1253. Reconstruct a 2-Row Binary Matrix

Medium Prefix Suffix Matrix

Leetcode Link

Problem Description

The problem provides us with criteria to reconstruct a 2-by-n binary matrix based on,

1. upper: the sum of the elements in the first row,
2. lower: the sum of the elements in the second row, and
3. colsum: an array where each value represents the sum of elements in that column.

Each element in the matrix can only be `0` or `1`. The goal is to reconstruct the original matrix such that the rows and columns sum up to the given `upper`, `lower`, and `colsum` values respectively. If multiple reconstructions are possible, any one of them is considered a valid solution. If no valid matrix can be reconstructed, the function should return an empty array.

Intuition

The intuition behind the solution is based on addressing the constraints one by one:

1. Since colsum sums are provided, we start by setting the elements of each column. If `colsum[i]` is `2`, it means that both the upper and lower elements in column `i` must be `1`. This is the only way to achieve a column sum of `2` in a binary matrix. By doing this, we also decrement both `upper` and `lower` counts by `1`.

2. For columns where `colsum[i]` is `0`, we have the option to choose which row to place the `1`. Here we use the current value of `upper` and `lower` as a heuristic to decide. If `upper` is larger, we place the `1` in the upper row (and decrement `upper`), else we put it in the lower row (and decrement `lower`). This heuristic aims to balance the number of `1`s between the upper and lower rows throughout the process.

3. If at any point `upper` or `lower` becomes negative, this indicates that the given `upper`, `lower`, and `colsum` cannot lead to a valid matrix, therefore, we return an empty array.

4. After processing all columns, to ensure the sums are correct, `upper` and `lower` should both be `0` - if not, it means the sums do not add up appropriately, hence we return an empty array.

5. If all constraints are met, we return the reconstructed matrix.

This one-pass solution efficiently reconstructs the matrix (or reports impossibility) by incrementally building the solution while honoring the constraints presented by `upper`, `lower`, and `colsum`. Through careful manipulation of the matrix and tracking of `upper` and `lower`, we either find a valid configuration or recognize the problem as unsolvable.

Solution Approach

The implementation of the solution can be broken down into the following steps:

1. Initialize an `n x 2` matrix called `ans` to represent the reconstructed matrix, where `n` is the length of the `colsum` array, with all elements set to `0`.

2. Iterate through each element `v` of the `colsum` array by its index `j`. During each iteration:

   - If `v` is `2`, set both `ans[0][j]` and `ans[1][j]` to `1`. This is because the only way a column sum can be `2` in a binary matrix is if both rows have a `1` in that column. After setting these values, decrement both `upper` and `lower` by `1`.
   - If `v` is `1`, a decision needs to be made about which row to place the `1`. If `upper` is greater than `lower`, put the `1` in the upper row (`ans[0][j]`) and decrement `upper` by `1`. Otherwise, place it in the lower row (`ans[1][j]`) and decrement `lower` by `1`.
   - During both of the above operations, if `upper` or `lower` becomes negative, it indicates an inconsistency with the provided sums, meaning there's no possible way to reconstruct a valid matrix. Thus, return an empty array immediately.

3. After filling in the matrix based on the column sums, validate that both `upper` and `lower` are exactly `0`. This ensures that the constructed matrix satisfies the sum conditions for both rows. If either `upper` or `lower` is not `0`, return an empty array as it indicates an invalid solution.

4. If the algorithm passes all the checks, `ans` is a valid reconstruction of the matrix, so return the `ans` matrix.

The above approach does not require any complex data structures or algorithms; it simply utilizes an iterative logic that addresses the constraints provided by the problem statement. Here's a summary of the key patterns and data structures involved:

- **Pattern:** Greedy choice/Decision making – placing `1` optimistically based on the current state (`upper` vs `lower`) to maintain the balance between `upper` and `lower`.
- **Data Structure:** Two-dimensional list – to store the reconstructed matrix.
- **Algorithm:** Single-pass iteration – moving through the `colsum` array once and making decisions that adhere to the problem's constraints.

By adhering to these steps, the implementation successfully reconstructs a binary matrix that matches the defined conditions or recognizes when no such matrix can be created.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we're given `upper = 2`, `lower = 2`, and `colsum = [2, 1, 1, 0, 1]`.

Here's how we would apply the solution approach:

1. Initialize the `ans` matrix to be the same length as `colsum` with all zeros: `ans = [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]`.

