2713. Maximum Strictly Increasing Cells in a Matrix

Hard | Memoization | Array | Binary Search | Dynamic Programming | Matrix | Sorting | Leetcode Link

Problem Description

In this problem, we are given a 2D integer matrix where each index is 1-indexed, meaning that the top-left corner of the matrix starts at 1,1 instead of the usual 0,0 in programming. The challenge is to start from any cell within this matrix and move to any other cell in the same row or column. However, there is a condition for movement: you can only move to a cell if its value is strictly greater than the value of the current cell. This process continues until there are no more valid moves left.

The objective is to find the longest possible path through the matrix following the movement rule, and to return the number of cells visited during this path. This maximum number should represent the greatest possible number of cells one could visit starting from any cell in the matrix.

Intuition

The intuition behind the solution is to leverage the fact that we can only move to strictly greater cells. This imposes a natural order of movement from smaller to larger values. Therefore, all potential paths will have an increasing trend.

The solution approach involves dynamic programming; specifically, we follow these steps:

1. Use a graph model where each node represents a cell (i, j) and its value mat[i][j]. We must identify all potential moves from each cell.
2. Sort the nodes based on their values. Since we move from smaller to larger values, processing the nodes in increasing order ensures that by the time we process a cell, all potential moves into this cell have already been considered.
3. For all cells with the same value (possible since values may repeat), compute the length of the longest path ending in each cell. We can do this efficiently by keeping track of the maximum path length seen so far in each row and column.
4. Update the global maximum path length as the maximum of the result for each cell.

By processing cells in non-decreasing order and updating row and column maximums, we ensure that every possible move is accounted for, and we avoid double-counting. This way, we arrive at the optimal solution which is the maximum number of cells one can visit.

Solution Approach

The implementation of the solution follows these steps:

1. **Graph Representation**: The matrix is conceptualized as a graph where each cell is a node that can potentially connect to other nodes in its row and column if they contain larger values.

2. **Preprocessing**: A dictionary, `g`, is created to group cells by their values, with each value as a key and a list of tuples (coordinates of the cells) as the value.

```python
g = defaultdict(list)
for i in range(m):
    for j in range(n):
        g[mat[i][j]].append((i, j))
```

3. **Higher-Level Dynamic Programming Storage**: Two lists, `rowMax` and `colMax`, are initialized to keep track of the maximum path length seen so far for each row and column, respectively.

```python
rowMax = [0] * m
colMax = [0] * n
```

4. **Iterate Over Cells by Increasing Values**: Sort the keys of `g` which represent cell values to process cells in non-decreasing order.

```python
for _, pos in sorted(g.items()):
```

5. **Local Dynamic Programming Computation**: As each cell is processed, the maximum length of the path ending at that cell is computed using the previously stored values in `rowMax` and `colMax`.

```python
mx = []  # Local maximum path lengths for cells with the same value.
for i, j in pos:
    mx.append(1 + max(rowMax[i], colMax[j]))
    ans = max(ans, mx[-1])
```

6. **Global Maximum Path Update**: After processing cells of the same value, update `rowMax` and `colMax` with the new maximum path lengths.

```python
for k, (i, j) in enumerate(pos):
    rowMax[i] = max(rowMax[i], mx[k])
    colMax[j] = max(colMax[j], mx[k])
```

7. **Result**: Once all cells have been processed, `ans` will contain the maximum number of cells that can be visited starting from any cell in the matrix, which is then returned.

The dynamic programming pattern leveraged in "Local Dynamic Programming Computation" and "Global Maximum Path Update" is crucial as it guarantees the path length is calculated correctly while ensuring that each cell is considered once, and in the right order. This allows for the solution to have a time complexity of `O(m * n * log(m * n))` due to the sorting step, where `m` and `n` are the number of rows and columns of the matrix, respectively.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose our matrix is:

```
3 4 5
3 2 6
2 2 1
```

This is a 3×3 matrix with the top-left cell starting at index 1,1. Now, let's apply the solution approach:

1. **Graph Representation**: Each cell can be thought of as a node. We will move only to nodes with larger values.

2. **Preprocessing**: We create a dictionary to group cells by their values.
   - For the value 1: `g[1] = [(2, 2)]`
   - For the value 2: `g[2] = [(2, 0), (2, 1)]`
   - For the value 3: `g[3] = [(0, 0), (1, 0)]`
   - For the value 4: `g[4] = [(0, 1)]`
   - For the value 5: `g[5] = [(0, 2)]`
   - For the value 6: `g[6] = [(1, 2)]`
3. **Higher-Level Dynamic Programming Storage**: Initialize the `rowMax` and `colMax`.
   - `rowMax = [0, 0, 0]` for three rows
   - `colMax = [0, 0, 0]` for three columns
4. **Iterate Over Cells by Increasing Values**: We process the list of cell positions in ascending order of their values.

