2482. Difference Between Ones and Zeros in Row and Column

Medium Array Matrix Simulation Leetcode Link

Problem Description

In this LeetCode problem, we're tasked with creating a difference matrix `diff` from a given binary matrix `grid`. The `grid` is a matrix composed of 0s and 1s with `m` rows and `n` columns, and both matrices are 0-indexed, which means that counting starts from the top-left cell (0,0).

To construct the `diff` matrix, we follow these steps for each cell at position (i, j):

1. Calculate the total number of 1s (`onesRow_i`) in the ith row.
2. Calculate the total number of 1s (`onesCol_j`) in the jth column.
3. Calculate the total number of 0s (`zerosRow_i`) in the ith row.
4. Calculate the total number of 0s (`zerosCol_j`) in the jth column.
5. Set `diff[i][j]` to the sum of `onesRow_i` and `onesCol_j` subtracted by the sum of `zerosRow_i` and `zerosCol_j`.

Our goal is to return the `diff` matrix after performing these calculations for every cell in the `grid`.

Intuition

The intuition behind the solution is to use a straightforward approach by first calculating the sum of 1s in every row and column and storing them in two separate lists, `rows` and `cols`. This can be done by iterating over each element of the `grid`. If we encounter a 1, we increase the count for the respective row and column.

Once we have the sums of 1s for all rows and columns, we can calculate the difference matrix `diff`. For each cell in `diff[i][j]`, we want to add the number of 1s in the ith row and jth column, and then subtract the number of 0s in the ith row and jth column.

However, we can cleverly calculate the number of 0s by subtracting the number of 1s from the total number of elements in the row or column because the row sum of ones and zeros will always equal the total number of elements in that row or column.

For example, to get the number of 0s in the ith row, we subtract the number of 1s in that row from the total number of columns `n` (because each row has `n` elements), which gives us `zerosRow_i = n - onesRow_i`. Similarly, we get `zerosCol_j = m - onesCol_j`.

The final `diff` matrix value at `diff[i][j]` is then computed as `onesRow_i + onesCol_j - zerosRow_i - zerosCol_j`, which simplifies to `r + c - (n - r) - (m - c)` when plugging in the sums and the number of 0s. This computation is performed for all cells `(i, j)` in the grid to obtain the complete `diff` matrix.

Solution Approach

The implementation involves two main parts: first, computing the sum of 1s for each row and column; second, using these sums to calculate the `diff` matrix.

Let's break down the implementation step by step:

1. Initialize two lists `rows` and `cols` with length `m` and `n`, respectively, filled with zeros. These lists will keep track of the sum of 1s in each row and column. Initialize them with zeros as we haven't started counting yet.

2. Iterate over each cell in `grid` using nested loops. For each cell `(i, j)`, if the cell value is 1 (`v` in the code), increment `rows[i]` and `cols[j]` by 1. This loop runs through every element, ensuring that `rows` and `cols` accurately represent the number of 1s in their respective rows and columns.

3. After completing the sum of 1s, we initialize the `diff` matrix with zeros, creating an `m` by `n` matrix using list comprehension.

4. Now we iterate over each cell in the `diff` matrix. For every pair `(i, j)`, we calculate the value of `diff[i][j]` using the sums obtained previously. As derived before, the difference `r + c - (n - r) - (m - c)` simplifies to `2 * (r + c) - m - n`. This is because subtracting the zeros is the same as subtracting `m` or `n` and then adding back the number of ones in row `r` and column `c`.

5. The previous calculation is applied to all elements in the `diff` matrix by iterating through the ranges of `m` for rows and `n` for columns. This modifies the `diff` matrix to contain the correct difference values per the problem's definition.

In terms of algorithms and patterns, the solution uses a simple brute-force approach which runs in `O(m*n)` time because it requires iterating over all elements of the initial matrix to compute the sums, and then once more to compute the `diff` matrix.

