1582. Special Positions in a Binary Matrix

Easy Array Matrix

Leetcode Link

Problem Description

In this LeetCode problem, we're given a binary matrix `mat`, which is made up of only `0`s and `1`s. Our task is to count how many 'special positions' are in the matrix. A special position is defined as one where the value at that position is `1` and all other values in the same row and column are `0`. Here, the matrix is indexed starting at `(0,0)` for the top-left element.

Intuition

The intuition behind the solution is to first count the number of `1`s in each row and each column. If a position `(i, j)` has a `1`, and the corresponding counts for row `i` and column `j` are both exactly 1, then the position `(i, j)` is a special position. Here's the breakdown:

1. We initialize two lists, `r` and `c`, to keep track of the count of `1`s in each row and each column, respectively.
2. By double looping through the matrix, we update these counts.
3. After populating `r` and `c`, we go through the matrix again, checking if a `1` is in a position `(i, j)` such that `r[i]` and `c[j]` are both exactly 1.
4. If the condition from step 3 is met, we increment our `ans` variable, which holds the count of special positions.
5. We return the value of `ans` as the final result.

This approach works because for a position with a `1` to be special, it must be the only `1` in its row and column. By counting the `1`s in each row and column first, we have all the information we need to efficiently determine if a position is special during our second pass through the matrix.

Solution Approach

The solution approach follows a straightforward algorithmic pattern that is quite common in matrix-related problems, which includes the following steps:

1. **Initialize Count Arrays**: The code initializes two arrays `r` and `c` with lengths equal to the number of rows `m` and columns `n` of the input matrix, respectively. These arrays are used to keep track of the sum of `1`s in each row and column, hence initialized to all `0`s.

2. **Populate Count Arrays**: The solution uses a nested loop where `i` iterates over the rows and `j` iterates over the columns. For each cell in the matrix, if the value `mat[i][j]` is `1`, the sum in the corresponding count arrays `r[i]` and `c[j]` are incremented by `1`. This allows us to accumulate the number of `1`s for each row and each column in their respective counters.

3. **Identify Special Positions**: With the populated count arrays, we loop through the matrix for the second time. During this iteration, we check if the value at `mat[i][j]` is `1` and if `r[i]` and `c[j]` are both equal to `1`. This condition verifies that the current position `(i, j)` is special as it is the only `1` in its row and column.

4. **Count Special Positions**: If the condition in the previous step is satisfied, we increment the variable `ans` which is used to count the number of special positions.

5. **Return the Result**: Once the entire matrix has been scanned during the second iteration, the `ans` variable contains the total count of special positions. This value is returned as the output.

The data structures used are quite simple and effective; we are using two one-dimensional arrays (`r` for rows and `c` for columns) to keep the sums. The algorithmic pattern employed is also straightforward, involving iterations and condition checking. This approach is efficient since each element in the matrix is processed a constant number of times, resulting in a time complexity of O(m*n), where `m` and `n` are the number of rows and columns in the matrix, respectively. The space complexity is O(m+n), which is required for the row and column sum arrays.

Example Walkthrough

Let's consider a 3×4 binary matrix `mat` for our example:

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

Following our algorithmic steps:

1. **Initialize Count Arrays**: We initialize two arrays `r` with length 3 (number of rows) and `c` with length 4 (number of columns), to all `0`s. So, `r = [0, 0, 0]` and `c = [0, 0, 0, 0]`.

2. **Populate Count Arrays**: We iterate through the matrix `mat`:

   - For `i=0` (first row), we find `mat[0][0]` is `1`, so we increment `r[0]` and `c[0]` by 1. Now, `r = [1, 0, 0]` and `c = [1, 0, 0, 0]`.
   - For `i=1` (second row), we find `mat[1][2]` is `1`, so we increment `r[1]` and `c[2]` by 1. Now, `r = [1, 1, 0]` and `c = [1, 0, 1, 0]`.
   - For `i=2` (third row), we find `mat[2][1]` is `1`, so we increment `r[2]` and `c[1]` by 1. Now, `r = [1, 1, 1]` and `c = [1, 1, 1, 0]`.

   After populating the counts arrays, `r` and `c` now accurately reflect the number of `1`s in each row and column.

3. **Identify Special Positions**: With the count arrays set up, we go through the matrix once more:

   - Check position `(0,0)`. Since `mat[0][0]` is `1` and both `r[0]` and `c[0]` are exactly 1, this is a special position.
   - Check position `(1,2)`. Since `mat[1][2]` is `1` and both `r[1]` and `c[2]` are exactly 1, this is also a special position.
   - Check position `(2,1)`. Since `mat[2][1]` is `1` and both `r[2]` and `c[1]` are exactly 1, this is a special position as well.

   Every `1` we encountered is indeed in a special position.

4. **Count Special Positions**: We increment our variable `ans` for each special position identified. As we found 3 special positions, `ans` would be 3.

5. **Return the Result**: Our function would return `ans`, the total count of special positions, which in this case is `3`.

In this straightforward example, our methodical walk-through demonstrates that the provided binary matrix `mat` contains three special positions, as identified using the solution approach. The row and column counts help efficiently pinpoint the special positions in just two scans of the matrix.

