1605. Find Valid Matrix Given Row and Column Sums

Medium Greedy Array Matrix

Leetcode Link

Problem Description

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 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 the minimum between the current row's sum and the current column's sum. The intuition behind this strategy is that by assigning the minimum value, we are ensuring that we never exceed the sum requirements for either the row or the column.

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 process continues until we've filled the entire matrix. At the end of this process, we have constructed a valid matrix that 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 algorithm for the solution takes advantage of a simple greedy approach and basic properties of matrices. The following steps are involved in the solution:

1. Initialize the matrix `ans` to be of size `m x n`, where `m` is the number of items in `rowSum`, and `n` is the number of items in `colSum`. All elements in `ans` are initialized to zero.
2. Iterate through each element of matrix `ans`, index by `i` for the rows and `j` for the columns.
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.
5. 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 build up to a globally optimal solution that meets all conditions.
- 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.

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

Example Walkthrough

To illustrate the solution approach, let's consider a small example where we have `rowSum = [5, 7]` and `colSum = [8, 4]`. We need to construct a matrix that satisfies these row and column sums.

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):

```
ans = [
  [0, 0],
  [0, 0]
]
```

2. Start iterating through the elements of the matrix. We begin with `i = 0` (first row) and `j = 0` (first column).
3. 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:

```
ans = [
  [5, 0],
  [0, 0]
]
```

5. Subtract the allocated value from `rowSum[0]` and `colSum[0]`:
   - New `rowSum = [0, 7]`
   - New `colSum = [3, 4]`
6. 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.
   - `rowSum` and `colSum` remain the same since the allocated number is 0.

```
ans = [
  [5, 0],
  [0, 0]
]
```

7. For `ans[1][0]`, we take the minimum of `rowSum[1]` and `colSum[0]`, which is `min(7, 3)` equals 3.
   - `ans[1][0]` is set to 3.
   - Update `rowSum` and `colSum`:
   - New `rowSum = [0, 4]`
   - New `colSum = [0, 4]`

```
ans = [
  [5, 0],
  [3, 0]
]
```

8. Finally for `ans[1][1]`, the minimum of `rowSum[1]` and `colSum[1]` is `min(4, 4)` equals 4.
   - `ans[1][1]` is set to 4.
   - Update `rowSum` and `colSum`:
   - New `rowSum = [0, 0]`
   - New `colSum = [0, 0]`

```
ans = [
  [5, 0],
  [3, 4]
]
```

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

```
ans = [
  [5, 0],
  [3, 4]
]
```

And thus we have reconstructed any 2D matrix using the solution approach, successfully meeting the given `rowSum` and `colSum` requirements.

Python Solution

```python
class Solution:
    def restoreMatrix(self, rowSums: List[int], colSums: List[int]) -> List[List[int]]:
        # Get the number of rows and columns from the given sums
        num_rows, num_cols = len(rowSums), len(colSums)

        # Initialize a matrix filled with zeros with dimensions (num_rows x num_cols)
        matrix = [[0] * num_cols for _ in range(num_rows)]

        # 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][j] by taking the minimum between
                # the current row sum and column sum, to satisfy both constraints
                value = min(rowSums[i], colSums[j])
                matrix[i][j] = value

                # Subtract the chosen value from the respective row sum and column sum
                rowSums[i] -= value
                colSums[j] -= value

        # Return the restored matrix that satisfies the row and column sums
        return matrix
```

Java Solution

```java
class Solution {

    // Restores the matrix given the sum of every row and every column.
    public int[][] restoreMatrix(int[] rowSum, int[] colSum) {
        // Get the number of rows and columns from the size of the passed arrays
        int rowCount = rowSum.length;
        int colCount = colSum.length;

        // 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;

                // After setting the value for restoredMatrix[row][col], subtract it from
                // the corresponding rowSum and colSum to maintain the sum constraints
                rowSum[row] -= value;
                colSum[col] -= value;
            }
        }
        // Return the restored matrix that matches the given row and column sums
        return restoredMatrix;
    }
}
```

C++ Solution

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

class Solution {
public:
    // 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.
    std::vector<std::vector<int>> restoreMatrix(std::vector<int>& row_sum, std::vector<int>& col_sum) {
        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 j = 0; j < cols; ++j) {
                // Find the minimum value between the current row sum and column sum.
                int value = std::min(row_sum[i], col_sum[j]);
                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 Solution

```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++) {
        for (let j = 0; j < numCols; j++) {
            // Determine the minimum value between the current row sum and column sum
            const minSum = Math.min(rowSums[i], colSums[j]);

            // Assign that minimum value to the current matrix cell
            matrix[i][j] = minSum;

            // Update the remaining sum for the current row and column
            rowSums[i] -= minSum;
            colSums[j] -= minSum;
        }
    }

    // Return the constructed matrix
    return matrix;
}
```

Time and Space Complexity

Time Complexity

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