# 861. Score After Flipping Matrix

Medium Greedy Bit Manipulation Array Matrix

## Problem Description

You are presented with a binary matrix `grid`, which is a 2D array containing `0`s and `1`s. Your task is to maximize the "score" of this matrix. The score is calculated by summing up all the rows considered as binary numbers. In order to raise the score, you can perform as many "moves" as you want. A move consists of picking any row or column and flipping all the bits in it; that is, all `0`s are changed to `1`s and vice versa.

For example, if you have a row `[1, 0, 1]` and you perform a move, this row will become `[0, 1, 0]`.

The primary goal is to find the highest score possible after performing an optimal sequence of moves.

## Intuition

The key observation to solve this problem is to realize that the most significant bit in a binary number has the highest value, so it should always be set to `1` to maximize the number. Therefore, the first step is to ensure all rows start with a `1`. If a row starts with a `0`, we flip the entire row.

The next step is to consider each column, starting from the second one since the first column should already have all `1`s after the initial step. At each column, we can maximize the score by making sure that the number of `1`s is greater than or equal to the number of `0`s. If there are more `0`s than `1`s, we flip the entire column. This is because flipping a column doesn't change the leftmost `1`s that we have set in the first step but increases the number of `1`s in the current column, therefore increasing the overall score.

By applying these two steps, first row-wise, then column-wise, we ensure that each bit contributes the maximum possible value to the score of the matrix.

Here's the reasoning for the solution:

1. Flip the rows if the first bit is `0` (outer loop over rows).
2. For each column, count the number of `1`s. If the number is less than half of the total rows, flip the column (inner loop over columns).
3. Calculate the score by adding the value of each column's bits to the total score.

In the provided code, `grid[i][j] ^= 1` is used to flip the bits (using the XOR operator). `max(cnt, m - cnt) * (1 << (n - j - 1))` is the calculation of the contribution of each column to the score, choosing to flip the column or not depending on which option gives more `1`s, and then multiplying by the value of the bit position, which decreases as we move right.

## Solution Approach

The solution takes a greedy approach to maximizing the score of the binary matrix by making sure that the most significant bit of every row is set to `1` and then ensuring that each column contains as many `1`s as possible.

Here's the breakdown of how the code implements the solution:

1. The first loop iterates through each row in the binary matrix `grid`. The size of the matrix is determined by `m` rows and `n` columns.
2. Inside this loop, we check if the first element of the row (`grid[i][0]`) is `0`. If so, we flip the entire row using a nested loop which toggles each element `grid[i][j] ^= 1`. This is done using the XOR operator `^`, which will flip a `0` to `1` and vice versa.
3. The outer loop with the variable `j` goes through each column. The inner loop counts the number of `1`s in the current column. Instead of flipping each bit when necessary, we're just counting the number of `1`s because we can compute the column contribution to the score mathematically, without modifying the matrix.
4. For each column, we calculate the maximum possible value it can contribute to the score based on the current state of bits in that column. This is computed by `max(cnt, m - cnt)`. We take the larger value between the number of `1`s (`cnt`) and the number of `0`s (`m - cnt`), considering that if the `0`s were in the majority, a column flip would change them to `1`s.
5. The column's contribution to the overall score is then computed by multiplying this maximum number with the binary value of the position. In a binary number, the leftmost bit is the most significant; hence, the value is computed by `1 << (n - j - 1)`, which is equivalent to raising `2` to the power of the column's index from the right (0-based). For example, if the column is the second from the right in a `4`-column grid, its value is `2^(4-2-1) = 2^(1) = 2`.
6. This contribution is added to `ans` for each column to build up the total score.
7. Once all columns have been processed, the variable `ans` holds the highest possible score, which is returned.

The algorithm efficiently calculates the maximum score possible without needing to actually perform the flips visually or record the state of the matrix after each flip, thanks to mathematical binary operations and the properties of binary numbers.

### Example Walkthrough

Let's take a simple example and walk through the solution approach to understand it better.

Consider the binary matrix, `grid`, represented by the following 2D array:

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

The matrix has 3 rows (`m = 3`) and 2 columns (`n = 2`). We want to maximize the "score" of this matrix through flips.

