1351. Count Negative Numbers in a Sorted Matrix

Easy Array Binary Search Matrix Leetcode Link

Problem Description

Given a matrix `grid` which has dimensions `m x n`, our task is to count all the negative numbers within this grid. The matrix is sorted in a non-increasing (also known as non-decreasing) order across both rows and columns—this means the numbers go from larger on the top-left to smaller on the bottom-right.

To visualize, we might have a matrix like this:

```
4   3   2  -1
3   2   1  -1
1   1  -1  -2
-1 -1  -2  -3
```

In the above example, you can see that rows and columns are sorted in non-increasing order and there are 8 negative numbers.

The goal is to count how many negative numbers are present.

Intuition

Since the grid is sorted in non-increasing order row-wise and column-wise, we can use a more efficient approach than checking each cell individually.

Here's the intuition step by step:

1. We start at the bottom-left corner of the matrix (`grid[m - 1][0]`). The reason for doing this is that if this number is negative, then all numbers to its right are guaranteed to be negative because the row is sorted in non-increasing order. On the other hand, if this number is not negative, we can move one column to the right.
2. We keep track of our current position with indices `i` for rows and `j` for columns. We set `ans` to 0 to keep track of our count of negative numbers.
3. We use a while loop that runs as long as `i` is within the rows and `j` is within the columns of the grid.
4. If the current element where we are (`grid[i][j]`) is negative, we add the count of remaining columns in that row to our answer (i.e., `ans += n - j`) because all of these will be negative. After doing that, we move one row up (`i -= 1`) since we're done with the current row.
5. If the element is not negative, we move one column to the right (`j += 1`) to check the next number in the same row.
6. The process is repeated until we've gone through all rows or columns.
7. We return the count `ans` which contains the number of negative numbers in the grid at the end of our loop.

The solution leverages the sorted property of the matrix to skip over non-negative elements and only count consecutive negatives, which makes it efficient.

Solution Approach

The solution uses a simple algorithm that takes advantage of the properties of the matrix, namely that it is sorted in non-increasing order in both rows and columns. No additional data structures are needed; we can operate directly on the given `grid`.

Here is a step-by-step walkthrough of the implementation:

1. Initiate two pointers, `i` for rows, starting from the last row (`m - 1`), and `j` for columns, starting from the first column (`0`).
2. Initiate a variable `ans` to store the count of negative numbers, initially set to 0.
3. Enter a while loop with the condition that `i` is greater than or equal to 0 (to make sure we are within the bounds of the rows) and `j` is less than `n` (to make sure we are within the bounds of the columns).
4. In the loop, check if the current element `grid[i][j]` is negative. If it is, increment `ans` by the number of columns remaining in the current row (`n - j`). This is because, since the rows are non-increasing, all elements to the right of `grid[i][j]` will also be negative. Then, decrease the row pointer `i` by 1, to move up to the previous row.
5. If `grid[i][j]` is not negative, just move to the next column by incrementing `j` by 1.
6. The loop continues until one of the exit conditions of the while loop is met (either all rows have been checked or the end of a row has been reached).
7. Finally, return the `ans` variable, which now contains the count of negative numbers in the grid.

The time complexity of this algorithm is $O(m + n)$, where m is the number of rows and n is the number of columns. This is because at each step of the algorithm, we're either moving one row up or one column to the right until we exit the matrix bounds. In the worst-case scenario, we'll make m+n moves.

The space complexity is $O(1)$ as we are not using any additional space proportional to the size of the input grid; instead, we're just using a few extra variables for counting and indexing.

No complex patterns are used here; the implementation is straightforward once the insight about the grid's sorted property is understood.

Example Walkthrough

Let's consider a smaller example to clearly illustrate the use of the solution approach in a concrete situation. Suppose we have the following `grid` matrix with dimensions `3 x 4`:

```
3   2   1  -1
2   1  -1  -2
1  -1  -2  -3
```

We want to count the number of negative numbers in this grid. Here's how we can apply the solution approach:

1. We initialize our pointers `i` with the index of the last row which is `2` for this grid (since we are using 0-based indexing), and `j` with the index of the first column which is `0`.
2. We set `ans` to `0` as our count of negative numbers.
3. We begin our while loop. Our initial element to check is `grid[2][0]`, which is `1`. Since it's not negative, we move one column to the right by incrementing `j` to `1`.
4. Now, `grid[2][1]` is `-1`, which is negative. We find that there are `4 - 1 = 3` columns remaining in the row, including the current column. So, we add `3` to `ans`, making it `3`, and move up a row by decrementing `i` to `1`.
5. Our new position is `grid[1][1]`, which is `1`. It's not negative, so we move one column to the right again by incrementing `j` to `2`.
6. We check `grid[1][2]`, it's `-1`. There are `4 - 2 = 2` columns remaining, so we add `2` to `ans`, which becomes `5`. We then move up a row by decrementing `i` to `0`.
7. The element `grid[0][2]` is `1`, not negative. We increment `j` to `3`.
8. At `grid[0][3]`, we have `-1`. With only `1` column at this last index, we add `1` to `ans`, resulting in a final count of `6`. Since there are no more rows above, the loop ends here as `i` is decremented and falls below `0`.

