832. Flipping an Image

Easy Array Two Pointers Matrix Simulation

Problem Description

The problem provides us with a two-dimensional square matrix `image`, where each element is a binary value (that is, either `0` or `1`). Our goal is to transform this matrix through two steps:

- First, we need to flip each row of the matrix horizontally. Flipping horizontally means that we reverse the order of the elements in each individual row. If we visualize each row as a sequence of pixels in an image, this operation would resemble looking at the row in a mirror.
- Second, we need to invert the entire matrix. Inverting means we swap each `0` with `1` and vice versa.

The task is to perform these operations on the given matrix and return the resulting matrix. A key point is that we need to flip first and then invert.

Intuition

To approach this problem, let's focus on a single row to understand the operations required. Flipping a row is straightforward: we simply need to reverse the elements in the row. This can be done by swapping the elements starting from both ends and moving towards the middle. For example, to flip `[1,1,0]` we swap the first and the last elements to get `[0,1,1]`.

Inverting a row then requires us to iterate over each element and switch `1`s to `0`s and `0`s to `1`s.

However, we can perform the flipping and inverting in a single pass to make our solution more efficient. Notice that if an element is different from its corresponding element on the opposite side, flipping will just move them around, and inverting won't change that fact—they will remain different. The only case where we need to act is when the elements are the same; then, we swap and immediately invert (since post-flipping, they would be mirrored and the same). This can be achieved using an XOR operation with `1` (since `0 ^ 1 = 1` and `1 ^ 1 = 0`).

Also, if the number of elements in a row is odd, there will be a single element in the center after flipping, which will not change its position. This element alone should be inverted.

The code given carries out this optimized algorithm. It iterates over each row, flipping and inverting when needed, and deals with the potential middle element of odd-length rows. Finally, it returns the modified matrix, now flipped and inverted as required by the problem description.

Solution Approach

The solution to this problem involves a straightforward yet smart iteration over the matrix, particularly focusing on each row one by one. Several concepts are at play here:

- **In-place Modification**: To save on space complexity, we perform the flipping and inverting of elements in place, modifying the original matrix without using any additional data structures for storage.
- **Two-Pointer Technique**: This is a common pattern used in array and string manipulations. Here, it is utilized to reverse the elements of each row. We start with two pointers, `i` and `j`, representing the start and end of a row, respectively.
- **Bit Manipulation**: Since we know our matrix will only contain binary digits `0` and `1`, we can utilize an efficient operation known as XOR (`^`) for inverting bits; `0` becomes `1`, and `1` becomes `0`.

Here is a walkthrough of the code to clarify the implementation of these concepts:

```
for row in image: # Iterating over each row of the matrix
    i, j = 0, n - 1 # Setting pointers to the beginning and end of the row
    while i < j: # Loop until the two pointers(problems/two pointers intro) meet in the middle
        if row[i] == row[j]: # Only swap and invert when the elements are the same
            row[i] ^= 1 # Invert the 'i'-th element using XOR
            row[j] ^= 1 # Invert the 'j'-th element using XOR
        # If the elements are different, no action is required since flip and invert would keep them different
        i, j = i + 1, j - 1 # Move the pointers closer to the center

    if i == j: # If the row has an odd number of elements, invert the central element
        row[i] ^= 1
```

After the `for` loop runs for each row, all rows have been flipped and inverted according to the problem's constraints. The matrix `image` is now modified in place and is returned as the final result.

In this method, the code achieves `O(n^2)` time complexity, since each element is visited once, and `O(1)` extra space complexity, as we only use a few variables for iteration and no additional storage for the result.

Example Walkthrough

Let's apply the provided solution approach on a small `3x3` example matrix:

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

- We will process this matrix row by row.
- Each row operation includes flipping and inverting.
- The flipping and inverting are done using two pointers for each row, and an XOR operation is used for inverting.

Processing the first row: `[1, 1, 0]`

- Pointers `i` and `j` start at positions `0` and `2` respectively.
- Elements at `i` and `j` (`1` and `0`) are different – no operation necessary since they will be different even after flipping and inverting.
- Move `i` to `1` and `j` to `1` (pointers meet in the middle).
- Since `i == j`, we invert the element at the center using XOR. The updated row is `[1, 1, 0]`.

Processing the second row: `[1, 1, 0]`

- `i = 0` and `j = 2`.
- Elements at `i` and `j` (`1` and `0`) are different – again, no operation necessary.
- Move `i` to `1` and `j` to `1` (pointers meet in the middle).
- Since `i == j`, we invert the element at the center using XOR. The updated row is `[1, 0, 0]`.

Processing the third row: `[0, 1, 1]`

- `i = 0` and `j = 2`.
- Elements at `i` and `j` (`0` and `1`) are different – no operation necessary.
- Move `i` to `1` and `j` to `1` (pointers meet in the middle).
- Since `i == j`, we invert the element at the center using XOR. The updated row is `[0, 0, 1]`.

Here's the matrix after processing all the rows:

```
Flipped and Inverted Matrix:
1 1 0
1 0 0
0 0 1
```

- Notice that in this example, the outer elements of each row didn't need inverting because they were different from each other.
- The only elements that were inverted are the ones that ended up in the center after flipping (which in this case occurred for every row).
- The final modified matrix is the result of the flipping and inverting operations.

