531. Lonely Pixel I

Medium Array Hash Table Matrix Leetcode Link

Problem Description

The given problem presents a two-dimensional grid called `picture`, represented as an `m x n` matrix consisting of two types of pixels - black represented by `'B'` and white represented by `'W'`. The task is to calculate the number of black pixels that are "lonely", which means they do not share their row or column with any other black pixels.

Intuition

The solution involves two main steps.

Firstly, we need to find the count of black pixels in every row and every column. To accomplish this, we create two arrays: one called `rows` for storing the counts of black pixels in each row, and one called `cols` for storing the counts in each column. We iterate through every element in the `picture` matrix; if we encounter a black pixel (`'B'`), we increment the corresponding elements in `rows` and `cols`.

Once we have the counts, we check for lonely pixels. We scan through our `rows` array to find rows that contain exactly one black pixel. For each of these rows, we loop through the columns. If we find that the black pixel in this row also resides in a column with only one black pixel, then it is a lonely black pixel. We increment our result counter (`res`) in this case.

By breaking down the problem into counting and then verifying against those counts, we can identify lonely pixels without checking the entire grid every single time we find a black pixel. This approach significantly optimizes our solution.

Solution Approach

The proposed solution algorithm employs a straightforward but effective two-pass method with additional space to keep track of counts. The data structures used are two auxiliary arrays (`rows` and `cols`) to record the number of black pixels in each row and column, respectively.

First Pass: Counting Black Pixels

In the first pass, we iterate over every cell in the `picture` matrix using a nested loop, where the outer loop runs through the rows, and the inner loop runs through the columns. When we come across a black pixel (denoted by `'B'`), we increment the count for the current row in `rows[i]` and the current column in `cols[j]`. This way, each index in the `rows` array will hold the number of black pixels in the corresponding row of the `picture`, and similarly for the `cols` array regarding the columns.

Initialize Count Arrays

```python
rows, cols = [0] * m, [0] * n
```

Counting Black Pixels

```python
for i in range(m):
    for j in range(n):
        if picture[i][j] == 'B':
            rows[i] += 1
            cols[j] += 1
```

Second Pass: Identifying Lonely Pixels

In the second pass, we are looking for rows that contain a single black pixel, which can be checked efficiently thanks to the `rows` array. For each row with a single black pixel, we examine each of its columns using another loop. If we find that a black pixel in the row also has a corresponding `cols` count of 1, we have found a lonely black pixel and increment our result count `res`. A key optimization here is that once we find the lonely black pixel in a row, we break the inner loop since there can't be more than one lonely black pixel in a single row.

Checking Lonely Pixels and Counting

```python
res = 0
for i in range(m):
    if rows[i] == 1:
        for j in range(n):
            if picture[i][j] == 'B' and cols[j] == 1:
                res += 1
                break
```

Returning the Result

Finally, after both passes are completed, we return the result `res`, which holds the number of lonely black pixels discovered in the `picture`.

This algorithm effectively reduces the time complexity by ensuring that each element in the picture is visited only a fixed number of times, avoiding repeated scans of entire rows or columns. The first pass runs in O(m * n), and the second pass will, in the worst case, also run in O(m * n), making the overall time complexity O(m * n). The additional space used for the `rows` and `cols` arrays is O(m + n), which is acceptable given that it leads to a more time-efficient solution.

Example Walkthrough

Let's assume we have the following `3 x 3` grid representing our `picture`:

```
[['W', 'B', 'W'],
 ['W', 'W', 'W'],
 ['B', 'W', 'B']]
```

In this grid, 'W' represents white pixels and 'B' represents black pixels. We will refer to rows and columns starting from index 0.

First Pass: Count Black Pixels

First, we initialize our `rows` and `cols` arrays to record the black pixel counts. Since our picture is `3 x 3`, both arrays will have three elements initially set to 0.

```
rows = [0, 0, 0]
cols = [0, 0, 0]
```

As we iterate through the picture, whenever we see a 'B', we increment the corresponding `rows` and `cols` counts.