The data structures are simple lists for tracking the sums of ones in rows and columns, and a 2D list for the `diff` matrix. No additional complex data structures or algorithms are needed, making the implementation both straightforward and efficient for the problem at hand.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with a binary `grid` of size `m x n` where `m = 2` and `n = 3`:

```
grid = [
  [1, 0, 1],
  [0, 1, 0]
]
```

We are expected to create the `diff` matrix following the steps described in the content. Here's the step-by-step breakdown:

1. Initialize two lists `rows` and `cols` with `m` and `n` zeros respectively, where `m` is the number of rows and `n` is the number of columns:

   `rows = [0, 0]` (for 2 rows)

   `cols = [0, 0, 0]` (for 3 columns)

2. Loop over each cell in `grid`. If we find a 1, increase the respective count in `rows` and `cols`:

   - For cell `(0, 0)`, `grid[0][0] = 1`, increment `rows[0]` and `cols[0]`: `rows = [1, 0]`, `cols = [1, 0, 0]`
   - For cell `(0, 1)`, `grid[0][1] = 0`, no increments.
   - For cell `(0, 2)`, `grid[0][2] = 1`, increment `rows[0]` and `cols[2]`: `rows = [2, 0]`, `cols = [1, 0, 1]`
   - For cell `(1, 0)`, `grid[1][0] = 0`, no increments.
   - For cell `(1, 1)`, `grid[1][1] = 1`, increment `rows[1]` and `cols[1]`: `rows = [2, 1]`, `cols = [1, 1, 1]`
   - For cell `(1, 2)`, `grid[1][2] = 0`, no increments.

3. Now that we have the sums of 1s in each row and column, we can initialize the `diff` matrix filled with zeros:

   ```
   diff = [
     [0, 0, 0],
     [0, 0, 0]
   ]
   ```

4. Next, iterate over each cell `(i, j)` in the `diff` matrix to calculate its value:

   - For cell `(0, 0)`, `diff[0][0] = 2 * (rows[0] + cols[0]) - m - n = 2 * (2 + 1) - 2 - 3 = 4`
   - For cell `(0, 1)`, `diff[0][1] = 2 * (rows[0] + cols[1]) - m - n = 2 * (2 + 1) - 2 - 3 = 4`
   - For cell `(0, 2)`, `diff[0][2] = 2 * (rows[0] + cols[2]) - m - n = 2 * (2 + 1) - 2 - 3 = 4`
   - For cell `(1, 0)`, `diff[1][0] = 2 * (rows[1] + cols[0]) - m - n = 2 * (1 + 1) - 2 - 3 = 0`
   - For cell `(1, 1)`, `diff[1][1] = 2 * (rows[1] + cols[1]) - m - n = 2 * (1 + 1) - 2 - 3 = 0`
   - For cell `(1, 2)`, `diff[1][2] = 2 * (rows[1] + cols[2]) - m - n = 2 * (1 + 1) - 2 - 3 = 0`

The `diff` matrix after setting the values is:

```
diff = [
  [4, 4, 4],
  [0, 0, 0]
]
```

This `diff` matrix represents the sum of 1s in each row and column, minus the sum of 0s for each respective cell in `grid`.

Python Solution

```python
class Solution:
    def onesMinusZeros(self, grid: List[List[int]]) -> List[List[int]]:
        # Determine the number of rows (m) and columns (n) in the grid
        num_rows, num_cols = len(grid), len(grid[0])

        # Initialize lists to store the sum of '1's in each row and column
        sum_rows = [0] * num_rows
        sum_cols = [0] * num_cols

        # Calculate the sum of '1's for each row and column
        for i in range(num_rows):
            for j in range(num_cols):
                sum_rows[i] += grid[i][j]  # Sum '1's for row i
                sum_cols[j] += grid[i][j]  # Sum '1's for column j

        # Initialize a list to store the resulting differences for each cell
        differences = [[0] * num_cols for _ in range(num_rows)]

        # Compute the differences for each cell in the grid
        for i in range(num_rows):
            for j in range(num_cols):
                # Calculate the difference by adding the sum of '1's in the current row and column
                # and subtracting the sum of '0's (computed by subtracting the sum of '1's from the total count)
                differences[i][j] = sum_rows[i] + sum_cols[j] - (num_cols - sum_rows[i]) - (num_rows - sum_cols[j])

        # Return the list containing the differences for each cell
        return differences

