1072. Flip Columns For Maximum Number of Equal Rows

Medium | Array | Hash Table | Matrix | Leetcode Link

Problem Description

In this problem, we are provided with a binary matrix, meaning that each cell in the matrix contains either 0 or 1. Our goal is to maximize the number of rows that are all made up of the same value, either all 0s or all 1s, by performing a certain operation. The operation allowed is flipping the values in a column; flipping transforms all 0s to 1s and all 1s to 0s in that particular column.

To visualize, imagine you have a matrix like this:

```
1 0 0
1 1 0
0 1 1
```

You can flip, for instance, the second and third columns to get:

```
1 1 1
1 0 1
0 0 0
```

Thus, you would have 2 rows with all values equal.

The problem asks us to find the maximum number of such rows possible after performing any number of flips.

Intuition

The intuition behind the solution is based on an observation about flipping columns. When we flip columns in a matrix, we don't actually care about the original values but rather the pattern of the rows, because any row pattern (i.e., sequence of zeros and ones) can be flipped to its inverse. Thus, two rows that are inverses (complements) of each other will become identical after flipping the same set of columns.

Therefore, the key insight is that we can pair rows that are identical or are inverses of each other because they can be made equal with the flip operation. If we represent a row by a tuple, we can ensure that if the first element is `0`, we keep it as is, and if the first element is `1`, we invert the entire row. We do this because the first element determines the flip pattern we should use: if it's `0`, no flip is needed for the first column, but if it's `1`, we should flip the first column.

The approach uses a `Counter` to keep track of how many identical or invertible row patterns there are. It iterates through each row in the matrix, treating it directly if it starts with a `0` or treating the inverted row (flipping all bits) if it starts with a `1`. The `Counter` will then have counts of how many times a certain pattern (or its inverse) appears in the matrix. The maximum of these counts is the number of rows that can be made equal by flipping the specified columns.

In summary, we rely on patterns and their complements to determine how many rows can be made equal through column flips by mapping each row to a unified pattern representation and counting the occurrences of these patterns.

Solution Approach

The implementation utilizes a `Counter` from the `collections` module in Python, which is a subclass of `dict` designed to count hashable objects. It implicitly counts how often each key is encountered. In the context of this problem, each key in the `Counter` will be a pattern (tuple) representing a row in the matrix after potentially inverting it.

Here is a breakdown of the steps involved in the implementation:

1. **Initialization**:
   - Create a new `Counter` object named `cnt`.
2. **Processing Rows**:
   - Iterate over each row in the input `matrix`.
   - For each row, check the first element:
     - If it is `0`, use the row as is.
     - If it is `1`, create a new tuple by flipping each element. The bitwise XOR operator `^` is used to flip bits - `x ^ 1` changes `0` to `1` and `1` to `0`.
3. **Counting Patterns**:
   - Convert the row (either used directly or inverted) to a tuple, which acts as an immutable and hashable object suitable for use as a key in the `Counter`.
   - Increment the count of this tuple in `cnt`. This step essentially counts how many times we have seen this particular pattern or its complement in the matrix.
4. **Finding the Maximum Count**:
   - After all rows are processed and counted, determine the maximum count using the `max` function on the `values()` of the `Counter`.
   - This maximum count represents the largest number of rows that can be made equal by flipping columns.

In essence, by using a `Counter` to tally row patterns, we are mapping the problem onto a frequency-count problem. Instead of directly manipulating the matrix, which would be costly, we represent each row by a pattern that considers flips and then look for the most common pattern. This drastically simplify the problem and allows us to find the solution efficiently.

Example Walkthrough

Let's walk through the solution approach using a small example of a binary matrix:

```
Matrix:
0 1 0
1 1 0
1 0 1
```

Following the solution approach step by step:

1. **Initialization**:
   - We create an empty `Counter` object named `cnt`.
2. **Processing Rows**:
   - We start by iterating over each row in the input matrix.
   - First row: `0 1 0`
     - The first element is `0`, so we use the row as is and move to step 3.
   - Second row: `1 1 0`
     - The first element is `1`, so we flip the entire row using the XOR operator: `0 0 1`.
   - Third row: `1 0 1`
     - The first element is `1`, so we flip the entire row using the XOR operator: `0 1 0`.
3. **Counting Patterns**:
   - We convert each row into a tuple for counting.
   - First row becomes the tuple `(0, 1, 0)`.
     - Increment `cnt[(0, 1, 0)]` by 1.
   - Second row (flipped) becomes the tuple `(0, 0, 1)`.
     - Increment `cnt[(0, 0, 1)]` by 1.
   - Third row (flipped) becomes the tuple `(0, 1, 0)` (same as the first row).
     - Increment `cnt[(0, 1, 0)]` by 1 again. Now `cnt[(0, 1, 0)]` is 2.
   - Our `Counter` now has two keys with counts: `[(0, 1, 0): 2, (0, 0, 1): 1]`.
4. **Finding the Maximum Count**:
   - We find the maximum count in `cnt`, which in this case is `2` corresponding to the pattern `(0, 1, 0)`.

Following these steps, we determine that the maximum number of rows with all values equal after flipping any number of columns is `2`. In this case, if we flip the first and the last columns of the original matrix, we can make the first and third rows identical (all zeros), achieving our goal.