After the first pass, our count arrays will look like this:

```
rows = [1, 0, 2] // There's 1 black pixel in the first row, none in the second, and 2 in the third.
cols = [1, 0, 1] // Likewise, there's 1 black pixel in the first and third columns, and none in the second.
```

Second Pass: Identifying Lonely Pixels

Next, we check for rows with a single black pixel. In our case, the first row (`rows[0] == 1`) contains exactly one black pixel. Now let's check if it's a lonely pixel.

We check the column where that pixel resides; it's at `picture[0][1]`, which is in column 1. The count in `cols[1]` is 0 since there are no other black pixels in that column, indicating this pixel is not lonely. Therefore, we do not increment `res`.

Moving on to the third row, `rows[2] == 2`. This row contains two black pixels, so they cannot be lonely by definition, and we skip checking the columns for this row.

Finally, we finish our scan, with `res` remaining at 0, indicating there are no lonely black pixels in this particular picture.

The algorithm completes and returns the result `res` which, for this example, is 0.

Python Solution

```python
from typing import List

class Solution:
    def findLonelyPixel(self, picture: List[List[str]]) -> int:
        # Get the dimensions of the picture
        num_rows, num_cols = len(picture), len(picture[0])

        # Initialize the arrays to track the count of black pixels in each row and column
        row_black_counts = [0] * num_rows
        col_black_counts = [0] * num_cols

        # Count the number of black pixels in each row and column
        for i in range(num_rows):
            for j in range(num_cols):
                if picture[i][j] == 'B':
                    row_black_counts[i] += 1
                    col_black_counts[j] += 1

        # Initialize the result to count lonely pixels
        lonely_pixel_count = 0

        # Iterate over each row of the picture
        for i in range(num_rows):
            # Proceed only if the row contains exactly one black pixel
            if row_black_counts[i] == 1:
                # Look for the lonely black pixel in the row
                for j in range(num_cols):
                    # Check if the current pixel is black and also the only one in its column
                    if picture[i][j] == 'B' and col_black_counts[j] == 1:
                        # Increment the count of lonely pixels and break as there will be no more in this row
                        lonely_pixel_count += 1
                        break

        # Return the count of lonely black pixels in the picture
        return lonely_pixel_count

```

Java Solution

```java
class Solution {

    /**
     * Finds the number of lonely pixels (i.e. 'B's) in the picture array.
     * A lonely pixel is defined as a 'B' that is the only one in its row and column.
     *
     * @param picture A 2D character array representing the picture.
     * @return The number of lonely pixels.
     */
    public int findLonelyPixel(char[][] picture) {
        // Get dimensions of the picture array.
        int rowsCount = picture.length;
        int colsCount = picture[0].length;

        // Initialize arrays to count the number of 'B's in each row and column.
        int[] rowCount = new int[rowsCount];
        int[] colCount = new int[colsCount];

        // Iterate over the picture to count how many 'B's are in each row and column.
        for (int i = 0; i < rowsCount; ++i) {
            for (int j = 0; j < colsCount; ++j) {
                if (picture[i][j] == 'B') {
                    rowCount[i]++;
                    colCount[j]++;
                }
            }
        }

        // Initialize the result variable to store the count of lonely pixels.
        int lonelyPixelsCount = 0;

        // Iterate over the row count array to find rows with exactly one 'B'.
        for (int i = 0; i < rowsCount; ++i) {
            if (rowCount[i] == 1) {
                // If a row has exactly one 'B', check columns for a lonely 'B'.
                for (int j = 0; j < colsCount; ++j) {
                    // Confirm that the 'B' is lonely (i.e., it's the only one in its row and column).
                    if (picture[i][j] == 'B' && colCount[j] == 1) {
                        lonelyPixelsCount++;

                        // Break since we found the lonely 'B' for this row, no need to check further.
                        break;
                    }
                }
            }
        }

        // Return the total count of lonely pixels found.
        return lonelyPixelsCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Finds the total count of lonely pixels ('B') in a grid.
    // A 'lonely' pixel is defined as a 'B' that is the only one in its row and column.
    int findLonelyPixel(vector<vector<char>>& picture) {
        int rowCount = picture.size();
        int colCount = picture[0].size();
        vector<int> rowCounts(rowCount, 0); // Stores counts of 'B's in each row
        vector<int> colCounts(colCount, 0); // Stores counts of 'B's in each column

        // First pass to calculate the row and column counts
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                if (picture[i][j] == 'B') {
                    ++rowCounts[i];
                    ++colCounts[j];
                }
            }
        }

        int lonelyPixels = 0; // Will hold the total count of lonely pixels

        // Second pass to identify lonely pixels
        for (int i = 0; i < rowCount; ++i) {
            // We only proceed if a row has a single 'B'
            if (rowCounts[i] == 1) {
                for (int j = 0; j < colCount; ++j) {
                    // Check if the current 'B' is lonely, that is, it is the only one its column as well
                    if (picture[i][j] == 'B' && colCounts[j] == 1) {
                        ++lonelyPixels;
                        break; // Break because we found the lonely pixel for this row
                    }
                }
            }
        }

        return lonelyPixels;
    }
};

