1504. Count Submatrices With All Ones

Medium Stack Array Dynamic Programming Matrix Monotonic Stack Leetcode Link

Problem Description

The problem presents a `m x n` binary matrix `mat`, where `m` represents the number of rows and `n` represents the number of columns. A binary matrix is a matrix where each element is either `0` or `1`. Our task is to count the number of unique submatrices within this matrix where every element is `1`. A submatrix is defined as any contiguous block of cells within the larger matrix, and it could be as small as a single cell or as large as the entire matrix, as long as all its elements are '1's.

Intuition

To solve this problem, an intuitive approach is to look at each cell and ask, "How many submatrices, for which this cell is the bottom-right corner, contain all ones?" By answering this question for every cell, we can sum these counts to find the total number of submatrices that have all ones.

To efficiently compute the count for each cell, we can use a dynamic programming approach, where we keep a running count of consecutive '1's in each row up to and including the current cell. This count will reset to 0 whenever we encounter a '0' as we go left to right in each row. This helps because the number of submatrices for which a cell is the bottom-right corner depends on the number of consecutive '1's to the left of that cell (in the same row) and above it (in the same column).

With this running count, we iterate over each cell (let's say at position `(i, j)`), and for each cell we look upwards, tracking the minimum number of consecutive '1's seen so far in the column. The number of submatrices that use the cell `(i, j)` as the bottom-right corner is the sum of these minimum values, because the submatrix size is constrained by the smallest count of consecutive '1's in any row above `(i, j)` including the row containing `(i, j)`.

By incrementing our answer with these counts for each cell, we end up with the total number of submatrices that have all ones.

Solution Approach

The solution provided is a dynamic programming approach that optimizes the brute force method. The code uses an auxiliary grid `g` with the same dimensions as the input matrix `mat`. Here's a step-by-step breakdown of the approach:

1. **Create a running count grid**: We initialize a grid `g` such that `g[i][j]` represents the number of consecutive '1's ending at position `(i, j)` in the matrix `mat`. This is calculated by iterating through each row of `mat`. If `mat[i][j]` is a '1', then `g[i][j]` is set to `1 + g[i][j - 1]` if `j` is not `0`, otherwise it's set to `1`. If `mat[i][j]` is '0', `g[i][j]` remains `0`.

2. **Iterate over all cells to count submatrices**: For every cell `(i, j)` in the matrix, we perform the following steps:

   - Initialize `col` to `infinity` to keep track of the smallest number of consecutive '1's found so far in the current column.
   - Move upwards from the current cell `(i, j)` to the top of the column `(0, j)`, updating `col` to be the minimum of its current value and `g[k][j]` where `k` ranges from `i` to `0`. This represents the number of consecutive '1's in column `j` at row `k`.
   - Add the value of `col` to `ans` for every row above (and including) the current cell. This adds the number of submatrices where cell `(i, j)` is the bottom-right corner, as explained in the intuition.

The data structures used are:

- An input matrix `mat` of size `m x n`, where `mat[i][j]` represents a cell in the original binary matrix.
- An auxiliary grid `g` of the same size as `mat`, where `g[i][j]` stores the number of consecutive '1's to the left of `mat[i][j]`.

The patterns and algorithms used include dynamic programming, to keep track of the running count of '1's in a row (state), and for each state, we calculate the maximum number of submatrices that can be formed using the bottom edge of that submatrix. This involves iterating upwards and finding the width (consecutive '1's) that is limiting the size of the submatrix.

In terms of complexity, the algorithm uses `O(m*n)` space for the grid `g` and `O(m^2 * n)` time, since it iterates through the matrix and for each cell, it potentially iterates through `m` elements above it in the column.

Example Walkthrough

Let's demonstrate the solution approach using a small example with a 3×3 binary matrix `mat` as input:

```
mat = [
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 0]
]
```

Following the dynamic programming approach, here's how we would walk through this instance step-by-step.

1. **Create a running count grid (`g`)**: We go through each row and build our auxiliary grid `g`. For the given matrix `mat`, `g` would look like this:

```
g = [
    [1, 0, 1],
    [1, 2, 3],
    [0, 1, 0]
]
```

Element `g[1][2]` is `3` because there are three consecutive `1`s ending at `mat[1][2]`.

