2643. Row With Maximum Ones

Easy Array Matrix

Problem Description

Given a matrix where each cell is either a `0` or a `1`, we need to identify which row contains the most `1`s. The matrix is formatted as a list of lists in `m x n` dimensions, where `m` represents the number of rows and `n` the number of columns.

Our goal is to find the row index (0-indexed) with the highest number of `1`s. If there are multiple rows with the same maximum count of `1`s, we prioritize the row with the smallest index.

For the output, we must return a list with two elements: the index of the row containing the maximum count of ones and the count of ones in that respective row.

Intuition

To arrive at the solution, we can iterate through each row of the matrix and count the number of `1`s in that row. We do this by using Python's built-in `count` method on lists, which efficiently returns the number of occurrences of a particular element—in this case, the numeral `1`.

We initialize a list `ans` with two elements `[0, 0]`. The first element represents the index of the row and the second element represents the count of `1`s.

As we iterate over each row:

1. We count the `1`s,
2. We compare this count with the current maximum (stored in `ans[1]`),
3. If the count is higher, we update `ans` with the current row index and the new maximum count.

Solution Approach

The implementation of the solution uses a straightforward algorithm that takes advantage of simple iteration and Python's built-in list operations to solve the problem efficiently.

Here is a step-by-step guide to the algorithm:

1. Initialize an answer list `ans = [0, 0]`. The first element is the row index (initialized to 0) and the second element is the count of `1`s (also initialized to 0).

2. Iterate over each row in the matrix with a loop structure, keeping track of the current row index using Python's `enumerate` function.

```
for i, row in enumerate(mat):
```

3. Within the loop, count the number of `1`s in the current row. This is done by using the `count` method on the row list:

```
cnt = row.count(1)
```

4. Check if the current row's `1`s count (`cnt`) is greater than the maximum count of `1`s found so far (`ans[1]`). If it is, then update the answer list with the current row index and the count:

```
if ans[1] < cnt:
    ans = [i, cnt]
```

5. Once all rows have been checked, the list `ans` contains the required index and the count of `1`s. This list is then returned as the final output.

```
return ans
```

By using this approach, the solution minimizes complexity by only passing through the matrix a single time, which gives it a time complexity of O(m*n), where `m` is the number of rows and `n` is the number of columns in the matrix. The space complexity is O(1) since no extra space is used proportionally to the size of the input, aside from the space needed to store the answer `ans`.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Imagine we have the following matrix:

```
matrix = [
    [0, 1, 1, 0],
    [0, 1, 1, 1],
    [1, 1, 0, 0],
    [0, 0, 0, 0]
]
```

We need to identify which row has the most `1`s.

1. Initialize an answer list `ans = [0, 0]`. This list will hold the index of the row with the maximum `1`s and the count of `1`s in that respective row.

2. Start iterating over each row in the matrix using `enumerate`:

   - For the first row (`i=0`), the count of `1`s is `2`.
     - `ans` remains `[0, 0]` because `2` (current row's `1`s count) is not greater than `0` (maximum `1`s count so far).
   - For the second row (`i=1`), the count of `1`s is `3`.
     - Update `ans` to `[1, 3]` because `3` is greater than the maximum `1`s count so far, which was `0`.
   - For the third row (`i=2`), the count of `1`s is `2`.
     - `ans` remains `[1, 3]` because `2` is not greater than `3` (current maximum `1`s count).
   - The fourth row (`i=3`) has a count of `1`s as `0`.
     - `ans` remains `[1, 3]` since `0` is not greater than `3`.

3. Now that all rows have been checked, `ans` contains the index of the row with the maximum number of `1`s and the count itself, which is `[1, 3]`.

Thus, the output is `[1, 3]`, meaning the second row (`0`-indexed) has the most number of `1`s, which are `3` in total.