Python Solution

```python
from collections import Counter

class Solution:
    def maxEqualRowsAfterFlips(self, matrix: List[List[int]]) -> int:
        # Initializing a counter to keep track of the frequency of each standardized row
        row_counter = Counter()

        # Iterating through each row in the matrix
        for row in matrix:
            # Standardizing the row:
            # If the first element is 0, keep the row unchanged; otherwise flip all bits
            standardized_row = tuple(row) if row[0] == 0 else tuple(1 - x for x in row)

            # Updating the counter for the standardized row
            row_counter[standardized_row] += 1

        # Returning the maximum frequency found in the counter as the result
        return max(row_counter.values())

```

Java Solution

```java
class Solution {
    public int maxEqualRowsAfterFlips(int[][] matrix) {
        // HashMap to keep track of the frequency of each unique pattern
        Map<String, Integer> patternFrequency = new HashMap<>();
        // Variable to keep track of the max number of equal rows after flips
        int maxEqualRows = 0;
        // The number of columns in the matrix
        int columnCount = matrix[0].length;

        // Iterate over each row in the matrix
        for (int[] row : matrix) {
            // Create a character array to represent the pattern
            char[] patternChars = new char[columnCount];
            // Build the pattern based on the first element in the row
            for (int i = 0; i < columnCount; ++i) {
                // XOR the first element with each element in the row
                // If row[0] == row[i], the result will be 0; otherwise, it will be 1
                patternChars[i] = (char) ('0' + (row[0] ^ row[i]));
            }
            // Convert the character array to a string representing the pattern
            String pattern = String.valueOf(patternChars);
            // Update the pattern frequency map with the new pattern,
            // incrementing the count of the pattern by 1
            int patternCount = patternFrequency.merge(pattern, 1, Integer::sum);
            // Update maxEqualRows if the current pattern count is greater
            maxEqualRows = Math.max(maxEqualRows, patternCount);
        }

        // Return the maximum number of equal rows that can be obtained after flips
        return maxEqualRows;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>
using namespace std;

class Solution {
public:
    int maxEqualRowsAfterFlips(vector<vector<int>>& matrix) {
        unordered_map<string, int> patternCount; // This will map row patterns to their counts
        int maxEqualRows = 0; // This will keep track of the maximum number of equal rows

        // Iterate through rows of the matrix
        for (auto& row : matrix) {
            string pattern; // Initialize an empty string to store the row pattern

            // Build the pattern for the given row considering flips
            for (int cell : row) {
                // If the first cell is 0, keep the number as is; otherwise flip the number
                char representation = '0' + (row[0] == 0 ? cell : cell ^ 1);
                pattern.push_back(representation);
            }

            // Increase the count for the current pattern
            int currentCount = ++patternCount[pattern];
            // Update maxEqualRows if the current pattern count exceeds it
            maxEqualRows = max(maxEqualRows, currentCount);
        }

        return maxEqualRows; // Return the maximum number of equal rows after flips
    }
};

```

Typescript Solution

```typescript
/**
 * Determines the maximum number of rows that can be made equal after a series of flips.
 * Flips can be performed on an entire row which flips all 0s to 1s, and vice versa.
 * @param {number[][]} matrix - The 2D array on which flips are performed.
 * @returns {number} - The maximum number of rows that can be made equal by flipping.
 */
function maxEqualRowsAfterFlips(matrix: number[][]): number {
    const countMap = new Map<string, number>(); // Map to store the frequency of each row pattern.
    let maxEqualRows = 0; // Variable for tracking the maximum number of equal rows.

    for (const row of matrix) {
        // If the first element of the row is a 1, we flip the entire row to make sorting consistent.
        if (row[0] === 1) {
            // Perform the flip by XOR'ing each element in the row with 1.
            for (let i = 0; i < row.length; i++) {
                row[i] ^= 1;
            }
        }
        // Convert the row to a string to use as a key in the map.
        const rowString = row.join('');
        // Update the count in the map for the given row pattern.
        // If it doesn't exist yet, initialize to 0 then add 1, else increment the current count.
        countMap.set(rowString, (countMap.get(rowString) || 0) + 1);
        // Update maxEqualRows with the maximum of the current value and the new count for this pattern.
        maxEqualRows = Math.max(maxEqualRows, countMap.get(rowString)!);
    }

    // Return the highest frequency of equal row patterns after flips.
    return maxEqualRows;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the number of iterations through each row of the `matrix` and operations performed for each row. Since we iterate once over each row, and for each row, we either take the tuple as-is if the starting element is 0 or iterate once more through the row to flip each bit and create a tuple, the time per row is `O(N)` where `N` is the number of columns. With `M` being the number of rows, iterating over all rows results in a time complexity of `O(M * N)`.

Furthermore, the `max` function that is called on the `Counter` object takes `O(U)` time, where `U` is the number of unique rows after normalizing which in the worst case could be `M`. Thus, the overall time complexity combines the row iterations and the `max` function, and remains `O(M * N)`.

Space Complexity

The space complexity depends on the space required to store the counter dictionary and the tuples created for each row. Each tuple can have at most `N` elements, and in the worst case, we could have `M` unique tuples if no two rows are the same or opposites. Therefore, the space complexity is `O(M * N)`, since `M` tuples of `N` elements each may need to be stored in the counter dictionary.

In all, the space complexity of the algorithm is `O(M * N)`.