfs` function for each cell respecting the strictly increasing condition.
6. **Apply Memoization**:
   - Each calculated path count from `dfs(i, j)` is stored to avoid redundant calculations.
   - When we encounter the same cell `(i, j)` during our DFS, we simply retrieve the stored count from our memoization cache.
7. **Aggregate and Modulo**:
   - The answer for each cell is added to a global counter.
   - Each addition is taken modulo `10**9 + 7` to deal with large numbers.

For this example, let's simply and say that `dfs(1,1)` found 2 paths, `dfs(1,2)` found 1 path, `dfs(1,3)` found 1 path, and so on for the remaining cells (imaginary numbers for the purpose of this example). We then add all these up to find the total number of strictly increasing paths in the grid modulo `10**9 + 7`.

In this manner, by exploring all paths starting from each cell, using DFS and optimizing our process with memoization, we can calculate the total count of strictly increasing paths in the grid. The final result reported will be the sum of the counts from all cells modulo `10**9 + 7`.

Python Solution

```python
from functools import cache  # Import necessary decorator for memoization
from itertools import pairwise  # Import the pairwise utility from itertools

class Solution:
    def countPaths(self, grid: List[List[int]]) -> int:
        # Define the DFS function with memoization to search paths
        @cache
        def dfs(row: int, col: int) -> int:
            # Initialize the number of paths with the current cell (1 path)
            num_paths = 1

            # Define directions for exploring adjacent cells (up, right, down, left)
            directions = (-1, 0, 1, 0, -1)  # (0, -1), (1, 0), (0, 1), (-1, 0)

            # Iterate over each pair of directions using pairwise
            for delta_row, delta_col in pairwise(directions + directions[:1]):
                new_row, new_col = row + delta_row, col + delta_col
                # Check if new cell is in bounds and is greater than current cell
                if 0 <= new_row < rows and 0 <= new_col < cols and grid[new_row][new_col] > grid[row][col]:
                    # If conditions satisfy, recurse on the new cell and update paths count
                    num_paths += dfs(new_row, new_col) % MOD

            # Return the total number of paths from this cell
            return num_paths

        # Define constant MOD to prevent overflow (using modulo operation)
        MOD = 10**9 + 7

        # Get the dimensions of the grid
        m, n = len(grid), len(grid[0])

        # Calculate the sum of paths starting from each cell in the grid
        # The result is the total number of strictly increasing paths
        total_paths = sum(dfs(i, j) for i in range(m) for j in range(n)) % MOD

        # Return the computed total paths
        return total_paths
```

Java Solution

```java
class Solution {
    private int[][] memoization;
    private int[][] grid;
    private int rows, cols;
    private int[] dirs;
    private final int MOD = (int) 1e9 + 7; // Using a constant for the modulus value for all operations

    // Count all possible unique paths from each cell
    public int countPaths(int[][] grid) {
        rows = grid.length;
        cols = grid[0].length;
        this.grid = grid;
        memoization = new int[rows][cols];
        int totalPaths = 0;

        // Iterate over each cell of the grid
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Sum all paths using depth-first search
                totalPaths = (totalPaths + dfs(i, j)) % MOD;
            }
        }
        return totalPaths;
    }

    // Perform a depth-first search to find all unique paths from the current cell
    private int dfs(int row, int col) {
        // Return pre-computed value if already processed
        if (memoization[row][col] != 0) {
            return memoization[row][col];
        }

        // At least one path exists starting from this cell (the cell itself)
        int pathCount = 1;

        // Direction vectors for exploring neighboring cells (up, right, down, left)
        int[] dx = {-1, 0, 1, 0};
        int[] dy = {0, 1, 0, -1};

        // Explore all 4 directions
        for (int direction = 0; direction < 4; ++direction) {
            int newRow = row + dx[direction];
            int newCol = col + dy[direction];
            // Continue DFS if the new cell is within the grid and has a larger value
            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols &&
                grid[newRow][newCol] > grid[row][col]) {
                pathCount = (pathCount + dfs(newRow, newCol)) % MOD; // Add paths from the new cell
            }
        }

        // Cache the result before returning
        return memoization[row][col] = pathCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int countPaths(vector<vector<int>>& grid) {
        const int MOD = 1e9 + 7;
        int m = grid.size(), n = grid[0].size();

        // Initialize memoization table with zeros (dynamic programming cache)
        vector<vector<int>> dp(m, vector<int>(n, 0));

        // Internal recursive depth-first search function with memoization
        function<int(int, int)> dfs = [&](int row, int col) {
            // If the number of paths from this cell has already been computed, return it
            if (dp[row][col]) {
                return dp[row][col];
            }

            // Start with a base case of 1 - the path that includes only the current cell
            int paths = 1;

            // Directions for exploring adjacent cells: up, right, down, left
            int dirs[5] = {-1, 0, 1, 0, -1};

            // Visit all neighboring cells following the increasing-value rule
            for (int k = 0; k < 4; ++k) {
                int x = row + dirs[k], y = col + dirs[k + 1];
                // Check for valid indices and ascending values according to the problem description
                if (x >= 0 && x < m && y >= 0 && y < n && grid[x][y] > grid[row][col]) {
                    paths = (paths + dfs(x, y)) % MOD; // Add the number of paths from the neighbor
                }
            }

            // Memoize the number of paths from this cell
            return dp[row][col] = paths;
        };

        // The final answer to store the sum of all paths in the grid
        int totalPaths = 0;

        // Iterate over each cell of the grid to compute all paths starting from each cell
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                // Sum paths from the current cell to the total paths
                totalPaths = (totalPaths + dfs(i, j)) % MOD;
            }
        }

        // Return the total number of valid paths in the grid, reduced by modulo
        return totalPaths;
    }
};
```