```

Typescript Solution

```typescript
// Stores the count of lonely pixels in a two-dimensional character grid.
// A lonely pixel is defined as one that is the only 'B' in both its row and column.

// Function to find the total count of lonely pixels ('B') in a grid.
function findLonelyPixel(picture: char[][]): number {
    let rowCount = picture.length; // Number of rows in the grid
    let colCount = picture[0].length; // Number of columns in the grid
    let rowCounts: number[] = new Array(rowCount).fill(0); // Array to store the count of 'B's in each row
    let colCounts: number[] = new Array(colCount).fill(0); // Array to store the count of 'B's in each column

    // First pass to calculate the 'B' count for each row and column
    for (let i = 0; i < rowCount; ++i) {
        for (let j = 0; j < colCount; ++j) {
            if (picture[i][j] === 'B') {
                rowCounts[i]++;
                colCounts[j]++;
            }
        }
    }

    let lonelyPixels = 0; // Initialize count of lonely pixels

    // Second pass to identify and count lonely pixels
    for (let i = 0; i < rowCount; ++i) {
        // Skip rows that do not contain exactly one 'B'
        if (rowCounts[i] === 1) {
            for (let j = 0; j < colCount; ++j) {
                // A 'B' is lonely if it's the only one in its column
                if (picture[i][j] === 'B' && colCounts[j] === 1) {
                    lonelyPixels++; // Increment the count of lonely pixels
                    break; // Move to the next row since we found a lonely pixel
                }
            }
        }
    }

    // Return the final count of lonely pixels
    return lonelyPixels;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be determined by analyzing the two nested loops that iterate through the `m x n` grid of the picture.

- The first pair of nested loops goes through each cell of the picture to count the number of 'B' in each row and column. This part of the code has a complexity of `O(m * n)`, as it needs to visit each cell exactly once.
- The second pair of nested loops goes through each row and, if it finds a row with exactly one 'B', it then iterates through the columns of that row to find a column with exactly one 'B'. In the worst case, this might seem like another `O(m * n)` operation. However, the inner loop breaks as soon as it finds the lonely 'B', and because there can be at most `min(m, n)` lonely pixels (since each requires a unique row and a unique column), the total time for this set of loops is also bounded by `O(m * n)`.

Thus, when adding both parts, the overall time complexity remains `O(m * n)` because the two parts are not nested within one another but are sequential.

Space Complexity

The space complexity of the code is derived from the extra space used to store the `rows` and `cols` counts.

- The `rows` array uses `O(m)` space, and the `cols` array uses `O(n)` space.

Since no other significant space is used by the algorithm, the total space complexity is `O(m + n)`.