2. **Iterate over all cells to count submatrices**:
   - For cell `(0, 0)` in `mat`, the count of submatrices for which it is the bottom-right corner is just `1`. Update `ans = 1`.
   - Cell `(0, 1)` is `0`, so it cannot be the bottom-right corner of any submatrix with all `1`s. `ans` remains `1`.
   - Cell `(0, 2)` can only be the bottom-right corner of a submatrix with size 1×1. Update `ans = 1 + 1 = 2`.
   - Cell `(1, 0)` can be the bottom-right corner of two submatrices: two of size 1×1 (`mat[1][0]` and `mat[0][0]`), so update `ans = 2 + 2 = 4`.
   - Cell `(1, 1)` is interesting; it can be the bottom-right corner of one submatrix of size 2×2, two submatrices of size 2×1 (above it), and an additional two submatrices of size 1×1 (same row). Update `ans = 4 + 5 = 9`.
   - For cell `(1, 2)`, the limiting factor is the `0` in cell `(0, 1)`. Thus, it can only form one submatrix of size 1×3 and three submatrices of size 1×1. Hence, we update `ans = 9 + 4 = 13`.
   - Skip cell `(2, 0)` as it is `0`.
   - Cell `(2, 1)` can form one submatrix of size 1×1. Update `ans = 13 + 1 = 14`.
   - Skip cell `(2, 2)` as it is `0`.

Thus, the final answer, the number of unique submatrices where every element is `1`, is `14` for the provided `mat` matrix.

In terms of data structures and patterns:

- The input matrix `mat` represents the original binary matrix.
- The auxiliary grid `g` helps to calculate how many `1`s we have consecutively to the left of a given cell, including the cell itself, which saves us time when checking the possibilities of forming submatrices.

The space complexity remains `O(m*n)` which is required for storing the grid `g`, and the time complexity is `O(m^2 * n)` due to the nested iterations used for each element to consider the cells above it. For larger matrices, this complexity is something to be aware of as it could become a performance bottleneck.

Python Solution

```python
class Solution:
    def numSubmat(self, mat: List[List[int]]) -> int:
        # Get the number of rows (m) and columns (n) of the matrix.
        num_rows, num_cols = len(mat), len(mat[0])

        # Initialize a matrix to store the number of continuous ones in each row.
        continuous_ones = [[0] * num_cols for _ in range(num_rows)]

        # Populate the continuous_ones matrix.
        for row in range(num_rows):
            for col in range(num_cols):
                # If we encounter a '1', count the continuous ones. If in first column,
                # it can only be 1 or 0. Otherwise, add 1 to the left cell's count.
                if mat[row][col]:
                    continuous_ones[row][col] = 1 if col == 0 else 1 + continuous_ones[row][col - 1]

        # Initialize a count for total number of submatrices.
        total_submatrices = 0

        # Calculate the number of submatrices for each cell as the bottom-right corner.
        for row in range(num_rows):
            for col in range(num_cols):
                # This will keep track of the smallest number of continuous ones in the current column,
                # up to the current row 'row'.
                min_continuous_ones = float('inf')

                # Travel up the rows from the current cell.
                for k in range(row, -1, -1):
                    # Find the smallest number of continuous ones in this column up to the k-th row.
                    min_continuous_ones = min(min_continuous_ones, continuous_ones[k][col])

                    # Add the smallest number to the total count.
                    total_submatrices += min_continuous_ones

        # Return the total number of submatrices.
        return total_submatrices

