566. Reshape the Matrix

Easy | Array | Matrix | Simulation — Leetcode Link

Problem Description

In this problem, you're required to reshape a given 2D array into a new 2D array with different dimensions. MATLAB has a similar function called `reshape`, which inspires this problem. The original matrix, named `mat`, is given along with two integers, `r` and `c`, which denote the desired number of rows and columns of the new matrix.

The new matrix is to be filled with all the elements of `mat`, following the row-wise order found in `mat`. In other words, the elements should be read from the original matrix row by row, and filled into the new matrix also row by row.

The reshaping operation should only be performed if it's possible. This means that the new matrix must be able to contain all elements of the original matrix, which implies that the number of elements in both the original and reshaped matrix must be the same (`m * n == r * c`). If the operation is not possible, the original matrix should be returned unchanged.

Intuition

The intuition behind the solution lies in understanding that a matrix is essentially a collection of elements arranged in a grid pattern, but it can also be represented as a single-line sequence of elements. The challenge is to map the index from this single-line sequence back into the indexes of a 2D array representation.

Since we are to fill the new matrix row by row, we can iterate through all the elements of the original matrix in a single loop that treats the matrix as if it were a flat array. We can calculate the original element positions using the length of the rows of the original matrix (`n`) and the position in the virtual single-line sequence (`i`).

For each element, we find the corresponding position in the reshaped matrix using `i // c` for the row (by integer division) and `i % c` for the column (using the modulus operation). These calculated positions correspond to the row and column in the reshaped matrix respectively. We directly assign the element from the original matrix to the new position in the reshaped matrix.

The solution approach uses this understanding of index mapping from a 1D representation to a 2D representation in a different shape, taking care to check whether the reshaping operation is valid before proceeding.

Solution Approach

The solution provided uses a `for` loop to iterate through all the elements of the original matrix in a sequential manner as if it were a 1D array. Here's a step-by-step explanation of the implementation:

1. **Dimension Check**: Before reshaping, we must ensure that the operation is feasible. It checks if the product of the dimensions of the original matrix (`m * n`) is equal to the product of the dimensions of the reshaped matrix (`r * c`). If not, it immediately returns the original matrix, as reshaping isn't possible.

2. **Initialization**: Assuming the reshape operation is valid, a new matrix `ans` is created with `r` rows and `c` columns, initialized to 0. This is the matrix which will hold the reshaped elements.

3. **Reshaping**: To fill the new matrix, the algorithm runs a `for` loop from `0` to `m * n - 1`, iterating over each element of `mat`.
   - It calculates the corresponding row index in `ans` by dividing the current index `i` by the number of columns `c` in the reshaped matrix (`i // c`), since `c` elements will fill one row of `ans` before moving to the next row.
   - To calculate the column index in `ans`, it uses the modulus of `i` with `c` (`i % c`). This number gives the exact position within a row where the current element should go.
   - Simultaneously, the row and column indices for the current element in the original matrix `mat` are calculated by dividing `i` by the number of columns in the original matrix (`n`) for the row, and getting the modulus of `i` with `n` for the column (`i % n`).
   - The value at the calculated row and column in the original matrix is then assigned to the corresponding position in the reshaped matrix.

By following these steps, we can reshape the original matrix into a new matrix with the desired dimensions. This approach uses simple mathematical calculations to map a linear iteration into a 2D matrix context, which is a common pattern when dealing with array transformation problems.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following 2D array `mat` and we want to reshape it into a 2D array with `r = 2` rows and `c = 4` columns.

Original matrix `mat`:

```
1 2
3 4
5 6
```

The size of the original matrix is 3 rows by 2 columns (m = 3, n = 2), so it has a total of 3 * 2 = 6 elements. Our target reshaped matrix is supposed to have 2 rows and 4 columns (r = 2, c = 4), and thus it also needs to have r * c = 2 * 4 = 8 elements. Since r * c does not equal m * n in this case (8 ≠ 6), this indicates that a reshape is not possible. The correct output should therefore be the original matrix:

```
1 2
3 4
5 6
```

Now, let's assume we desire a new shape with `r = 2` rows and `c = 3` columns. For this case, r * c = m * n (2 * 3 = 3 * 2), which means reshaping is possible. We want to convert it into the following 2D array:

1. **Dimension Check**:
   - The check confirms that since 3 * 2 (original matrix) equals 2 * 3 (new matrix), we can proceed with the reshape.
2. **Initialization**:
   - We initialize the new matrix `ans` with 2 rows and 3 columns:
     ```
     ans =
     0 0 0
     0 0 0
     ```
3. **Reshaping**:
   - We start iterating through each element in the original matrix with a single loop running from i = 0 to i = 5 (since there are 6 elements).
   - For i = 0, we read the first value of the original matrix, which is 1.
     - The row index in `ans` is computed as i // c = 0 // 3 = 0.
     - The column index in `ans` is computed as i % c = 0 % 3 = 0.
     - We place the value 1 in `ans` at (0, 0).
   - For i = 1, we read the second value of the original matrix, which is 2.
     - The row index in `ans` is computed as i // c = 1 // 3 = 0.
     - The column index in `ans` is computed as i % c = 1 % 3 = 1.
     - We place the value 2 in `ans` at (0, 1).
   - We continue this process for the remaining elements:
     ```
     When i = 2, mat[0][2] isn't valid as `n` is 2. We get mat value using mat[i // n][i % n], which is mat[2 // 2][2 % 2] = mat[
     When i = 3, mat[1][1] = 4 is placed in ans at (1, 0).
     When i = 4, mat[2][0] = 5 is placed in ans at (1, 1).
     When i = 5, mat[2][1] = 6 is placed in ans at (1, 2).
     ```

The fill process ends with the reshaped matrix `ans` looking like this:

```
1 2 3
4 5 6
```

By mapping each element's index from the original matrix to the new reshaped matrix, we have achieved the desired 2 rows by 3 columns structure using the row-wise order found in `mat`.