5. **Local Dynamic Programming Computation**: Calculate the maximum path length for each cell.
   - Starting with value 1: `g[1] = [(2, 2)]`, we have no prior larger values, so the length is 1 for `rowMax[2]` and `colMax[2]`.
   - For value 2: `g[2] = [(2, 0), (2, 1)]`, since they follow value 1, the length for both is 1. Update `rowMax[2]` to 1.
   - For value 3: `g[3] = [(0, 0), (1, 0)]`, no larger prior values are in the same row or column. The length remains 1 and updates `rowMax[0]` and `rowMax[1]`.
   - For value 4: `g[4] = [(0, 1)]`, it can only come from `(0, 0)` which has a value 3. The length is `1 + rowMax[0]`, so 2. Update `colMax[1]` with 2.
   - For value 5: `g[5] = [(0, 2)]`, could come from `(0, 1)` which has value 4. The length becomes `1 + colMax[1]`, so 3. Update `colMax[2]` with 3.
   - For value 6: `g[6] = [(1, 2)]`, could have moved from `(0, 2)` with value 5. Therefore, length is `1 + colMax[2]`, resulting in 4. This is now our `ans`.
6. **Global Maximum Path Update**: After processing, `rowMax` and `colMax` have been updated with the maximum path lengths as shown during the computation.

7. **Result**: The maximum number of cells visited is 4, starting from cell `(1, 2)` (1-indexed) and moving through cells `(0, 2)`, `(0, 1)`, and `(0, 0)` in that order, following increasing cell values.

The careful consideration of cell values and ensuring we process them from smallest to largest while updating path lengths as we go allows us to determine the maximum visitable path efficiently.

Python Solution

```python
from collections import defaultdict

class Solution:
    def max_increasing_cells(self, matrix):
        # Determine the size of the matrix
        rows, cols = len(matrix), len(matrix[0])

        # Create a dictionary to hold cell values and their positions
        value_to_positions = defaultdict(list)

        # Populate the dictionary with positions for each value in the matrix
        for i in range(rows):
            for j in range(cols):
                value_to_positions[matrix[i][j]].append((i, j))

        # Initialize arrays to keep track of maximum increasing path length ending at any row or column
        max_row_length = [0] * rows
        max_col_length = [0] * cols

        # Variable to keep track of the maximum length of increasing cells found
        max_length = 0

        # Process cells in ascending order of their values
        for _, positions in sorted(value_to_positions.items()):
            current_max = []
            # Calculate the maximum length for the current set of cells with the same value
            for i, j in positions:
                increase_amount = 1 + max(max_row_length[i], max_col_length[j])
                current_max.append(increase_amount)
                # Update the overall max_length
                max_length = max(max_length, increase_amount)

            # Update the max_row_length and max_col_length for the current set of cells
            for index, (i, j) in enumerate(positions):
                max_row_length[i] = max(max_row_length[i], current_max[index])
                max_col_length[j] = max(max_col_length[j], current_max[index])

        # Return the maximum length of the increasing cells path found
        return max_length
```

Java Solution

```java
class Solution {
    public int maxIncreasingCells(int[][] mat) {
        // Dimensions of the matrix
        int rows = mat.length;
        int cols = mat[0].length;

        // TreeMap to organize cells by their values (handles sorting automatically)
        TreeMap<Integer, List<int[]>> valueToCellsMap = new TreeMap<>();

        // Fill the TreeMap with cells, where each cell is represented by its coordinates [i, j]
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                valueToCellsMap.computeIfAbsent(mat[i][j], k -> new ArrayList<>()).add(new int[]{i, j});
            }
        }

        // Arrays to track the maximum increase count per row and column
        int[] rowMaxIncrements = new int[rows];
        int[] colMaxIncrements = new int[cols];

        // Variable to store the final answer, which is the maximum length of increasing path
        int maxLengthOfIncreasingPath = 0;

        // Iterate through the TreeMap, processing cells in ascending order of their values
        for (var entry : valueToCellsMap.entrySet()) {
            var cells = entry.getValue();
            int[] localMaxIncrements = new int[cells.size()];
            int index = 0; // Used to update localMaxIncrements as we iterate through cells

            // Determine the maximum increasing path ending at each cell based on previous rows/cols
            for (var cell : cells) {
                int row = cell[0];
                int col = cell[1];
                localMaxIncrements[index] = Math.max(rowMaxIncrements[row], colMaxIncrements[col]) + 1;
                maxLengthOfIncreasingPath = Math.max(maxLengthOfIncreasingPath, localMaxIncrements[index]);
                index++;
            }

            // Update the rowMaxIncrements and colMaxIncrements arrays based on the new localMaxIncrements
            for (index = 0; index < localMaxIncrements.length; ++index) {
                int rowToUpdate = cells.get(index)[0];
                int colToUpdate = cells.get(index)[1];
                rowMaxIncrements[rowToUpdate] = Math.max(rowMaxIncrements[rowToUpdate], localMaxIncrements[index]);
                colMaxIncrements[colToUpdate] = Math.max(colMaxIncrements[colToUpdate], localMaxIncrements[index]);
            }
        }
        // Return the maximum length of the increasing path found
        return maxLengthOfIncreasingPath;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the length of the longest strictly increasing path in the matrix
    int maxIncreasingCells(vector<vector<int>>& matrix) {
        int rows = matrix.size();         // Number of rows in the matrix
        int cols = matrix[0].size();      // Number of columns in the matrix

        // Map to hold the matrix values as keys and their corresponding positions as values
        map<int, vector<pair<int, int>>> valueToPositions;

        // Populate the map with the matrix values and their positions
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                valueToPositions[matrix[i][j]].emplace_back(i, j);
            }
        }

        // Vectors to keep track of the max path length for each row and column
        vector<int> rowMax(rows, 0);
        vector<int> colMax(cols, 0);

        int maxLength = 0; // Variable to store the length of the longest path found

        // Process each group of positions belonging to the same value in increasing order
        for (auto& valueAndPositions : valueToPositions) {
            vector<int> currentMaxLengths;
            for (auto& [row, col] : valueAndPositions.second) {
                // Calculate the max path length at the current position
                currentMaxLengths.push_back(max(rowMax[row], colMax[col]) + 1);
                // Update the overall max length found so far
                maxLength = max(maxLength, currentMaxLengths.back());
            }

            // Update rowMax and colMax with the new max lengths
            for (int k = 0; k < currentMaxLengths.size(); ++k) {
                auto& [row, col] = valueAndPositions.second[k];
                rowMax[row] = max(rowMax[row], currentMaxLengths[k]);
                colMax[col] = max(colMax[col], currentMaxLengths[k]);
            }
        }

        return maxLength;  // Return the length of the longest strictly increasing path
    }
};
```

