1605. Find Valid Matrix Given Row and Column Sums

Problem Description

Greedy Array

Matrix

Medium

- Tobiciii Descriptio

goal is to reconstruct any 2D matrix that has non-negative integers and satisfies the given rowSum and colSum. The size of the matrix will be rowSum.length rows and colSum.length columns. You are guaranteed that there is at least one valid matrix that meets the requirements.

Intuition

To approach this problem, we can iterate through the matrix, starting from the top-left cell, and assign each cell a value that is

In this problem, you are provided with two arrays, rowSum and colSum. These arrays represent the sum of the elements in each

row and column, respectively, of a 2D matrix. However, you are not given the actual values in the matrix, only these sums. Your

the minimum between the current row's sum and the current column's sum. The intuition behind this strategy is that by assigning

By iteratively updating the sums, we are accounting for the value we've placed in each cell. When the minimum value is chosen, it reduces the remaining row or column sum without overshooting the required sum. After an allocation, we deduct the chosen value from both sums. This strategy preserves the constraints for every step since we only allocate as much as we can 'afford' from both the row and the column at that point.

the minimum value, we are ensuring that we never exceed the sum requirements for either the row or the column.

fulfills both row and column sum requirements because we've continuously deduct the allocated numbers from our sum arrays, ensuring that the end state satisfies the original conditions.

Solution Approach

The process continues until we've filled the entire matrix. At the end of this process, we have constructed a valid matrix that

The algorithm for the solution takes advantage of a simple <u>greedy</u> approach and basic properties of matrices. The following steps are involved in the solution:

Initialize the matrix ans to be of size $m \times n$, where m is the number of items in rowSum, and n is the number of items in

2. Iterate through each element of matrix ans, index by i for the rows and j for the columns.

colSum. All elements in ans are initialized to zero.

3. For each element ans[i][j], calculate the minimum value between rowSum[i] and colSum[j], which indicates the allowable value for this position that satisfies both the row's and column's sum constraints.

4. Assign ans [i] [j] this minimum value, which is the greedy choice of allocating as much as possible to the current cell without exceeding the sums for its row and column.

Decrease the value of rowSum[i] and colSum[j] by the amount just allocated to ans[i][j]. This step is important as it

updates the remaining sums for the row and column, reflecting the fact that we've already used part of those sums for

- element ans[i][j].

 6. Continue this process throughout the entire matrix, filling in each element with the calculated value.
- Data Structures:
 A 2D array ans to store the matrix values that we are building.
- The input arrays rowSum and colSum are modified in place to keep track of the remaining sums after each allocation.
 Patterns and Properties Used:
 Greedy approach: By constantly making the locally optimal choice of picking the minimum possible value for each cell, we

This approach guarantees that by the end, when all cells are filled, the remaining sums for both rows and columns will be zero,

indicating that we've met all the rowSum and colSum constraints correctly. Our solution effectively constructs a valid matrix with

To illustrate the solution approach, let's consider a small example where we have rowSum = [5, 7] and colSum = [8, 4]. We

Matrix properties: The solution exploits the property that each row and column in the matrix has a fixed sum, and any
allocation in an individual cell affects both the row's and column's sum.

build up to a globally optimal solution that meets all conditions.

need to construct a matrix that satisfies these row and column sums.

rowSum and colSum remain the same since the allocated number is 0.

non-negative integers that fits the given row and column sums.

Example Walkthrough

- We'll follow the steps outlined in the solution approach:

 1. Initialize ans as a 2×2 matrix with all elements set to zero (since rowSum.length is 2 and colSum.length is 2):
- [0, 0], [0, 0]]

Compute the minimum between rowSum[i] and colSum[j]. For ans[0][0], this minimum is min(5, 8) which equals 5.

4. Assign ans [0] [0] the value 5:

[5, 0],

[0, 0]

ans = [

ans = [

[5, 0],

[5, 0],

[3, 0]

matrix is:

[5, 0],

requirements.

[3, 4]

ans = [

Python

Java

class Solution {

class Solution:

ans[1][0] is set to 3.

New rowSum = [0, 4]

New colSum = [0, 4]

ans [1] [1] is set to 4.

New rowSum = [0, 0]

New colSum = [0, 0]

Update rowSum and colSum:

Update rowSum and colSum:

[0, 0]

ans = [

5. Subtract the allocated value from rowSum[0] and colSum[0]:

Start iterating through the elements of the matrix. We begin with i = 0 (first row) and j = 0 (first column).

```
New rowSum = [0, 7]
New colSum = [3, 4]
Repeat steps 2-5 for the next element, ans [0] [1]:
The minimum of rowSum[0] and colSum[1] is min(0, 4) which equals 0.
ans [0] [1] is set to 0.
```

3. Finally for ans[1][1], the minimum of rowSum[1] and colSum[1] is min(4, 4) equals 4.

For ans [1] [0], we take the minimum of rowSum[1] and colSum[0], which is min(7, 3) equals 3.