Solution Implementation

Python

```python
class Solution:
    def flip_and_invert_image(self, image):
        """
        Flips the image horizontally, then inverts it, and returns the resulting image.
        :param image: A list of lists representing a binary matrix.
        :return: The flipped and inverted image as a list of lists.
        """

        # Get the number of columns in the image
        num_columns = len(image)

        # Traverse each row of the image
        for row in image:
            # Set two pointers, starting from the beginning and the end of the row respectively
            left, right = 0, num_columns - 1

            # Continue swapping pixels until the pointers meet or cross
            while left < right:
                # If the values at the current pixels are the same, invert them
                if row[left] == row[right]:
                    row[left] = 1 - row[left]  # Flip the pixel value
                    row[right] = 1 - row[right]  # Flip the pixel value

                # Move the pointers towards the center
                left += 1
                right -= 1

            # If the number of columns is odd, flip the central pixel
            if left == right:
                row[left] = 1 - row[left]

        # Return the modified image
        return image

# Example usage:
# sol = Solution()
# image = [[1,1,0],[1,0,1],[0,0,0]]
# print(sol.flip_and_invert_image(image)) # Output: [[1,0,0],[0,1,0],[1,1,1]]
```

Java

```java
class Solution {

    // The method flips the image horizontally and then inverts it.
    public int[][] flipAndInvertImage(int[][] image) {
        // Loop over each row in the image
        for (int[] row : image) {
            // Initialize two pointers for the start and end of the row
            int start = 0, end = row.length - 1;

            // Continue swapping until the pointers meet or cross
            while (start <= end) {
                // If the elements at the start and end are the same, flip them
                if (row[start] == row[end]) {
                    row[start] ^= 1; // XOR with 1 will flip 0 to 1 and 1 to 0
                    row[end] = row[start]; // The flipped value is the same for start and end
                }
                // If start and end pointers are pointing to the same element,
                // that means there's an odd number of elements in the row and
                // we need to flip the middle element once.
                // Note that this will also be handled correctly in the above if block.
                if (start == end) {
                    row[start] ^= 1; // Flip the middle element
                }
                // Move the pointers inward
                start++;
                end--;
            }
        }
        // Return the image after flipping and inverting
        return image;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<vector<int>> flipAndInvertImage(vector<vector<int>>& image) {
        // Iterate over each row of the image
        for (auto& row : image) {
            // Initialize two pointers, i starting from the beginning and j from the end of the row
            int i = 0, j = row.size() - 1;
            // Loop until the two pointers meet or cross
            for (; i < j; ++i, --j) {
                // If the current elements at i and j are the same, flip them
                // Flipping is done by using XOR operation with 1
                if (row[i] == row[j]) {
                    row[i] ^= 1;
                    row[j] ^= 1;
                }
            }
            // If there is a middle element (odd number of elements), flip it
            if (i == j) {
                row[i] ^= 1;
            }
        }
        // Return the modified image after flipping and inverting
        return image;
    }
};
```

TypeScript

```typescript
/**
 * Flips the image horizontally, then inverts it (1's become 0's and vice versa).
 * @param {number[][]} image - A 2D array representing a binary image.
 * @return {number[][]} - The modified image after flipping and inverting
 */
function flipAndInvertImage(image: number[][]): number[][] {
    // Iterate over each row in the image
    for (const row of image) {
        let leftIndex = 0; // Starting index from the left side
        let rightIndex = row.length - 1; // Starting index from the right side

        // Flip and invert the row using two pointers approach
        for (; leftIndex < rightIndex; ++leftIndex, --rightIndex) {
            // If the left and right values are the same, invert them
            if (row[leftIndex] === row[rightIndex]) {
                row[leftIndex] ^= 1;
                row[rightIndex] ^= 1;
            }
        }
        // If we have an odd number of elements, invert the middle one
        if (leftIndex === rightIndex) {
            row[leftIndex] ^= 1;
        }
    }
    // Return the image after performing flip and invert
    return image;
}

// Example use:
// const resultImage = flipAndInvertImage([[1,1,0],[1,0,1],[0,0,0]]);
// console.log(resultImage);
```

```python
class Solution:
    def flip_and_invert_image(self, image):
        """
        Flips the image horizontally, then inverts it, and returns the resulting image.
        :param image: A list of lists representing a binary matrix.
        :return: The flipped and inverted image as a list of lists.
        """

        # Get the number of columns in the image
        num_columns = len(image)

        # Traverse each row of the image
        for row in image:
            # Set two pointers, starting from the beginning and the end of the row respectively
            left, right = 0, num_columns - 1

            # Continue swapping pixels until the pointers meet or cross
            while left < right:
                # If the values at the current pixels are the same, invert them
                if row[left] == row[right]:
                    row[left] = 1 - row[left]  # Flip the pixel value
                    row[right] = 1 - row[right]  # Flip the pixel value

                # Move the pointers towards the center
                left += 1
                right -= 1

            # If the number of columns is odd, flip the central pixel
            if left == right:
                row[left] = 1 - row[left]

        # Return the modified image
        return image

# Example usage:
# sol = Solution()
# image = [[1,1,0],[1,0,1],[0,0,0]]
# print(sol.flip_and_invert_image(image)) # Output: [[1,0,0],[0,1,0],[1,1,1]]
```

Time and Space Complexity

The time complexity of the code is `O(n^2)` where `n` is the length of each row of the `image`. This is because the algorithm iterates over each element of the image once. Each row has `n` elements, and there are `n` rows.

The space complexity of the code is `O(1)` since the operation is done in-place and does not require any additional space that is dependent on the input size. The variables `i`, `j`, and `n` use a constant amount of extra space.