73. Set Matrix Zeroes

Medium Array Hash Table Matrix

Leetcode Link

Problem Description

Given a 2D matrix of integers, the challenge is to modify the matrix such that if any cell contains the value `0`, then all cells in the same row and column as that `0` are set to `0`. The crucial requirement is that this transformation must be done 'in place,' which means you cannot use any additional memory to store large temporary data structures like a copy of the original matrix. The goal is to achieve the required transformation efficiently with respect to space.

Intuition

To solve this problem we need to track the rows and columns that must be set to `0` without using extra space for another matrix. We can use two boolean variables (`i0` and `j0`) to keep track of whether the first row and the first column should be zeroed as they will serve as markers.

Here is an intuition for the solution approach:

- If we find a `0` in any cell of the matrix, the row and column of that cell must be zeroed out. However, if we immediately reset all cells in the same row and column to `0`, information about other `0`s in those rows and columns could be lost.
- To work around this, we can use the first cell of each row and column as a flag that indicates whether that entire row or column should be set to `0`.
- First, we check if the first row and first column should be zeroed by looking for any `0`s and store this information in `i0` for the first row, and `j0` for the first column.
- We iterate over the rest of the matrix (to avoid overriding `i0` and `j0` flags) and if we find a `0`, we set the first cell of that row and column to `0`. These markers will be used later to update the rows and columns.
- After marking, we iterate over the matrix again, ignoring the first row and column, and update cells to `0` if their row or column marker is `0`.
- Finally, we use `i0` and `j0` to set the first row and column to `0` if needed.

This approach allows us to keep track of which rows and columns need to be zeroed without using additional space, hence giving us an in-place solution.

Solution Approach

The solution involves a few logical steps to minimize space complexity while ensuring the entire row and column are set to `0` for each `0` in the matrix. Here's a step-by-step breakdown:

1. **Initialization**: We initialize two boolean variables `i0` and `j0`. We use `i0` to check if the first row needs to be set to `0` by iterating through the first row. Similarly, `j0` is used to check if the first column needs to be set to `0` by iterating down the first column.
2. **Marking Rows and Columns**: To preserve the information of where `0`s are in the matrix without using extra space, we use the first row and column of the matrix itself. As we iterate through the matrix starting from element at `(1,1)`, if a `0` is encountered at position `(i,j)`, we mark the first element of row `i` (at position `(i,0)`) and the first element of column `j` (at position `(0,j)`) to `0`.

   This way, we're using the matrix itself to keep track of rows and columns that will eventually be zeroed out. This clever trick avoids the need for additional data structures for marking purposes.
3. **Zeroing Rows and Columns**: Having marked the rows and columns that need to be updated, we now iterate again through the matrix starting from element at `(1,1)` to `m-1 x n-1`. For any cell `(i,j)`, we check if the first element of row `i` or the first element of column `j` is `0`. If either is `0`, this implies the current cell should also be `0`, and we update it accordingly.
4. **Finalizing the First Row and Column**: We check `i0` and `j0` from the first step. If `i0` is true, we zero out the first row. Similarly, if `j0` is true, we zero out the first column.

This in-place algorithm cleverly exploits the matrix's own first row and column to store the state needed to carry out the required transformation. This results in an `O(1)` space complexity as no additional storage proportional to the size of the matrix is needed, and the time complexity remains `O(m x n)` since we need to potentially look at all elements in the `m x n` matrix twice.

Example Walkthrough

Let's consider a small 3×3 example matrix to illustrate the solution approach:

```
1 Matrix before:
2 [ 1, 1, 1],
3 [ 1, 0, 1],
4 [ 1, 1, 1]
```

1. **Initialization**: `i0` and `j0` are initially set to `false`. We inspect the first row and first column to check for any `0`s:
   - First row has no `0`, thus `i0` remains `false`.
   - First column also has no `0`, so `j0` remains `false`.
2. **Marking Rows and Columns**: We start inspecting from (1,1):
   - No `0` at (1,1) — we move on.
   - We find a `0` at (1,3), so we mark the first element of row 1 (at position (1,0)) and the first element of column 3 (at position (0,3)) to `0`.