Solution Implementation

Python

```
class Solution:
    def row_and_maximum_ones(self, matrix: List[List[int]]) -> List[int]:
        # Initialize a variable to keep track of the row index
        # with the maximum number of ones, and the number of ones in that row.
        result = [0, 0]

        # Enumerate over the rows of the matrix.
        for i, row in enumerate(matrix):
            # Count the number of ones in the current row.
            ones_count = row.count(1)

            # If the count of ones in the current row is areater than
            # the maximum found so far, update the result.
            if result[1] < ones_count:
                result = [i, ones_count]  # Store the current row index and the count of ones.

        return result  # Return the result list containing the row index and the maximum count of ones.
```

Java

```
class Solution {

    /**
     * This method finds the row with the maximum number of ones in a binary matrix.
     * @param mat A binary matrix of integers where each integer is either 0 or 1.
     * @return An array containing the index of the row with the maximum number of ones
     *         and the count of ones in that row.
     */
    public int[] rowAndMaximumOnes(int[][] mat) {
        int[] result = new int[2]; // Holds the index of the row and maximum number of ones.

        for (int rowIndex = 0; rowIndex < mat.length; rowIndex++) { // Iterate over each row in the matrix.
            int onesCount = 0; // Counter for number of ones in the current row.

            for (int cellValue : mat[rowIndex]) { // Iterate over each element in the current row.
                if (cellValue == 1) { // Check if the current element is a one.
                    onesCount++; // Increment the ones counter.
                }
            }

            // Update result if the current row has more ones than previously found.
            if (result[1] < onesCount) {
                result[0] = rowIndex; // Set the index of the row with the most ones.
                result[1] = onesCount; // Set the new maximum number of ones.
            }
        }

        return result; // Return the array containing the row index and maximum number of ones.
    }
}
```

C++

```
class Solution {
public:
    // Function to find the row with the maximum number of ones and return that row index and count of ones in it
    vector<int> rowAndMaximumOnes(vector<vector<int>>& matrix) {
        vector<int> result(2); // Initialize result vector to store row index and max count of ones
        int maxOnes = 0; // Variable to store the current maximum number of ones

        // Iterate through each row in the matrix
        for (int rowIndex = 0; rowIndex < matrix.size(); ++rowIndex) {
            int onesCount = 0; // Variable to count the ones in the current row

            // Count the number of ones in the current row
            for (int element : matrix[rowIndex]) {
                onesCount += element == 1;
            }

            // If the current row has more ones than the previous maximum, update the result
            if (onesCount > maxOnes) {
                result[0] = rowIndex; // Update row index
                result[1] = onesCount; // Update max count of ones
                maxOnes = onesCount; // Update max ones for comparison in subsequent iterations
            }
        }

        // Return the result vector containing the index of the row and the maximum number of ones
        return result;
    }
};
```

TypeScript

```
// Function to find the row with the maximum number of ones in a binary matrix
// and return an array containing the row index and the count of ones.
function rowAndMaximumOnes(mat: number[][]): number[] {
    // Initialize the answer array with the first element as the row index
    // and the second element as the count of ones.
    const answer: number[] = [0, 0];

    // Iterate through each row of the matrix
    for (let rowIndex = 0; rowIndex < mat.length; rowIndex++) {
        // Count the number of ones in the current row by summing up the values
        const onesCount = mat[rowIndex].reduce((total, value) => total + value, 0);

        // If the count of ones in the current row is greater than the current maximum,
        // update the answer array with the new row index and ones count
        if (answer[1] < onesCount) {
            answer[0] = rowIndex;
            answer[1] = onesCount;
        }
    }

    // Return the answer array with the row index and count of ones for the row with the maximum number of ones
    return answer;
}

class Solution:
    def row_and_maximum_ones(self, matrix: List[List[int]]) -> List[int]:
        # Initialize a variable to keep track of the row index
        # with the maximum number of ones, and the number of ones in that row.
        result = [0, 0]

        # Enumerate over the rows of the matrix.
        for i, row in enumerate(matrix):
            # Count the number of ones in the current row.
            ones_count = row.count(1)

            # If the count of ones in the current row is areater than
            # the maximum found so far, update the result.
            if result[1] < ones_count:
                result = [i, ones_count]  # Store the current row index and the count of ones.

        return result  # Return the result list containing the row index and the maximum count of ones.
```

Time and Space Complexity

The time complexity of the given code is `O(m*n)`, where `m` is the number of rows and `n` is the number of columns in the matrix `mat`. This is because the code iterates through each row with `enumerate(mat)` and counts the number of `1`'s in that row with `row.count(1)`, which requires traversing the entire row.

The space complexity of the given code is `O(1)`. The additional space used by the algorithm is constant and does not scale with the input size since only a fixed-size list `ans` of length 2 is used to store the result and no other additional data structures are allocated.