329. Longest Increasing Path in a Matrix

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

Problem Description

This LeetCode problem asks us to find the length of the longest increasing path in a given 2D matrix of integers. The path can be constructed by moving from each cell in one of four directions - left, right, up, or down - but not diagonally. The path must consist of a sequence of numbers where each number is greater than the one before it. As a path cannot wrap around the border of the matrix, once you reach the edge, you cannot go any further in that direction. The goal is to determine the maximum length of such an increasing path.

Intuition

To arrive at the solution, we can use depth-first search (DFS) to explore each possible path from each cell in the matrix. The main task is to traverse the matrix in such a way that we find the longest increasing path starting from every given cell. At each cell `(i, j)`, we look up, down, left, and right for any adjacent cells that have a higher value than the current cell, and continue on from there. Each movement represents a potential path, and we use recursion to explore all paths. We need to keep track of the length of the path we've found, and update the maximum length when we discover a longer path.

To improve the efficiency of our search, we can use memoization to store the results of the subproblems that have already been computed. In this code, `dfs(i, j)` represents the length of the longest path starting from cell `(i, j)`. The `@cache` decorator from Python's `functools` module ensures that once we compute `dfs(i, j)` for a particular cell `(i, j)`, we do not recompute it again - the result is stored and reused.

We iterate through every cell in the matrix, invoking `dfs(i, j)` to get the length of the longest increasing path starting from that cell, keeping track of the maximum value as we go. By the end of this process, we will have explored all the possible paths and found the longest possible path length in the matrix, which will be our final answer.

Solution Approach

The solution uses depth-first search (DFS) along with memoization to efficiently find the longest increasing path. Here is a step-by-step explanation of how the solution approach is implemented:

- We begin by defining a `dfs` function that takes the coordinates `(i, j)` of a cell as arguments. This function will return the length of the longest increasing path starting from this cell. To remember the length of paths we've already found, we decorate this function with the `@cache` decorator, enabling memoization.
- Within the `dfs` function, we initialize a variable `ans` to 0 to keep track of the longest path found starting from the given cell.
- We create a loop that iterates over the adjacent cells to the current cell `(i, j)`. This loop uses `pairwise` to iterate through the directional offsets `(-1, 0, 1, 0, -1)` to access the left, right, up, and down neighboring cells by adding these offsets to the current cell's indices.
- For each neighboring cell `(x, y)`, we check if the cell is within the matrix bounds and if its value is greater than the current cell's value, which would make it a candidate for the increasing path.
- If the adjacent cell `(x, y)` meets the conditions, we call `dfs(x, y)` and update `ans` with the maximum value between the current `ans` and the result of `dfs(x, y)`.
- After exploring all possible paths from the current cell, we return `ans + 1`, with the `+ 1` accounting for the length of the path including the current cell.
- Outside of the `dfs` function, we determine the dimensions of the matrix `m` and `n`. We then iterate through every cell in the matrix, calling `dfs(i, j)` for each and finding the maximum value across all cells. This maximum value is the length of the longest increasing path in the matrix.
- We finally return this maximum value as the solution to the problem.

The application of DFS, along with memoization via the `@cache` decorator, allows us to efficiently avoid recomputation of paths that have already been computed. This makes the algorithm significantly faster, especially for larger matrices, as we are able to reuse previously computed results for the subproblems.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have the following 2D matrix:

```
1 Matrix:
1 2 4
3 2 5
```

We want to find the length of the longest increasing path in this matrix.

Following the solution approach, we apply DFS and memoization as outlined:

1. We define our `dfs` function with memoization to keep track of the longest path starting from each cell.
2. We start with the cell at `(0, 0)` which has the value 3. We look at its neighbors `(0, 1)` with value 4.
3. Since 4 is greater than 3, it's a valid step in the increasing path. We call `dfs(0, 1)`.
4. At `(0, 1)`, with the value 4, the only valid increasing step is to move down to `(1, 1)` with the value 5. We call `dfs(1, 1)`.
5. At `(1, 1)`, with the value 5, there are no further adjacent cells with larger values, so its longest path is just itself, with a length of 1. The value is memoized.
6. Returning to `(0, 1)` with value 4, we add 1 for the current cell to the maximum path length we can get by moving from it, which is 1 for the `(1, 1)` cell. The total is 2, which is memoized for `(0, 1)`.
7. Now we go back to the starting cell `(0, 0)`. With the memoized results, we know the longest path from `(0, 1)` is of length 2. So for `(0, 0)`, the length is 2 plus the current cell: 3.
8. Next, we check the cell `(1, 0)`.
9. The cell `(1, 0)` has a value of 2 and only one neighbor `(1, 1)` with a greater value. We call `dfs(1, 1)`, but this time, the path length is already memoized.
10. For `(1, 0)`, we take the memoized path length of `(1, 1)`, which is 1, and add 1 for the current cell, giving us a total path length of 2.
11. Therefore, the possible path lengths starting from each cell are 3 from `(0, 0)` and 2 from both `(0, 1)` and `(1, 0)`. The cell at `(1, 1)` does not lead to any other cell, so its path length is 1.
12. We iterate through each cell in the matrix and choose the maximum length from the results of the `dfs` calls. The final result for the longest increasing path in this example is 3, which starts from cell `(0, 0)` and goes through `(0, 1)` to `(1, 1)`.

