1504. Count Submatrices With All Ones

Medium Stack Array Dynamic Programming Matrix

Problem Description

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.

The problem presents a $m \times n$ binary matrix mat, where m represents the number of rows and n represents the number of

Monotonic Stack

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:

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]

the bottom-right corner, as explained in the intuition.

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

Create a running count grid: We initialize a grid g such that g[i][j] represents the number of consecutive '1's ending at

- 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 data structures used are:

• 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].

• An input matrix mat of size m x n, where mat[i][j] represents a cell in the original binary matrix.

- 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.
- and for each cell, it potentially iterates through m elements above it in the column. **Example Walkthrough**

In terms of complexity, the algorithm uses 0(m*n) space for the grid g and $0(m^2*n)$ time, since it iterates through the matrix

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

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:

[1, 0, 1],

a = 1

[1, 2, 3], [0, 1, 0]

additional two submatrices of size 1×1 (same row). Update $\frac{1}{2}$ and $\frac{1}{2}$ = 9.

 \circ Cell (2, 1) can form one submatrix of size 1×1. Update ans = 13 + 1 = 14.

 1×1 . Hence, we update ans = 9 + 4 = 13.

Skip cell (2, 0) as it is 0.

Skip cell (2, 2) as it is 0.

In terms of data structures and patterns:

The input matrix mat represents the original binary matrix.

time when checking the possibilities of forming submatrices.

aware of as it could become a performance bottleneck.

def numSubmat(self, mat: List[List[int]]) -> int:

Populate the continuous ones matrix.

for col in range(num cols):

if mat[row][col]:

for col in range(num cols):

Get the number of rows (m) and columns (n) of the matrix.

Initialize a count for total number of submatrices.

up to the current row 'row'.

for k in range(row, -1, -1):

min_continuous_ones = float('inf')

Travel up the rows from the current cell.

return count; // returning the total count of submatrices

int rows = mat.size(), cols = mat[0].size(); // dimensions of the matrix

// Pre-calculate the width of the continuous '1's in each row, left to right

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 minWidth = INT MAX; // set minWidth to maximum possible value initially

for (int bottomRow = topRow; bottomRow >= 0 && minWidth > 0; --bottomRow) {

totalCount += minWidth; // Add the minimum width to the totalCount

int numSubmat(vector<vector<int>>& mat) {

for (int row = 0; row < rows; ++row) {</pre>

if (mat[row][col] == **1**) {

for (int col = 0; col < cols; ++col) {</pre>

for (int topRow = 0; topRow < rows; ++topRow) {

for (int col = 0; col < cols; ++col) {</pre>

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

// Evaluate bottom-up for each submatrix

// Main logic to find the total number of submatrices with 1

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

If we encounter a '1', count the continuous ones. If in first column,

continuous_ones[row][col] = 1 if col == 0 else 1 + continuous_ones[row][col - 1]

This will keep track of the smallest number of continuous ones in the current column,

it can only be 1 or 0. Otherwise, add 1 to the left cell's count.

Calculate the number of submatrices for each cell as the bottom-right corner.

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

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

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

∘ 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

Solution Implementation

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

• The auxiliary grid g helps to calculate how many 1s we have consecutively to the left of a given cell, including the cell itself, which saves us

The space complexity remains 0(m*n) which is required for storing the grid g, and the time complexity is $0(m^2*n)$ due to the

nested iterations used for each element to consider the cells above it. For larger matrices, this complexity is something to be

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

for row in range(num rows):

total_submatrices = 0

for row in range(num rows):

Python

class Solution:

```
# 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
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 i = 0; i < cols; ++i) {
                // If the cell contains a '1', calculate the width
                if (mat[i][i] == 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][i]);
                    count += minWidth; // accumulate the count with the current minWidth
```

vector<vector<int>> width(rows, vector<int>(cols, 0)); // 2D vector to track the width of '1's ending at mat[i][j]

minWidth = min(minWidth, width[bottomRow][col]); // find minimal width of '1's column-wise

TypeScript

class Solution {

public:

```
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 '1
    // Pre-calculate the width of continuous '1's for each cell in the row
    for (let row = 0; row < rows; ++row) {</pre>
        for (let col = 0; col < cols; ++col) {</pre>
            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) {</pre>
        for (let col = 0; col < cols; ++col) {</pre>
            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
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):
```

total_submatrices += min_continuous_ones # Return the total number of submatrices. return total_submatrices

Time and Space Complexity

total_submatrices = 0

for row in range(num rows):

if mat[row][col]:

for col in range(num cols):

up to the current row 'row'.

for k in range(row, -1, -1):

min_continuous_ones = float('inf')

Travel up the rows from the current cell.

Initialize a count for total number of submatrices.

The input matrix is of size $m \times n$. Let's analyze the time and space complexity:

Add the smallest number to the total count.

If we encounter a '1', count the continuous ones. If in first column,

continuous ones[row][col] = 1 if col == 0 else 1 + continuous ones[row][col - 1]

This will keep track of the smallest number of continuous ones in the current column,

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])

it can only be 1 or 0. Otherwise, add 1 to the left cell's count.

Calculate the number of submatrices for each cell as the bottom-right corner.

The first double for loop where the matrix g is being populated runs in 0(m * n) time as it visits each element once. The nested triple for loop structure calculates the number of submatrices. For each element mat[i][j], we are going

Time Complexity:

Space Complexity:

- upwards and counting the number of rectangles ending at that cell. This part has a worst-case time complexity of 0(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 0(m * n + m * n * m) which simplifies to $0(m^2 * n)$.
- Variables col and ans use a constant amount of extra space, 0(1). The total space complexity of the algorithm is 0(m * n) due to the auxiliary matrix g.

We have an auxiliary matrix g the same size as the input matrix which takes 0(m * n) space.