

# 1253. Reconstruct a 2-Row Binary Matrix

## Problem Description

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

- upper: the sum of the elements in the first row,
- lower: the sum of the elements in the second row, and
- 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:

- Since column 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.
- For columns where colsum[i] is 1, 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 1s between the upper and lower rows throughout the process.
- 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.
- 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.
- 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:

- 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.
- 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 can sum to 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.
- 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.
- 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:

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

```
1 from typing import List
2
3 class Solution:
4     def reconstructMatrix(self, upper: int, lower: int, col_sum: List[int]) -> List[List[int]]:
5         # Calculate the number of columns based on the col_sum list length
6         num_cols = len(col_sum)
7         # Initialize a 2xN matrix with zeroes
8         ans_matrix = [[0] * num_cols for _ in range(2)]
9
10        # Iterate over each value in the col_sum list
11        for idx, sum_val in enumerate(col_sum):
12            # If the sum for a column is 2, place a 1 in both upper and lower rows
13            if sum_val == 2:
14                ans_matrix[0][idx] = ans_matrix[1][idx] = 1
15                upper -= 1
16                lower -= 1
17
18            # If the sum is 1, decide to place it in the row with the larger remaining count
19            elif sum_val == 1:
20                if upper > lower:
21                    upper -= 1
22                    ans_matrix[0][idx] = 1
23                else:
24                    lower -= 1
25                    ans_matrix[1][idx] = 1
26
27            # If at any point the remaining count for upper or lower becomes negative,
28            # it is impossible to construct a valid matrix, so we return an empty list
29            if upper < 0 or lower < 0:
30                return []
31
32        # After filling the matrix, if both the upper and lower sums are reduced to zero,
33        # we have a valid matrix; otherwise, return an empty list
34        return ans_matrix if upper == 0 and lower == 0 else []
35
36 # Example usage
37 solution = Solution()
38 upper = 2
39 lower = 3
40 col_sum = [2, 2, 1, 1]
41 print(solution.reconstructMatrix(upper, lower, col_sum))
42
```

## Java Solution

```
1 class Solution {
2
3     // Function to reconstruct a 2-row binary matrix based on column sum indicators and given upper/lower row sum targets
4     public List<List<Integer>> reconstructMatrix(int upperSum, int lowerSum, int[] columnSum) {
5         // Length of the column.
6         int numColumns = columnSum.length;
7
8         // Initializing the lists that will represent the two rows of the matrix.
9         List<Integer> upperRow = new ArrayList<>();
10        List<Integer> lowerRow = new ArrayList<>();
11
12        // Iterate through each column to build the two rows.
13        for (int col = 0; col < numColumns; ++col) {
14            int upperValue = 0, lowerValue = 0;
15
16            // If the column sum is 2, then both rows must have a 1 in this column.
17            if (columnSum[col] == 2) {
18                upperValue = 1;
19                lowerValue = 1;
20                upperSum--; // Decrement the upper row sum.
21                lowerSum--; // Decrement the lower row sum.
22            }
23            // If the column sum is 1, determine which row to place the 1 in.
24            else if (columnSum[col] == 1) {
25                // Preference is given to the upper row if its sum is greater than the lower row's sum.
26                if (upperSum > lowerSum) {
27                    upperValue = 1;
28                    upperSum--; // Decrement the upper row sum.
29                } else {
30                    lowerValue = 1;
31                    lowerSum--; // Decrement the lower row sum.
32                }
33            }
34
35            // Check for an invalid state; if either sum becomes negative, the solution is not feasible.
36            if (upperSum < 0 || lowerSum < 0) {
37                break;
38            }
39
40            // Add values to their respective rows.
41            upperRow.add(upperValue);
42            lowerRow.add(lowerValue);
43        }
44
45        // After processing all columns, if both sums are zero, a valid solution has been found.
46        return (upperSum == 0 && lowerSum == 0) ? List.of(upperRow, lowerRow) : List.of();
47    }
48 }
49
```

## C++ Solution

```
1 class Solution {
2 public:
3     // Function to reconstruct a 2-row matrix based on the column sum and individual row sums.
4     vector<vector<int>> reconstructMatrix(int upperSum, int lowerSum, vector<int>& columnSum) {
5         int n = columnSum.size(); // The total number of columns.
6
7         // Initialize a 2D vector with dimensions 2 x n, filled with zeros.
8         vector<vector<int>> reconstructedMatrix(2, vector<int>(n));
9
10        // Iterate through each column to build the reconstructed matrix.
11        for (int col = 0; col < n; ++col) {
12            // If the sum for the current column is 2,
13            // set both rows in the current column to 1.
14            if (columnSum[col] == 2) {
15                reconstructedMatrix[0][col] = 1;
16                reconstructedMatrix[1][col] = 1;
17                upperSum--; // Decrement upper row sum.
18                lowerSum--; // Decrement lower row sum.
19            }
20
21            // If the sum for the current column is 1,
22            // set only one row in the current column to 1.
23            if (columnSum[col] == 1) {
24                if (upperSum > lowerSum) {
25                    // Prefer placing a 1 in the upper row if the upper sum is greater.
26                    upperSum--;
27                    reconstructedMatrix[0][col] = 1;
28                } else {
29                    // Otherwise, place a 1 in the lower row.
30                    lowerSum--;
31                    reconstructedMatrix[1][col] = 1;
32                }
33            }
34
35            // If at any point the remaining sums for upper or lower rows become negative,
36            // it indicates that a valid reconstruction is not possible.
37            if (upperSum < 0 || lowerSum < 0) {
38                break;
39            }
40        }
41
42        // If after processing all columns, the remaining sums for upper or lower rows are not zero,
43        // return an empty matrix as it means a reconstruction was not possible.
44        // Otherwise, return the successfully reconstructed matrix.
45        return (upperSum == 0 && lowerSum == 0) ? reconstructedMatrix : vector<vector<int>>();
46    }
47 };
48
```

## Typescript Solution

```
1 function reconstructMatrix(upperSum: number, lowerSum: number, columnSums: number[]): number[][] {
2     const numColumns = columnSums.length; // Number of columns in the matrix
3     const matrix: number[][] = Array(2)
4         .fill(0)
5         .map(() => Array(numColumns).fill(0)); // Initialize a 2-row matrix with zeros
6
7     // Iterate through each column
8     for (let j = 0; j < numColumns; ++j) {
9         if (columnSums[j] === 2) {
10            // If the column sum is 2, place 1s in both upper and lower rows
11            matrix[0][j] = matrix[1][j] = 1;
12            upperSum--; // Decrement the upper row sum
13            lowerSum--; // Decrement the lower row sum
14        } else if (columnSums[j] === 1) {
15            // If the column sum is 1, decide which row to place a 1 in based on the remaining sums
16            if (upperSum > lowerSum) {
17                matrix[0][j] = 1;
18                upperSum--;
19            } else {
20                matrix[1][j] = 1;
21                lowerSum--;
22            }
23        }
24        // If at any point the sum for upper or lower rows becomes negative, it's not possible to construct the matrix
25        if (upperSum < 0 || lowerSum < 0) {
26            return []; // Return an empty array in case of failure
27        }
28    }
29    // Check if all sums have been fully utilized
30    return upperSum === 0 && lowerSum === 0 ? matrix : [];
31 }
32
33 // Example usage:
34 // const result = reconstructMatrix(2, 3, [2, 2, 1, 1]);
35 // console.log(result);
36
```

## 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).