```

Java Solution

```java
class Solution {

    // Method to count the number of submatrices with all ones
    public int numSubmat(int[][] mat) {
        int rows = mat.length; // the number of rows in given matrix
        int cols = mat[0].length; // the number of columns in given matrix
        int[][] width = new int[rows][cols]; // buffer to store the width of consecutive ones ending at (i, j)

        // Compute the width matrix
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If the cell contains a '1', calculate the width
                if (mat[i][j] == 1) {
                    width[i][j] = (j == 0) ? 1 : 1 + width[i][j - 1];
                }
                // '0' cells are initialized as zero, no need to explicitly set them
            }
        }

        int count = 0; // variable to accumulate the count of submatrices

        // For each position in the matrix, calculate the number of submatrices
        // that can be formed ending at (i, j)
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Start with a large number to minimize with the width of rows above
                int minWidth = Integer.MAX_VALUE;
                // Move up from row 'i' to '0' and calculate the minWidth for submatrices ending at (i, j)
                for (int k = i; k >= 0 && minWidth > 0; --k) {
                    minWidth = Math.min(minWidth, width[k][j]);
                    count += minWidth; // accumulate the count with the current minWidth
                }
            }
        }

        return count; // returning the total count of submatrices
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int numSubmat(vector<vector<int>>& mat) {
        int rows = mat.size(), cols = mat[0].size(); // dimensions of the matrix
        vector<vector<int>> width(rows, vector<int>(cols, 0)); // 2D vector to track the width of '1's ending at mat[i][j]

        // Pre-calculate the width of the continuous '1's in each row, left to right
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                if (mat[row][col] == 1) {
                    width[row][col] = col == 0 ? 1 : 1 + width[row][col - 1];
                }
                // if mat[row][col] is 0, width[row][col] stays 0, which was set initially
            }
        }

        int totalCount = 0; // count of all possible submatrices

        // Main logic to find the total number of submatrices with 1
        for (int topRow = 0; topRow < rows; ++topRow) {
            for (int col = 0; col < cols; ++col) {
                int minWidth = INT_MAX; // set minWidth to maximum possible value initially
                // Evaluate bottom-up for each submatrix
                for (int bottomRow = topRow; bottomRow >= 0 && minWidth > 0; --bottomRow) {
                    minWidth = min(minWidth, width[bottomRow][col]); // find minimal width of '1's column-wise
                    totalCount += minWidth; // Add the minimum width to the totalCount
                }
            }
        }

        return totalCount; // Return the final totalCount of all submatrices
    }
};

```

Typescript Solution

```typescript
function numSubmat(mat: number[][]): number {
    const rows = mat.length; // number of rows in the matrix
    const cols = mat[0].length; // number of columns in the matrix
    let width: number[][] = new Array(rows).fill(0).map(() => new Array(cols).fill(0)); // create 2D array for tracking continuous

    // Pre-calculate the width of continuous '1's for each cell in the row
    for (let row = 0; row < rows; ++row) {
        for (let col = 0; col < cols; ++col) {
            if (mat[row][col] === 1) {
                width[row][col] = col === 0 ? 1 : width[row][col - 1] + 1;
                // If mat[row][col] is 1, either start a new sequence of '1's or extend the current sequence
            }
            // If mat[row][col] is 0, width stays 0 (array initialized with 0's)
        }
    }

    let totalCount = 0; // Initialize count of all possible submatrices

    // Iterate through each cell in the matrix to calculate submatrices
    for (let topRow = 0; topRow < rows; ++topRow) {
        for (let col = 0; col < cols; ++col) {
            let minWidth = Number.MAX_SAFE_INTEGER; // Set minWidth to a high value (safe integer limit)

            // Iterate bottom-up to calculate all possible submatrices for each cell
            for (let bottomRow = topRow; bottomRow >= 0 && minWidth > 0; --bottomRow) {
                minWidth = Math.min(minWidth, width[bottomRow][col]); // Update minWidth column-wise for the current submatrix
                totalCount += minWidth; // Accumulate the count of 1's for this submatrix width
            }
        }
    }

    return totalCount; // Return the total count of submatrices with all 1's
}

// Example use:
// const matrix = [[1, 0, 1], [1, 1, 0], [1, 1, 0]];
// console.log(numSubmat(matrix)); // Output will depend on the provided matrix

```

Time and Space Complexity

The input matrix is of size `m x n`. Let's analyze the time and space complexity:

Time Complexity:

1. The first double `for` loop where the matrix `g` is being populated runs in `O(m * n)` time as it visits each element once.

2. The nested triple `for` loop structure calculates the number of submatrices. For each element `mat[i][j]`, we are going upwards and counting the number of rectangles ending at that cell. This part has a worst-case time complexity of `O(m * n * m)` because for each element in the matrix (`m * n`), we are potentially traveling upwards to the start of the column (`m`).

Thus, the total time complexity is `O(m * n + m * n * m)` which simplifies to `O(m^2 * n)`.

Space Complexity:

1. We have an auxiliary matrix `g` the same size as the input matrix which takes `O(m * n)` space.

2. Variables `col` and `ans` use a constant amount of extra space, `O(1)`.

The total space complexity of the algorithm is `O(m * n)` due to the auxiliary matrix `g`.