Typescript Solution

```typescript
// Function to count the number of increasing paths in a 2D grid
// Each path moves to an adjacent cell with a strictly larger value.
function countPaths(grid: number[][]): number {
    const MODULO = 1e9 + 7; // The constant to ensure result stays within the bounds
    const rows = grid.length; // Number of rows in the grid
    const cols = grid[0].length; // The number of columns
    const pathCounts = new Array(rows).fill(0).map(() => new Array(cols).fill(0)); // Grid to cache path counts
    const directions: number[] = [-1, 0, 1, 0, -1]; // Direction vectors for exploration

    // Helper function using Depth-First Search to compute increasing path count starting from (i, j)
    const dfs = (i: number, j: number): number => {
        if (pathCounts[i][j]) { // If the path count is already computed for (i, j), return it
            return pathCounts[i][j];
        }
        let pathSum = 1; // Start with a path count of 1 for the current cell
        // Explore all adjacent cells in 4 possible directions (up, right, down, left)
        for (let k = 0; k < 4; ++k) {
            const newR = i + directions[k];
            const newC = j + directions[k + 1];
            // Check if the next cell is within bounds and has a strictly greater value
            if (newR >= 0 && newR < rows && newC >= 0 && newC < cols && grid[i][j] < grid[newR][newC]) {
                pathSum = (pathSum + dfs(newR, newC)) % MODULO; // Recursively compute path count for the next cell
            }
        }
        pathCounts[i][j] = pathSum; // Cache the computed path count for (i, j)
        return pathSum;
    };

    let totalCount = 0; // Total number of increasing paths
    // Loop through all cells in the grid to start dfs computation
    for (let i = 0; i < rows; ++i) {
        for (let j = 0; j < cols; ++j) {
            totalCount = (totalCount + dfs(i, j)) % MODULO; // Add up all path counts using DFS
        }
    }

    return totalCount; // Return the total number of increasing paths
}
```

Time and Space Complexity

The time complexity of this code is not strictly $O(m * n)$ because it employs a recursive depth-first search (DFS) with memoization. Consider that each cell in the grid may potentially be visited from its four neighboring cells, but with memoization, each cell is only computed once. Therefore, the time complexity is $O(m * n * 4)$ due to the DFS traversal, but since we can discount the constant factor, it simplifies to $O(m * n)$.

For space complexity, the memoization `@cache` will at most store a result for each unique call to $dfs(i, j)$, which equates to each cell in the grid, $m * n$. Additionally, the recursive calls add to the stack depth, which in the worst case might involve all cells. Therefore, the space complexity is also $O(m * n)$.