Python Solution

```python
class Solution:
    def numSpecial(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 row_sum and col_sum to keep track of the sum of each row and column
        row_sum = [0] * num_rows
        col_sum = [0] * num_cols

        # Calculate the sum of elements for each row and column
        for i, row in enumerate(mat):
            for j, value in enumerate(row):
                row_sum[i] += value
                col_sum[j] += value

        # Initialize variable 'special_count' to count special positions
        special_count = 0

        # Check for special positions where the value is 1
        # and its row and column sums are both exactly 1
        for i in range(num_rows):
            for j in range(num_cols):
                if mat[i][j] == 1 and row_sum[i] == 1 and col_sum[j] == 1:
                    # Increment the count of special positions
                    special_count += 1

        # Return the final count of special positions
        return special_count

```

Java Solution

```java
class Solution {
    public int numSpecial(int[][] mat) {
        int numRows = mat.length, numCols = mat[0].length;
        int[] rowCount = new int[numRows];
        int[] colCount = new int[numCols];

        // Calculate the sum of each row and each column
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                rowCount[i] += mat[i][j];
                colCount[j] += mat[i][j];
            }
        }

        int specialCount = 0;

        // Iterate through the matrix to find special elements
        // A special element is defined as the element that is the only '1' in its row and column
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                // Check if the current element is '1' and its corresponding
                // row and column sums are '1' which would mean it's a special element
                if (mat[i][j] == 1 && rowCount[i] == 1 && colCount[j] == 1) {
                    specialCount++;
                }
            }
        }

        // Return the total count of special elements found in the matrix
        return specialCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the number of special positions in a binary matrix.
    // A position (i, j) is called special if mat[i][j] is 1 and all other elements in row i and column j are 0.
    int numSpecial(vector<vector<int>>& mat) {
        int numRows = mat.size();           // Number of rows in the matrix
        int numCols = mat[0].size();       // Number of columns in the matrix
        vector<int> rowCount(numRows, 0);   // Row count to store the sum of each row
        vector<int> colCount(numCols, 0);   // Column count to store the sum of each column

        // Fill rowCount and colCount by summing the values in each row and column
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                rowCount[i] += mat[i][j];
                colCount[j] += mat[i][j];
            }
        }

        int specialCount = 0;  // Variable to store the number of special positions found

        // Search for special positions. A position (i, j) is special if
        // mat[i][j] is 1 and the sum of both row i and column j is 1.
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                if (mat[i][j] == 1 && rowCount[i] == 1 && colCount[j] == 1) {
                    specialCount++;  // Increment count if a special position is found
                }
            }
        }

        return specialCount;  // Return the total count of special positions
    }
};

```

Typescript Solution

```typescript
function countSpecialElements(matrix: number[][]): number {
    // Get the number of rows and columns from the matrix
    const rowCount = matrix.length;
    const colCount = matrix[0].length;

    // Create arrays to store the sum of elements in each row and column,
    // and initialize each element of the arrays to 0
    const rowSums = new Array(rowCount).fill(0);
    const colSums = new Array(colCount).fill(0);

    // First pass: Calculate the number of 1's in each row and column
    for (let rowIndex = 0; rowIndex < rowCount; rowIndex++) {
        for (let colIndex = 0; colIndex < colCount; colIndex++) {
            // If the element at the current position is 1, increment
            // the corresponding row and column sums
            if (matrix[rowIndex][colIndex] === 1) {
                rowSums[rowIndex]++;
                colSums[colIndex]++;
            }
        }
    }

    // Initialize the result variable which will hold the count of special elements
    let specialCount = 0;

    // Second pass: Check for special elements, which are the elements
    // that are the only 1 in their row and column
    for (let rowIndex = 0; rowIndex < rowCount; rowIndex++) {
        for (let colIndex = 0; colIndex < colCount; colIndex++) {
            // Check if the current element is 1 and if it's the only one
            // in its row and column, if so increment the specialCount
            if (matrix[rowIndex][colIndex] === 1 && rowSums[rowIndex] === 1 && colSums[colIndex] === 1) {
                specialCount++;
            }
        }
    }

    // Return the count of special elements
    return specialCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed by looking at the number of nested loops and the operations within them.

- The code first initializes the row and column count arrays `r` and `c`, which is `O(m)` and `O(n)` respectively, where `m` is the number of rows and `n` is the number of columns in the input `mat`.
- The first nested for loop iterates through all elements of the matrix to populate `r` and `c`, which will be `O(m * n)` since every element is visited once.
- The second nested for loop also iterates through the entire matrix to count the number of special elements based on the conditions that rely on the previous computations stored in `r` and `c`. This is also `O(m * n)`.

Hence, the overall time complexity is `O(m * n)` because this dominates the overall performance of the code.

Space Complexity

The space complexity of the code includes the space used for the input and any auxiliary space used:

- The input matrix itself does not count towards auxiliary space complexity as it is given.
- Two arrays `r` and `c` of length `m` and `n` are created to keep track of the sum of each row and column, which gives us `O(m + n)`.

Therefore, the auxiliary space complexity is `O(m + n)`.