- ans = [
 [5, 0],
 [3, 4]
- And thus we have reconstructed any 2D matrix using the solution approach, successfully meeting the given rowSum and colSum

Solution Implementation

Iterate through each cell in the matrix
for i in range(num rows):
 for j in range(num cols):
 # Determine the value at matrix[i][i] by taking the minimum between

def restoreMatrix(self, rowSums: List[int], colSums: List[int]) -> List[List[int]]:

Initialize a matrix filled with zeros with dimensions (num_rows x num_cols)

the current row sum and column sum, to satisfy both constraints

Return the restored matrix that satisfies the row and column sums

// Get the number of rows and columns from the size of the passed arrays

// Return the restored matrix that matches the given row and column sums

// Restores the matrix given the sum of every row and every column.

public int[][] restoreMatrix(int[] rowSum, int[] colSum) {

Subtract the chosen value from the respective row sum and column sum

Get the number of rows and columns from the given sums

num_rows, num_cols = len(rowSums), len(colSums)

matrix = [[0] * num_cols for _ in range(num_rows)]

value = min(rowSums[i], colSums[j])

matrix[i][j] = value

rowSums[i] -= value

colSums[j] -= value

int rowCount = rowSum.length;

int colCount = colSum.length;

rowSum[row] -= value;

colSum[col] -= value;

#include <algorithm> // to use the std::min function

// Restores a matrix based on given row and column sums.

// - row sum represents the sum of the elements for each row.

// - col sum represents the sum of the elements for each column.

// Assign that minimum value to the current matrix cell

Get the number of rows and columns from the given sums

num_rows, num_cols = len(rowSums), len(colSums)

Iterate through each cell in the matrix

matrix = [[0] * num_cols for _ in range(num_rows)]

value = min(rowSums[i], colSums[j])

// Update the remaining sum for the current row and column

def restoreMatrix(self, rowSums: List[int], colSums: List[int]) -> List[List[int]]:

Initialize a matrix filled with zeros with dimensions (num_rows x num_cols)

Determine the value at matrix[i][i] by taking the minimum between

Subtract the chosen value from the respective row sum and column sum

the current row sum and column sum, to satisfy both constraints

Return the restored matrix that satisfies the row and column sums

matrix[i][j] = minSum;

rowSums[i] -= minSum;

colSums[j] -= minSum;

// Return the constructed matrix

for i in range(num rows):

for j in range(num cols):

matrix[i][j] = value

rowSums[i] -= value

colSums[j] -= value

return matrix;

class Solution:

return restoredMatrix;

#include <vector>

class Solution {

public:

return matrix

At this point, we have filled the matrix while satisfying the row and column sums described by rowSum and colSum. The final

```
// Initialize the matrix to return
int[][] restoredMatrix = new int[rowCount][colCount];

// Iterate over each cell in the matrix to calculate its value
for (int row = 0; row < rowCount; ++row) {
    for (int col = 0; col < colCount; ++col) {
        // Determine the minimum value to satisfy both rowSum and colSum constraints
        int value = Math.min(rowSum[row], colSum[col]);
        restoredMatrix[row][col] = value;</pre>
```

// After setting the value for restoredMatrix[row][col], subtract it from

std::vector<std::vector<int>> restoreMatrix(std::vector<int>& row_sum, std::vector<int>& col_sum) {

// the corresponding rowSum and colSum to maintain the sum constraints

```
int rows = row sum.size(); // Number of rows in the matrix.
        int cols = col sum.size(); // Number of columns in the matrix.
        std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols));
        // Iterate through each cell of the matrix
        for (int i = 0; i < rows; ++i) {
            for (int i = 0; i < cols; ++i) {
                // Find the minimum value between the current row sum and column sum.
                int value = std::min(row sum[i], col sum[i]);
                matrix[i][j] = value; // Assign the calculated value to the current cell.
                // Subtract the assigned value from both row and column sums.
                row sum[i] -= value;
                col_sum[j] -= value;
        // The matrix is now restored such that it meets the row and column sums criteria.
        return matrix;
};
TypeScript
function restoreMatrix(rowSums: number[], colSums: number[]): number[][] {
    const numRows = rowSums.length; // Number of rows based on rowSums length
    const numCols = colSums.length; // Number of columns based on colSums length
    // Initialize a 2D array 'matrix' with dimensions numRows x numCols filled with zeros
    const matrix: number[][] = Array.from({ length: numRows }, () => new Array(numCols).fill(0));
    // Iterate over each cell in the matrix
    for (let i = 0; i < numRows; i++) {</pre>
        for (let i = 0; i < numCols; i++) {</pre>
            // Determine the minimum value between the current row sum and column sum
            const minSum = Math.min(rowSums[i], colSums[j]);
```

Time and Space Complexity Time Complexity

return matrix

because the code contains two nested loops: the outer loop iterating over each row, and the inner loop iterating over each column. On each iteration, it performs a constant amount of work (calculating the minimum and updating rowSum and colSum).

Space Complexity

The time complexity of the given code is 0(m * n), where m is the number of rows and n is the number of columns. This is

```
The space complexity of the code is 0(m * n), this space is used to store the answer matrix ans. Other than that, the only extra space used is a constant amount for variables like x, i, j, and the space to copy and update the rowSum and colSum arrays, which does not depend on the size of the input and thus does not increase the order of space complexity.
```