Applying the steps from the solution to this small example matrix demonstrates the increase in efficiency gained from memoization, as we avoid recalculating the longest path from each cell more than once.

Python Solution

```python
from typing import List
from functools import lru_cache

class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        # Define the dimensions of the matrix
        rows, cols = len(matrix), len(matrix[0])

        # Define direction vectors for traversing up, down, left, and right
        directions = [-1, 0, 1, 0, -1]  # Up, Right, Down, Left

        # Use memoization to cache the results of previous computations
        @lru_cache(maxsize=None)
        def dfs(row: int, col: int) -> int:
            # Start with the longest path of 1 that includes the current cell
            max_path_length = 1

            # Explore all possible directions from the current cell
            for dir_row, dir_col in pairwise(directions):
                new_row, new_col = row + dir_row, col + dir_col
                # Check if the new position is within bounds and increases in value
                if 0 <= new_row < rows and 0 <= new_col < cols and matrix[new_row][new_col] > matrix[row][col]:
                    # Update the longest path length if moving to the new position
                    max_path_length = max(max_path_length, 1 + dfs(new_row, new_col))

            return max_path_length

        # Iterate over all matrix cells to find the starting point of the longest increasing path
        max_path = 0
        for i in range(rows):
            for j in range(cols):
                max_path = max(max_path, dfs(i, j))

        return max_path

# Example usage
# solution = Solution()
# print(solution.longestIncreasingPath([[9,9,4],[6,6,8],[2,1,1]]))  # Output: 4
```

Java Solution

```java
class Solution {
    private int numRows;
    private int numCols;
    private int[][] grid; // Renamed "matrix" to "grid" for clarity
    private int[][] dp; // Renamed "f" to "dp" for denoting the "dynamic programming" cache

    public int longestIncreasingPath(int[][] matrix) {
        numRows = matrix.length;
        numCols = matrix[0].length;
        grid = matrix; // Assigning the matrix to grid
        dp = new int[numRows][numCols];
        int longestPath = 0;

        // Iterate over each cell in the matrix
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                // Update the longest path found so far
                longestPath = Math.max(longestPath, dfs(row, col));
            }
        }

        return longestPath;
    }

    private int dfs(int row, int col) {
        // If the current cell already contains a computed length, return it
        if (dp[row][col] != 0) {
            return dp[row][col];
        }

        // Direction vectors for the 4 adjacent cells (up, right, down, left)
        int[] directions = {-1, 0, 1, 0, -1};

        // Explore all adjacent cells
        for (int k = 0; k < 4; ++k) {
            int nextRow = row + directions[k];
            int nextCol = col + directions[k + 1];

            // Check if the next cell is within bounds and has a larger value than the current cell
            if (nextRow >= 0 && nextRow < numRows && nextCol >= 0 && nextCol < numCols && grid[nextRow][nextCol] > grid[row][col]) {
                // Update the length of the longest increasing path from the current cell
                dp[row][col] = Math.max(dp[row][col], dfs(nextRow, nextCol));
            }
        }

        // Increment the stored length by 1 to include the current cell and return it
        return ++dp[row][col];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        int rows = matrix.size(), cols = matrix[0].size();
        int directions[5] = {-1, 0, 1, 0, -1}; // Directions for moving in the matrix
        vector<vector<int>> cache(rows, vector<int>(cols, 0)); // Memoization cache
        int longestPath = 0; // Initialize the length of the longest path
        // Directions array representing the four possible movements (up, right, down, left)
        vector<int> directions = {-1, 0, 1, 0, -1};

        // Depth-first search function to find the longest increasing path
        function<int(int, int)> dfs = [&](int row, int col) -> int {
            // If we've already computed this position, return the cached value
            if (cache[row][col]) {
                return cache[row][col];
            }
            // Check all four possible directions
            for (int k = 0; k < 4; ++k) {
                int newRow = row + directions[k]; // Calculate new row position
                int newCol = col + directions[k + 1]; // Calculate new column position
                // If the new position is within bounds and is greater than the current cell
                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && matrix[newRow][newCol] > matrix[row][col]) {
                    // Recursively explore the new position and update the cache
                    cache[row][col] = max(cache[row][col], dfs(newRow, newCol));
                }
            }
            // After exploring all directions, increment the path length by 1 and return
            return ++cache[row][col];
        };

        // Iterate through each cell in the matrix
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Compute the longest increasing path starting from (i, j)
                longestPath = max(longestPath, dfs(i, j));
            }
        }
        // Return the length of the longest path after processing all cells
        return longestPath;
    }
};
```