   Now the matrix looks like this with markers set:

```
1 [ [1, 2, 0],  // Column 3 marked
2   [0, 0, 0],  // Row 1 marked and the encountered 0
3   [1, 0, 0] ]
```

3. **Zeroing Rows and Columns**: With our markers set, we iterate through the matrix starting at (1,1) again and update according to markers:
   - Cell (1,1) stays `1` since neither the first element of row 1 nor column 1 is `0`.
   - Cell (1,2) becomes `0` because the first element of column 2 (position (0,2)) is `0`.
   - Continue through the matrix, checking the respective row or column start for a `0` marker.

   Updated matrix now looks like this:

```
1 [ [1, 0, 0],  // Column 3 zeroed out
2   [0, 0, 0],  // Row 1 zeroed out
3   [1, 0, 0] ] // Column 2 zeroed out because of the marker at (0,2)
```

4. **Finalizing the First Row and Column**: Since `i0` and `j0` are both `false`, we know the first row and column can remain unchanged.

The final matrix after performing the in-place transformation looks like:

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

This demonstrates how the matrix is correctly transformed according to the problem's rules without using any additional space, as all the zeroing information is stored within the original matrix itself.

Python Solution

```python
class Solution:
    def set_zeroes(self, matrix: List[List[int]]) -> None:
        # Get the dimensions of the matrix
        nrows, ncols = len(matrix), len(matrix[0])

        # Determine if the first row has any zeroes
        first_row_has_zero = any(value == 0 for value in matrix[0])
        # Determine if the first column has any zeroes
        first_col_has_zero = any(matrix[row][0] == 0 for row in range(nrows))

        # Use the first row and column as flags for zeroes
        # Start from 1 to avoid overwriting the first row and column flags
        for row in range(1, nrows):
            for col in range(1, ncols):
                # If an element is zero, mark its row and column in the first row and column
                if matrix[row][col] == 0:
                    matrix[row][0] = matrix[0][col] = 0

        # Set matrix elements to zero based on flags in the first row and column,
        # ignoring the first row and column themselves
        for row in range(1, nrows):
            for col in range(1, ncols):
                if matrix[row][0] == 0 or matrix[0][col] == 0:
                    matrix[row][col] = 0

        # If the first row had zeroes, set all elements in the first row to zero
        if first_row_has_zero:
            for col in range(ncols):
                matrix[0][col] = 0

        # If the first column had zeroes, set all elements in the first column to zero
        if first_col_has_zero:
            for row in range(nrows):
                matrix[row][0] = 0
```