Typescript Solution

```typescript
// Define the type alias for a matrix
type Matrix = number[][];

// Define the type alias for positions which is an array of tuples with row and column
type Positions = Array<[number, number]>;

// Function to find the length of the longest strictly increasing path in the matrix
function maxIncreasingCells(matrix: Matrix): number {
    const rows = matrix.length;                       // Number of rows in the matrix
    const cols = matrix[0].length;                    // Number of columns in the matrix

    // Map to hold the matrix values as keys and their corresponding positions as values
    const valueToPositions = new Map<number, Positions>();

    // Populate the map with the matrix values and their positions
    for (let i = 0; i < rows; ++i) {
        for (let j = 0; j < cols; ++j) {
            const value = matrix[i][j];
            if (!valueToPositions.has(value)) {
                valueToPositions.set(value, []);
            }
            valueToPositions.get(value)!.push([i, j]);
        }
    }

    // Arrays to keep track of the max path length for each row and column
    const rowMax: number[] = new Array(rows).fill(0);
    const colMax: number[] = new Array(cols).fill(0);

    let maxLength = 0;  // Variable to store the length of the longest path found

    // Process each group of positions belonging to the same value in increasing order of values
    Array.from(valueToPositions.keys())
        .sort((a, b) => a - b) // Ensure that keys/values are processed in sorted order
        .forEach(value => {
            const positions = valueToPositions.get(value)!;
            const currentMaxLengths: number[] = [];

            positions.forEach(([row, col]) => {
                // Calculate the max path length at the current position
                const currentLength = Math.max(rowMax[row], colMax[col]) + 1;
                currentMaxLengths.push(currentLength);
                // Update the overall max length found so far
                maxLength = Math.max(maxLength, currentLength);
            });

            // Update rowMax and colMax with the new max lengths
            positions.forEach(([row, col], k) => {
                rowMax[row] = Math.max(rowMax[row], currentMaxLengths[k]);
                colMax[col] = Math.max(colMax[col], currentMaxLengths[k]);
            });
        });

    return maxLength;  // Return the length of the longest strictly increasing path
}
```

Time and Space Complexity

The time complexity of the given code can be broken down into a few parts:

1. Building the graph `g`: We iterate over all the elements in the `mat`, which has a size of `m*n`. For each element, we perform a constant-time operation (`append`). Thus, this part of the algorithm runs in `O(m*n)` time.

2. Sorting the keys of the graph `g`: Since `g` can have at most `m*n` unique keys (in the worst case where all elements are unique), the time complexity of sorting these keys is `O(m*n*log(m*n))`.

3. Calculating the maximum increasing path: We iterate again over the entries sorted in `g`. For each value, we iterate over all positions with that value and update `rowMax` and `colMax`. The inner loop will run at most `m*n` times in total. For each inner loop, updating `rowMax` and `colMax` is a constant-time operation. Hence, the time complexity for this part is `O(m*n)`.

Combining all the steps, the total time complexity of the code is dominated by the sorting step, so it is `O(m*n*log(m*n))`.

The space complexity of the given code can be analyzed as follows:

1. The graph `g`, which in the worst case, contains `m*n` keys with a single-value list, leading to a space complexity of `O(m*n)`.

2. The `rowMax` and `colMax` arrays, which add up to a space complexity of `O(m + n)`.

3. The `mx` list, which within a single iteration over `g`'s keys, in the worst case, might store up to `min(m, n)` elements (in the case of all positions having the same value), so its space complexity is `O(min(m, n))`.

Therefore, the total space complexity, considering the space required for input and auxiliary space, is `O(m*n + m + n)`, which simplifies to `O(m*n)` as it is the dominating term.