Python Solution

```python
from typing import List

class Solution:
    def matrixReshape(self, mat: List[List[int]], rows: int, cols: int) -> List[List[int]]:
        # Get the dimensions of the original matrix
        original_rows, original_cols = len(mat), len(mat[0])

        # If the original matrix can't be reshaped into the desired dimensions, return the original matrix
        if original_rows * original_cols != rows * cols:
            return mat

        # Initialize the reshaped matrix with zeroes
        reshaped_matrix = [[0] * cols for _ in range(rows)]

        # Iterate through the number of elements in the matrix
        for i in range(original_rows * original_cols):
            # Compute the new row and column indices for the reshaped matrix
            new_row = i // cols
            new_col = i % cols

            # Compute the old row and column indices from the original matrix
            old_row = i // original_cols
            old_col = i % original_cols

            # Assign the corresponding element from the original matrix to the reshaped matrix
            reshaped_matrix[new_row][new_col] = mat[old_row][old_col]

        # Return the reshaped matrix
        return reshaped_matrix

```

Java Solution

```java
class Solution {
    // Method to reshape a matrix to the desired number of rows (r) and columns (c)
    public int[][] matrixReshape(int[][] mat, int r, int c) {
        // Get the number of rows (m) and columns (n) of the input matrix
        int m = mat.length, n = mat[0].length;

        // If the total number of elements in the input and output matrices don't match,
        // return the original matrix
        if (m * n != r * c) {
            return mat;
        }

        // Initialize the reshaped matrix with the desired number of rows and columns
        int[][] reshapedMatrix = new int[r][c];

        // Loop through each element of the input matrix in row-major order
        for (int i = 0; i < m * n; ++i) {
            // Calculate the new row index for the current element in the reshaped matrix
            int newRow = i / c;
            // Calculate the new column index for the current element in the reshaped matrix
            int newCol = i % c;

            // Calculate the original row index for the current element in the input matrix
            int originalRow = i / n;
            // Calculate the original column index for the current element in the input matrix
            int originalCol = i % n;

            // Assign the element from the original position to the new position
            reshapedMatrix[newRow][newCol] = mat[originalRow][originalCol];
        }

        // Return the reshaped matrix
        return reshapedMatrix;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to reshape a matrix to a new size of r x c
    vector<vector<int>> matrixReshape(vector<vector<int>>& matrix, int r, int c) {
        int originalRows = matrix.size(); // Number of rows in the original matrix
        int originalCols = matrix[0].size(); // Number of columns in the original matrix

        // If the total number of elements is not the same, return the original matrix
        if (originalRows * originalCols != r * c) {
            return matrix;
        }

        vector<vector<int>> reshapedMatrix(r, vector<int>(c)); // Create a new matrix to hold the reshaped output

        // Loop through each element of the original matrix
        for (int i = 0; i < originalRows * originalCols; ++i) {
            // Calculate the new row index in the reshaped matrix
            int newRow = i / c;
            // Calculate the new column index in the reshaped matrix
            int newCol = i % c;
            // Calculate the corresponding row index in the original matrix
            int originalRow = i / originalCols;
            // Calculate the corresponding column index in the original matrix
            int originalCol = i % originalCols;
            // Place the element from the original matrix to the correct position in the reshaped matrix
            reshapedMatrix[newRow][newCol] = matrix[originalRow][originalCol];
        }

        return reshapedMatrix; // Return the reshaped matrix
    }
};

```

Typescript Solution

```typescript
// Function to reshape a matrix into a new one with 'r' rows and 'c' columns.
// If the total number of elements does not match the new dimensions, return the original matrix.
function matrixReshape(mat: number[][], r: number, c: number): number[][] {
    // Get the dimensions of the original matrix
    let originalRowCount = mat.length;
    let originalColumnCount = mat[0].length;

    // Check if reshape is possible by comparing number of elements
    if (originalRowCount * originalColumnCount != r * c) return mat;

    // Create the new matrix with 'r' rows and 'c' columns, initialized with zeroes
    let reshapedMatrix = Array.from({ length: r }, () => new Array(c).fill(0));

    // 'flatIndex' tracks the current index in the flattened original matrix
    let flatIndex = 0;

    // Iterate through the original matrix and copy values into the new reshaped matrix
    for (let i = 0; i < originalRowCount; ++i) {
        for (let j = 0; j < originalColumnCount; ++j) {
            // Calculate the new row and column indices in the reshaped matrix
            let newRow = Math.floor(flatIndex / c);
            let newColumn = flatIndex % c;

            // Assign the value from original matrix to the reshaped matrix
            reshapedMatrix[newRow][newColumn] = mat[i][j];

            // Increment the flat index as we move through the elements
            ++flatIndex;
        }
    }

    // Return the reshaped matrix
    return reshapedMatrix;
}

```

Time and Space Complexity

The time complexity of the code is `O(m * n)`, where `m` is the number of rows and `n` is the number of columns in the input matrix `mat`. This is because the code iterates over all elements in the input matrix exactly once to fill the reshaped matrix.

The space complexity of the code is `O(r * c)`, which is required for the output matrix `ans`. Although the input and output matrices have the same number of elements, the output matrix is a new structure in memory with `r` rows and `c` columns, therefore, the space complexity reflects this new structure we're creating.