```

Java Solution

```java
class Solution {
    public int[][] onesMinusZeros(int[][] grid) {
        // Get the dimensions of the grid
        int rowCount = grid.length;
        int colCount = grid[0].length;

        // Create arrays to hold the count of 1s in each row and column
        int[] rowOnesCount = new int[rowCount];
        int[] colOnesCount = new int[colCount];

        // Calculate the total number of 1s in each row and column
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                int value = grid[i][j];
                rowOnesCount[i] += value;
                colOnesCount[j] += value;
            }
        }

        // Initialize a matrix to store the difference between ones and zeros for each cell
        int[][] differences = new int[rowCount][colCount];

        // Calculate the difference for each cell and populate the differences matrix
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                int onesTotal = rowOnesCount[i] + colOnesCount[j]; // Total number of 1s in the row i and column j
                int zerosTotal = (colCount - rowOnesCount[i]) + (rowCount - colOnesCount[j]); // Total number of 0s in the row i and
                differences[i][j] = onesTotal - zerosTotal;
            }
        }

        // Return the final matrix of differences
        return differences;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // This function takes a 2D grid of binary values and calculates the new grid
    // such that each cell in the new grid will contain the number of 1s minus the
    // number of 0s in its row and column in the original grid.
    vector<vector<int>> onesMinusZeros(vector<vector<int>>& grid) {
        // Dimensions of the original grid
        int rowCount = grid.size();
        int colCount = grid[0].size();

        // Vectors to store the sums of values in each row and column
        vector<int> rowSums(rowCount, 0);
        vector<int> colSums(colCount, 0);

        // Calculate the sums of 1s in each row and column
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                int value = grid[i][j];
                rowSums[i] += value;
                colSums[j] += value;
            }
        }

        // Create a new 2D grid to store the differences
        vector<vector<int>> differenceGrid(rowCount, vector<int>(colCount, 0));

        // Calculate the ones minus zeros difference for each cell
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                // The difference is the sum of ones in the row and column
                // minus the number of zeroes (which is rows/cols minus the sum of ones)
                differenceGrid[i][j] = rowSums[i] + colSums[j] - (colCount - rowSums[i]) - (rowCount - colSums[j]);
            }
        }

        // Return the new grid with the calculated differences
        return differenceGrid;
    }
};

```

Typescript Solution

```typescript
// Counts the number of 1's minus the number of 0's in each row and column for a 2D grid
function onesMinusZeros(grid: number[][]): number[][] {
    // Determine the number of rows and columns in the grid
    const rowCount = grid.length;
    const colCount = grid[0].length;

    // Initialize arrays to keep the counts of 1's for each row and column
    const rowOnesCount = new Array(rowCount).fill(0);
    const colOnesCount = new Array(colCount).fill(0);

    // First pass: Count the number of 1's in each row and column
    for (let i = 0; i < rowCount; i++) {
        for (let j = 0; j < colCount; j++) {
            if (grid[i][j] === 1) {
                rowOnesCount[i]++;
                colOnesCount[j]++;
            }
        }
    }

    // Prepare the answer grid with the same dimensions as the input grid
    const answerGrid = Array.from({ length: rowCount }, () => new Array(colCount).fill(0));

    // Second pass: Calculate ones minus zeros for each cell
    for (let i = 0; i < rowCount; i++) {
        for (let j = 0; j < colCount; j++) {
            // Sum the counts of 1's for the current row and column
            let sumOnes = rowOnesCount[i] + colOnesCount[j];
            // Count the zeros by subtracting the number of 1's from total row and column counts
            let sumZeros = (rowCount - rowOnesCount[i]) + (colCount - colOnesCount[j]);
            // Subtract the count of zeros from the number of ones and assign it to the answer grid
            answerGrid[i][j] = sumOnes - sumZeros;
        }
    }

    // Return the answer grid containing ones minus zeros for each cell
    return answerGrid;
}

```

Time and Space Complexity

Time Complexity

The given code consists of three distinct loops that iterate over the elements of the grid:

1. The first two loops (nested) are executed to calculate the sum of the values in each row and column. These loops go through all the elements of the matrix once. Therefore, for a matrix of size `m x n`, the time complexity of this part is `O(m * n)`.

2. The third set of nested loops is used to calculate the `diff` matrix. They also iterate over every element in the matrix, leading to a time complexity of `O(m * n)` for this part as well.

Adding both parts together doesn't change the overall time complexity since they are sequential, not nested within each other. Hence, the overall time complexity of the algorithm is `O(m * n)`.

Space Complexity

Analyzing the space complexity:

1. Two additional arrays `rows` and `cols` are created, which have lengths `m` and `n`, respectively. This gives a space complexity of `O(m + n)`.

2. A new matrix `diff` of size `m x n` is allocated to store the results. This contributes `O(m * n)` to the space complexity.

3. The space taken up by variables `m`, `n`, `i`, `j`, `r`, `c`, and `v` is constant, `O(1)`.

Therefore, the total space complexity of the algorithm is `O(m * n + m + n)`. Since `m * n` dominates for large matrices, the overall space complexity can be simplified to `O(m * n)`.