Thus, the `grid` contains `6` negative numbers.

With this example, it is clear how the algorithm smartly traverses the matrix, leveraging the sorted property to count negative numbers efficiently, without needing to check each element.

Python Solution

```python
class Solution:
    def countNegatives(self, grid: List[List[int]]) -> int:
        # Get the dimensions of the grid
        num_rows, num_cols = len(grid), len(grid[0])

        # Initialize pointers for the row and column
        row_index, col_index = num_rows - 1, 0

        # Initialize counter for negative numbers
        negative_count = 0

        # Iterate over the grid, starting from the bottom left corner
        while row_index >= 0 and col_index < num_cols:
            # If current element is negative
            if grid[row_index][col_index] < 0:
                # All elements in the current row to the right are also negative
                negative_count += num_cols - col_index

                # Move up to the previous row
                row_index -= 1
            else:
                # Move right to the next column
                col_index += 1

        # Return the total count of negative numbers
        return negative_count
```

Java Solution

```java
class Solution {
    public int countNegatives(int[][] grid) {
        int rowCount = grid.length; // The number of rows in the grid.
        int colCount = grid[0].length; // The number of columns in the grid.
        int negativeCount = 0; // Initialize a count for negative numbers.

        // Start from the bottom-left corner of the grid and move upwards in rows and rightwards in columns.
        for (int rowNum = rowCount - 1, colNum = 0; rowNum >= 0 && colNum < colCount;) {
            // If the current element is negative, all the elements to its right are also negative.
            if (grid[rowNum][colNum] < 0) {
                negativeCount += colCount - colNum; // Add the remaining elements in the row to the count.
                rowNum--; // Move up to the previous row to continue checking for negatives.
            } else {
                colNum++; // Move right to the next column to check for non-negative elements.
            }
        }

        return negativeCount; // Return the total count of negative numbers in the grid.
    }
}
```

C++ Solution

```cpp
#include <vector> // Necessary for using vectors

class Solution {
public:
    // Function to count the number of negative numbers in a 2D grid
    int countNegatives(std::vector<std::vector<int>>& grid) {
        int rowCount = grid.size(); // Number of rows in the grid
        int colCount = grid[0].size(); // Number of columns in the grid
        int negativeCount = 0; // Initialize the counter for negative numbers

        // Start from the bottom-left corner of the grid
        int currentRow = rowCount - 1;
        int currentColumn = 0;

        // Loop through the grid diagonally
        while (currentRow >= 0 && currentColumn < colCount) {
            // If the current element is negative, then all elements to its right are also negative
            if (grid[currentRow][currentColumn] < 0) {
                negativeCount += colCount - currentColumn; // Add all negative numbers in the current row
                --currentRow; // Move up to the previous row to continue with the negative number sweep
            } else {
                // If the current element is not negative, move right to the next column
                ++currentColumn;
            }
        }

        // Return the total count of negative numbers found in the grid
        return negativeCount;
    }
};
```

Typescript Solution

```typescript
// Counts the number of negative numbers in a row-wise and column-wise sorted grid.
// @param grid - A 2D array of numbers where each row and column is sorted in non-increasing order.
// @returns The total count of negative numbers in the grid.
function countNegatives(grid: number[][]): number {
    // Number of rows in the grid
    const rowCount = grid.length;
    // Number of columns in the grid - assumes at least 1 row exists
    const columnCount = grid[0].length;
    // Counter for negative numbers
    let negativeCount = 0;

    // Start from the bottom-left corner of the grid
    let row = rowCount - 1;
    let column = 0;

    // Loop until we reach the top of the grid or the end of a row
    while (row >= 0 && column < columnCount) {
        // If the current number is negative,
        // add all remaining negatives in the row to the counter
        // (as the row is sorted in non-increasing order)
        if (grid[row][column] < 0) {
            // All numbers to the right of the current position are negative
            negativeCount += columnCount - column;
            // Move up to the previous row since we've counted all negatives in the current row
            row--;
        } else {
            // If the current number is non-negative, move right to the next column
            column++;
        }
    }

    // Return total count of negative numbers
    return negativeCount;
}
```

Time and Space Complexity

The provided code implements an algorithm to count the number of negative numbers in an `m x n` grid, which is assumed to be non-increasing both row-wise and column-wise. The algorithm starts from the bottom-left corner of the grid and moves upwards or to the right depending on the sign of the current cell value.

Time Complexity:

The time complexity of the code can be analyzed as follows:

- The outer loop runs a maximum of `m` times if we're moving upwards from the bottom row to the topmost row in the worst case.
- The inner condition can lead to moving rightwards across up to `n` columns.

However, you will never revisit a row or a column once you've moved on from it (since you move up if you find a negative and move right if you find a non-negative number), meaning each cell is visited at most once. Therefore, the maximum number of steps is `m + n`.

Thus, the time complexity of the algorithm is `O(m + n)`.

Space Complexity:

The space complexity of the code is:

- Constant extra space is used for the variables `m`, `n`, `i`, `j`, and `ans`.
- No additional data structures are being used that grow with the size of the input.

Considering the above points, the overall space complexity of the algorithm is `O(1)`, as it uses a constant amount of extra space regardless of the input size.