2. Iterate through each element in `colsum`:

   - For `colsum[0]` which equals `2`, set `ans[0][0]` and `ans[1][0]` to `1`. After this, decrement both `upper` and `lower` by 1: `upper = 1`, `lower = 1`.
   - For `colsum[1]` which equals `1`, since `upper` is not less than `lower`, set `ans[0][1]` to `1` and decrement `upper` by 1: `upper = 0`.
   - For `colsum[2]` which equals `1`, now since `upper` is equal to `lower`, we can decide to place the `1` in either row. Let's choose the lower row for (a change). Set `ans[1][2]` to `1` and decrement `lower` by 1: `lower = 0`.
   - `colsum[3]` is `0`, so we do not change anything for this column.
   - Lastly, for `colsum[4]` which equals `1`, `upper` is `0` and `lower` is `1`. Therefore, we set `ans[1][4]` to `1` and decrement `lower` by 1: `lower = 0`.

3. After processing all columns, check `upper` and `lower`. Both are `0`, so the matrix satisfies the given row sums.

4. The final `ans` matrix is valid and matches the defined conditions. Therefore, we return `ans` matrix which is `[[1, 1, 0, 0, 0], [1, 0, 1, 0, 1]]`.

The reconstructed matrix now successfully represents a valid solution because:

- The sum of the upper row is `2`, which matches the given `upper = 2`.
- The sum of the lower row is `2`, which matches the given `lower = 2`.
- The column sums `[2, 1, 1, 0, 1]` match the given array `colsum`.

By following the described solution approach, we were able to reconstruct a matrix that fulfills both the individual column sums and the overall row sums.

Python Solution

```python
from typing import List

class Solution:
    def reconstructMatrix(self, upper: int, lower: int, col_sum: List[int]) -> List[List[int]]:
        # Calculate the number of columns based on the col_sum list length
        num_cols = len(col_sum)
        # Initialize a 2xN matrix with zeroes
        ans_matrix = [[0] * num_cols for _ in range(2)]

        # Iterate over each value in the col_sum list
        for idx, sum_val in enumerate(col_sum):
            # If the sum for a column is 2, place a 1 in both upper and lower rows
            if sum_val == 2:
                ans_matrix[0][idx] = ans_matrix[1][idx] = 1
                upper -= 1
                lower -= 1

            # If the sum is 1, decide to place it in the row with the larger remaining count
            elif sum_val == 1:
                if upper > lower:
                    upper -= 1
                    ans_matrix[0][idx] = 1
                else:
                    lower -= 1
                    ans_matrix[1][idx] = 1

            # If at any point the remaining count for upper or lower becomes negative,
            # it is impossible to construct a valid matrix, so we return an empty list
            if upper < 0 or lower < 0:
                return []

        # After filling the matrix, if both the upper and lower sums are reduced to zero,
        # we have a valid matrix; otherwise, return an empty list
        return ans_matrix if upper == 0 and lower == 0 else []

# Example usage
# solution = Solution()
# upper = 2
# lower = 2
# col_sum = [2, 1, 1, 1]
# print(solution.reconstructMatrix(upper, lower, col_sum))
```