**Step by Step Solution:**

1. Make sure that all rows start with `1`:
   - Row 1 starts with `0`, so we flip it: `0 1` becomes `1 0`.
   - Row 3 also starts with `0`, we flip it: `0 1` becomes `1 0`.

   After this step, our grid looks like this:

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

2. Now, we need to maximize `1`s in each column. We note that the first column already has all `1`s because of the initial row flips. So we consider the second column.
3. Count the number of `1`s in the second column:
   - There is only 1 `1` in the second column.

   We want the majority of the bits in each column to be `1`. Since there is only 1 '1' and 2 '0's in the second column, we'll consider this column flipped for maximization purposes.
4. Calculate the contribution of each column to the total score:
   - The first column, having all `1`s, contributes `2^(n-1-0) = 2^(2-1-0) = 2^1 = 2` for each row, totaling `2 * 3 = 6`.
   - The second column, we consider flipped (for scoring purposes), so it effectively has 2 `1`s, and each contributes `2^(n-2-0) = 2^(2-2-0) = 2^0 = 1` to the score. The total from this column will be `1 * 2 = 2`.
5. The total score is then the sum of the contributions from step 4:
   - Total score = Score from column 1 + Score from column 2
   - Total score = 6 + 2
   - Total score = 8

The highest possible score for the given matrix after the optimal sequence of moves is 8.

## Solution Implementation

### Python

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

        # Flip the rows where the first element is 0
        for i in range(num_rows):
            if grid[i][0] == 0:
                for j in range(num_cols):
                    # XOR operation to flip the bits (0 becomes 1, and 1 becomes 0)
                    grid[i][j] ^= 1

        # Initialize the variable to store the result
        result = 0

        # Iterate over columns
        for j in range(num_cols):
            # Count the number of 1s in the current column
            num_ones = sum(grid[i][j] for i in range(num_rows))
            # Maximize the column value by flipping if necessary (the majority should be 1s)
            # Use the maximum of the number of 1s or the number of 0s in the column
            max_value = max(num_ones, num_rows - num_ones)

            # Add the value of the column to the result
            # The value is determined by the number of 1s times the value of the bit
            # position (1 << (num_cols - j - 1)) is the bit value of the current column
            result += max_value * (1 << (num_cols - j - 1))

        # Return the maximized score after considering all rows and columns
        return result
```

### Java

```java
class Solution {
    public int matrixScore(int[][] grid) {
        // Get the number of rows (m) and columns (n) in the grid.
        int numRows = grid.length, numCols = grid[0].length;

        // Step 1: Flip the rows where the first element is 0.
        for (int row = 0; row < numRows; ++row) {
            // Check if the first column of the current row is 0.
            if (grid[row][0] == 0) {
                // Flip the entire row.
                for (int col = 0; col < numCols; ++col) {
                    grid[row][col] ^= 1; // Bitwise XOR operation flips the bit.
                }
            }
        }

        // Step 2: Calculate the final score after maximizing the columns.
        int ans = 0; // This variable stores the final score.

        // Loop through each column.
        for (int col = 0; col < numCols; ++col) {
            int colCount = 0; // Count the number of 1s in the current column.
            for (int row = 0; row < numRows; ++row) {
                // Add to the column count if the bit is 1.
                colCount += grid[row][col];
            }
            // Maximize the column by choosing the maximum between colCount and (numRows - colCount).
            // Use bit shifting (1 << (numCols - col - 1)) to represent the value of the column.
            ans += Math.max(colCount, numRows - colCount) * (1 << (numCols - col - 1));
        }

        // Return the final score of the binary matrix after row and column manipulations.
        return ans;
    }
}
```

### C++

```cpp
class Solution {
public:
    int matrixScore(vector<vector<int>>& A) {
        // Number of rows in the matrix
        int num_rows = A.size();
        // Number of columns in the matrix
        int num_cols = A[0].size();