Typescript Solution

```typescript
// Type definition for the matrix of numbers.
type Matrix = number[][];

// Cache for storing the length of the longest increasing path from a particular cell.
let cache: Matrix;

// Directions array representing the movement offsets for the four cardinal directions.
const directions: number[] = [
    -1, 0, // up
    0, 1,  // right
    1, 0,  // down
    0, -1, // left
];

/**
 * Performs a depth-first search to find the longest increasing path from a given cell.
 * @param row - The row index of the current cell.
 * @param column - The column index of the current cell.
 * @param matrix - The input matrix of numbers.
 * @returns The length of the longest increasing path from the current cell.
 */
function dfs(row: number, column: number, matrix: Matrix): number {
    // If we already computed the longest path from this cell, return the cached value.
    if (cache[row][column] > 0) {
        return cache[row][column];
    }

    // Initialize the maximum length from the current cell to 1 (the cell itself).
    let maxLength = 1;

    // Explore all possible directions from the current cell.
    for (const [dx, dy] of directions.pairwise()) {
        const newRow = row + dx;
        const newColumn = column + dy;

        // Check if the new cell is within bounds and has a value greater than the current cell.
        if (
            newRow >= 0 && newRow < matrix.length &&
            newColumn >= 0 && newColumn < matrix[0].length &&
            matrix[newRow][newColumn] > matrix[row][column]
        ) {
            // Recursively compute the longest path from the neighboring cell.
            const length = 1 + dfs(newRow, newColumn, matrix);

            // Update the maximum length if the computed path is longer.
            maxLength = Math.max(maxLength, length);
        }
    }

    // Cache the computed maximum length for the current cell.
    cache[row][column] = maxLength;

    return maxLength;
}

/**
 * Computes the length of the longest increasing path in a matrix.
 * @param matrix - The input matrix of numbers.
 * @returns The length of the longest increasing path in the matrix.
 */
function longestIncreasingPath(matrix: Matrix): number {
    // Handle an empty matrix case by returning 0 early.
    if (matrix.length === 0 || matrix[0].length === 0) {
        return 0;
    }

    // Initialize rows and columns counts, and the cache matrix.
    const rows = matrix.length;
    const columns = matrix[0].length;
    cache = Array.from({ length: rows }, () => Array(columns).fill(0));

    // The overall maximum length of the longest increasing path.
    let globalMaxLength = 0;

    // Iterate over each cell in the matrix.
    for (let i = 0; i < rows; ++i) {
        for (let j = 0; j < columns; ++j) {
            // Use DFS to find the longest path from the current cell and update the global maximum.
            globalMaxLength = Math.max(globalMaxLength, dfs(i, j, matrix));
        }
    }

    // Return the global maximum length found.
    return globalMaxLength;
}
```

Time and Space Complexity

Time Complexity

The time complexity of this algorithm is $O(m * n)$ where m is the number of rows and n is the number of columns in the input matrix. This is because in the worst case scenario, we perform a depth-first search (DFS) starting from each cell in the matrix. The memoization using `@cache` ensures that each cell `(i, j)` is only processed once for its longest increasing path, and the DFS from that cell terminates once it has explored all increasing paths from that cell. Hence, each cell's DFS contributes $O(1)$ amortized time, and since we have $m * n$ cells, the overall time is $O(m * n)$.

Space Complexity

The algorithm has a space complexity of $O(m * n)$ as well, due to the memoization cache that potentially stores the results for each cell in the matrix. The space required is for the recursive call stack and the cache storage. In the worst case, the recursive call stack could go as deep as $m * n$ if there is an increasing path through every cell. Thus, with the cache and call stack combined, the space complexity is $O(m * n)$.