Approach

Intuition

By maximizing the number of set bits in the most significant bit positions, we can achieve maxximum possoble score.

Approach: Greedy + Toggling

Here's how the approach works:

1. **Initial Score Initialization**: We initially assume that all the bits in the most significant position are set bits, meaning first column contains all 1s, which can be done by multiplying the number of rowws n with the value represented by the most significant bit (1 << (m - 1)).
2. **Column Toggling**: We then iterate over the remaining columns (from the second column to the last column) to determine whether toggling (flipping) the entire column would increase the score or not.
3. **Value of the Current Column**: We then calculate a value val for each column j represented by that column's bit position (1 << (m - 1 - j)).
4. **Counting Matching Rows**: We then count the number of rows represented by set , where the value in the current column j matches the value in the first column (0 or 1) by iterating over all rows and comparing grid[i][j] with grid[i][0].
5. **To Toggle or Not**: If the number of rows set is greater than or equal to n/2 (half the number of rows), its beneficial for us to not toggle (flip) the column. Otherwise, its better to toggle (flip) the entire column to increase the score.
6. **Updating the Score**: We then update the res score by adding the maximum value between set * val (if the column is kept unchanged) and (n - set) * val (if the column is toggled), ensuring that the score is maximized by either keeping the current column unchanged or toggling it.
7. **Resulting Score**: After iterating through all columns, we return the maximum score res.

This approach works because toggling a row or a column affects all the bits in that row or column simultaneously. By ensuring that the most significant bits are set to 1 in as many rows as possible, and then adjusting the remaining columns based on the majority values, we can maximize the score of the matrix.

Dry - Run

Given Input:
grid =

[0, 0, 1, 1]

[1, 0, 1, 0]

[1, 1, 0, 0]

- Initialize variables:
  - n = 3(number of rows)
  - m = 4 (number of columns)
  - res = (1 << (m - 1)) * n = (1 << 3) * 3 = 8 * 3 = 24
  - [1, 1, 0, 0]
  - [1, 0, 1, 0]

[1, 1, 0, 0]

since our 1st row's 1st element is 0 and we have to make all the elements in our 1st column as 1s to maximize our score, we'll toggle our first row

- Iterate through columns (j = 1 to m - 1):
  - j = 1:
    - val = 1 << (m - 1 - j) = 1 << 2 = 4
    - set = 2(since grid[0][1] == grid[0][0] and grid[2][1] == grid[2][0])
    - res += Math.max(set, n - set) * val
= Math.max(2, 1) * 4
= 2 * 4 =8
    - res = 24 + 8 = 32
    - no need to toogle this column
  - j = 2:
    - val = 1 << (m - 1 - j) = 1 << 1 = 2
    - set = 2 (since grid[1][2] == grid[1][0] and grid[2][2] == grid[2][0])
    - res += Math.max(set, n - set) * val
= Math.max(2, 1) * 2
= 2 * 2 = 4
    - res = 32 + 4 = 36
    - [1 ,1 ,1 ,0]
    - [1 ,0 ,0 ,0]

[1 ,1 ,1 ,0]

  - j = 3:
    - val = 1 << (m - 1 - j) = 1 << 0 = 1

- set = 0 (since grid[0][3] != grid[0][0], grid[1][3] != grid[1][0], and grid[2][3] != grid[2][0])
- res += Math.max(set, n - set) * val
  = Math.max(0, 3) * 1
  = 3 * 1 = 3
- res = 36 + 3 = 39
- [1 ,1 ,1 ,1]
- [1 ,0 ,1 ,1]

[1 ,1 ,1 ,1]

- Return the final result:
  - return res = 39

Complexity

- Time complexity: O(n * m)
- Space complexity: O(1)