        // Flip the rows where the first element is 0 to maximize the leading digit
        for (int row = 0; row < num_rows; ++row) {
            if (A[row][0] == 0) {
                for (int col = 0; col < num_cols; ++col) {
                    A[row][col] ^= 1; // Xor with 1 will flip 0s to 1s and vice versa
                }
            }
        }

        int total_score = 0;
        // Iterate through columns to maximize the score
        for (int col = 0; col < num_cols; ++col) {
            int col_count = 0; // Count of 1s in the current column

            // Count the number of 1s in the current column
            for (int row = 0; row < num_rows; ++row) {
                col_count += A[row][col];
            }

            // Determine the value to add to the score for this column
            // We take the larger number between 'col_count' and 'num_rows - col_count'
            // to maximize the column value (since either all 1s or all 0s since we can flip)
            int max_col_value = std::max(col_count, num_rows - col_count);
            // The value of a set bit in the answer is equal to 2^(num_cols - col - 1)
            total_score += max_col_value * (1 << (num_cols - col - 1));
        }

        return total_score; // Return the maximized score after performing the operations
    }
};
```

### TypeScript

```typescript
function matrixScore(grid: number[][]): number {
    const rows = grid.length; // number of rows in the grid
    const cols = grid[0].length; // number of columns in the grid

    // Step 1: Toggle rows to ensure the first column has all 1s
    for (let row = 0; row < rows; ++row) {
        if (grid[row][0] === 0) {
            // If the first cell is 0, toggle the entire row
            for (let col = 0; col < cols; ++col) {
                grid[row][col] ^= 1; // XOR with 1 will toggle the bit
            }
        }
    }

    let score = 0; // Initialize the total score

    // Step 2: Maximize each column by toggling if necessary
    for (let col = 0; col < cols; ++col) {
        let countOnes = 0; // Count of ones in the current column
        for (let row = 0; row < rows; ++row) {
            // Count how many 1s are there in the current column
            countOnes += grid[row][col];
        }
        // Choose the max between countOnes and the number of 0s (which is rows - countOnes)
        score += Math.max(countOnes, rows - countOnes) * (1 << (cols - col - 1));
    }

    return score; // Return the total score
}
```

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

        # Flip the rows where the first element is 0
        for i in range(num_rows):
            if grid[i][0] == 0:
                for j in range(num_cols):
                    # XOR operation to flip the bits (0 becomes 1, and 1 becomes 0)
                    grid[i][j] ^= 1

        # Initialize the variable to store the result
        result = 0

        # Iterate over columns
        for j in range(num_cols):
            # Count the number of 1s in the current column
            num_ones = sum(grid[i][j] for i in range(num_rows))
            # Maximize the column value by flipping if necessary (the majority should be 1s)
            # Use the maximum of the number of 1s or the number of 0s in the column
            max_value = max(num_ones, num_rows - num_ones)

            # Add the value of the column to the result
            # The value is determined by the number of 1s times the value of the bit
            # position (1 << (num_cols - j - 1)) is the bit value of the current column
            result += max_value * (1 << (num_cols - j - 1))

        # Return the maximized score after considering all rows and columns
        return result
```

## Time and Space Complexity

### Time Complexity

The time complexity of the given solution involves iterating over each cell of the grid to potentially toggle its value, followed by counting the number of 1's in each column to maximize the row score.

- The first loop, which toggles the cells if the leftmost bit is 0, traverses all cells in the grid and performs a constant-time XOR operation. This double loop runs in `O(m * n)` time, where `m` is the number of rows and `n` is the number of columns.
- The second loop calculates the count of 1's in each column and determines the maximum score by considering the current and the inverse value count. This also runs in `O(m * n)` time since it involves iterating over each column for all rows.

The two operations above are consecutive, resulting in a total time complexity of `O(m * n) + O(m * n)`, which simplifies to `O(m * n)` because constant factors are dropped.

### Space Complexity

The space complexity of the algorithm is determined by any additional space that we use relative to the input size:

- No extra space is used for processing besides a few variables for iteration (`i`, `j`) and storing intermediate results (such as `cnt` and `ans`). These use a constant amount of space irrespective of the input size.
- Since the grid is modified in place, and no additional data structures are used to store intermediate results, the space complexity remains constant.

Given the fixed number of variables with space independent of the input size, the space complexity is `O(1)`, which signifies constant space complexity.