Java Solution

```java
class Solution {

    // Function to reconstruct a 2-row binary matrix based on column sum indicators and given upper/lower row sum targets
    public List<List<Integer>> reconstructMatrix(int upperSum, int lowerSum, int[] columnSum) {
        // Length of the column.
        int numColumns = columnSum.length;

        // Initializing the lists that will represent the two rows of the matrix.
        List<Integer> upperRow = new ArrayList<>();
        List<Integer> lowerRow = new ArrayList<>();

        // Iterate through each column to build the two rows.
        for (int colSum : columnSum) {
            int upperValue = 0, lowerValue = 0;

            // If the column sum is 2, then both rows must have a 1 in this column.
            if (colSum[colSum] == 2) {
                upperValue = 1;
                lowerValue = 1;
                // Decrement the upper row sum.
                upperSum--;
                // Decrement the lower row sum.
                lowerSum--;
            }
            // If the column sum is 1, determine which row to place the 1 in.
            else if (columnSum[colSum] == 1) {
                // Preference is given to the upper row if its sum is greater than the lower row's sum.
                if (upperSum > lowerSum) {
                    upperValue = 1;
                    // Decrement the upper row sum.
                    upperSum--;
                } else {
                    lowerValue = 1;
                    // Decrement the lower row sum.
                    lowerSum--;
                }
            }

            // Check for an invalid state; if either sum becomes negative, the solution is not feasible.
            if (upperSum < 0 || lowerSum < 0) {
                break;
            }

            // Add values to their respective rows.
            upperRow.add(upperValue);
            lowerRow.add(lowerValue);
        }

        // After processing all columns, if both sums are zero, a valid solution has been found.
        return (upperSum == 0 && lowerSum == 0) ? List.of(upperRow, lowerRow) : List.of();
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to reconstruct a 2-row matrix based on the column sum and individual row sums.
    vector<vector<int>> reconstructMatrix(int upperSum, int lowerSum, vector<int>& columnSum) {
        int n = columnSum.size(); // The total number of columns.

        // Initialize a 2D vector with dimensions 2 x n, filled with zeroes.
        vector<vector<int>> reconstructedMatrix(2, vector<int>(n));

        // Iterate through each column to build the reconstructed matrix.
        for (int col = 0; col < n; ++col) {
            // If the sum for the current column is 2,
            // set both rows in the current column to 1.
            if (columnSum[col] == 2) {
                reconstructedMatrix[0][col] = 1;
                reconstructedMatrix[1][col] = 1;
                upperSum--; // Decrement upper row sum.
                lowerSum--; // Decrement lower row sum.
            }

            // If the sum for the current column is 1,
            // set only one row in the current column to 1.
            if (columnSum[col] == 1) {
                // Prefer placing a 1 in the upper row if the upper sum is greater.
                if (upperSum > lowerSum) {
                    reconstructedMatrix[0][col] = 1;
                } else {
                    // Otherwise, place a 1 in the lower row.
                    lowerSum--;
                    reconstructedMatrix[1][col] = 1;
                }
            }

            // If at any point the remaining sums for upper or lower rows become negative,
            // it indicates that a valid reconstruction is not possible.
            if (upperSum < 0 || lowerSum < 0) {
                break;
            }
        }

        // If after processing all columns, the remaining sums for upper or lower rows are not zero,
        // return an empty matrix as it means a reconstruction was not possible.
        // Otherwise, return the successfully reconstructed matrix.
        return (upperSum == 0 && lowerSum == 0) ? reconstructedMatrix : vector<vector<int>>();
    }
};
```

Typescript Solution

```typescript
function reconstructMatrix(upperSum: number, lowerSum: number, columnSums: number[]): number[][] {
    const numColumns = columnSums.length; // Number of columns in the matrix
    const matrix: number[][] = Array(2)
        .fill(0)
        .map(() => Array(numColumns).fill(0)); // Initialize a 2-row matrix with zeros

    // Iterate through each column
    for (let i = 0; i < numColumns; ++i) {
        if (columnSums[i] === 2) {
            // If the column sum is 2, place 1s in both upper and lower rows
            matrix[0][i] = matrix[1][i] = 1;
            upperSum--; // Decrement the upper row sum
            lowerSum--; // Decrement the lower row sum
        } else if (columnSums[i] === 1) {
            // If the column sum is 1, decide which row to place a 1 in based on the remaining sums
            if (upperSum > lowerSum) {
                matrix[0][i] = 1;
                upperSum--;
            } else {
                matrix[1][i] = 1;
                lowerSum--;
            }
        }
        // If at any point the sum for upper or lower rows becomes negative, it's not possible to construct the matrix
        if (upperSum < 0 || lowerSum < 0) {
            return []; // Return an empty array in case of failure
        }
    }
    // Check if all sums have been fully utilized
    return upperSum === 0 && lowerSum === 0 ? matrix : [];
}

// Example usage:
// const result = reconstructMatrix(2, 3, [2, 2, 1, 1]);
// console.log(result);
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the single loop that iterates through the `colsum` list. Assuming that the length of `colsum` is n, the loop runs n times. Since the operations inside the loop (assignment, arithmetic operations, and comparisons) are all constant time operations, the time complexity of the loop is $O(n)$. Therefore, the overall time complexity of the function is $O(n)$.

Space Complexity

The space complexity is determined by the additional space used by the program that is not part of the input. The main additional space used in the function is for storing the answer in the `ans` list, which is a 2 by n matrix. Thus, the space used by `ans` is $O(2 * n)$ which simplifies to $O(n)$. Other variables used in the function (like `upper`, `lower`, and the loop counter `j`) use a constant space $O(1)$. Therefore, the overall space complexity of the function is $O(n)$.