Java Solution

```java
class Solution {
    public void setZeroes(int[][] matrix) {
        int rowCount = matrix.length; // number of rows in the matrix
        int colCount = matrix[0].length; // number of columns in the matrix
        boolean firstRowHasZero = false; // flag to check if the first row contains a zero
        boolean firstColHasZero = false; // flag to check if the first column contains a zero

        // Check if the first row has any zeros
        for (int col = 0; col < colCount; ++col) {
            if (matrix[0][col] == 0) {
                firstRowHasZero = true;
                break;
            }
        }

        // Check if the first column has any zeros
        for (int row = 0; row < rowCount; ++row) {
            if (matrix[row][0] == 0) {
                firstColHasZero = true;
                break;
            }
        }

        // Use the first row and column as markers.
        // Set matrix[i][0] and matrix[0][j] to 0 if matrix[i][j] is 0
        for (int row = 1; row < rowCount; ++row) {
            for (int col = 1; col < colCount; ++col) {
                if (matrix[row][col] == 0) {
                    matrix[row][0] = 0;
                    matrix[0][col] = 0;
                }
            }
        }

        // Iterate over the matrix again using the first row and column as reference,
        // and set the elements to 0 accordingly.
        for (int row = 1; row < rowCount; ++row) {
            for (int col = 1; col < colCount; ++col) {
                if (matrix[row][0] == 0 || matrix[0][col] == 0) {
                    matrix[row][col] = 0;
                }
            }
        }

        // Nullify the first row if needed
        if (firstRowHasZero) {
            for (int col = 0; col < colCount; ++col) {
                matrix[0][col] = 0;
            }
        }

        // Nullify the first column if needed
        if (firstColHasZero) {
            for (int row = 0; row < rowCount; ++row) {
                matrix[row][0] = 0;
            }
        }
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int rowCount = matrix.size(), colCount = matrix[0].size();
        bool firstRowHasZero = false, firstColHasZero = false;

        // Check if the first row has a zero
        for (int col = 0; col < colCount; ++col) {
            if (matrix[0][col] == 0) {
                firstRowHasZero = true;
                break;
            }
        }

        // Check if the first column has a zero
        for (int row = 0; row < rowCount; ++row) {
            if (matrix[row][0] == 0) {
                firstColHasZero = true;
                break;
            }
        }

        // Use first row and column as markers, set matrix[i][0] and matrix[0][j] to 0 if matrix[i][j] is 0
        for (int row = 1; row < rowCount; ++row) {
            for (int col = 1; col < colCount; ++col) {
                if (matrix[row][col] == 0) {
                    matrix[row][0] = 0;
                    matrix[0][col] = 0;
                }
            }
        }

        // Iterate over the matrix, set elements to 0 if their row or column marker is 0
        for (int row = 1; row < rowCount; ++row) {
            for (int col = 1; col < colCount; ++col) {
                if (matrix[row][0] == 0 || matrix[0][col] == 0) {
                    matrix[row][col] = 0;
                }
            }
        }

        // If the first row had zero, set all elements in the first row to 0
        if (firstRowHasZero) {
            for (int col = 0; col < colCount; ++col) {
                matrix[0][col] = 0;
            }
        }

        // If the first column had zero, set all elements in the first column to 0
        if (firstColHasZero) {
            for (int row = 0; row < rowCount; ++row) {
                matrix[row][0] = 0;
            }
        }
    }
};
```

Typescript Solution

```typescript
/**
 * Modifies a given matrix by setting entire rows and columns to zero if an element in them is zero.
 * Operates in-place.
 * @param {number[][]} matrix - A 2D array of numbers to be modified.
 */
function setZeroes(matrix: number[][]): void {
    // Dimensions of the matrix
    const rowCount = matrix.length;
    const colCount = matrix[0].length;

    // Flags indicating if the first row/column contains a zero
    const isFirstRowZero = matrix[0].includes(0);
    const isFirstColZero = matrix.some(row => row[0] === 0);

    // Use first row/column as markers for zero rows/columns.
    for (let row = 1; row < rowCount; ++row) {
        for (let col = 1; col < colCount; ++col) {
            if (matrix[row][col] === 0) {
                // Mark the respective first row and column cell
                matrix[row][0] = 0;
                matrix[0][col] = 0;
            }
        }
    }

    // Set matrix cells to zero based on the marks
    for (let row = 1; row < rowCount; ++row) {
        for (let col = 1; col < colCount; ++col) {
            if (matrix[row][0] === 0 || matrix[0][col] === 0) {
                matrix[row][col] = 0;
            }
        }
    }

    // If the first row needs to be set to zero
    if (isFirstRowZero) {
        for (let col = 0; col < colCount; col++) {
            matrix[0][col] = 0;
        }
    }

    // If the first column needs to be set to zero
    if (isFirstColZero) {
        for (let row = 0; row < rowCount; row++) {
            matrix[row][0] = 0;
        }
    }
}
```

Time and Space Complexity

The given code has a time complexity of `O(m * n)` where `m` is the number of rows and `n` is the number of columns in the given matrix. This is because the algorithm visits each cell twice, once to mark the rows and columns that should be zeroed and once to actually set the zeros.

For space complexity, the algorithm is `O(1)` as it uses constant extra space regardless of the size of the matrix. This is achieved by using the first row and first column of the matrix to store flags indicating whether a row or column should be set to zero.