566. Reshape the Matrix

April 29, 2017 | 32.9K views

Previous Next

In MATLAB, there is a very useful function called 'reshape', which can reshape a matrix into a new one with different size but keep its original data.

You're given a matrix represented by a two-dimensional array, and two **positive** integers *r* and *c* representing the **row** number and **column** number of the wanted reshaped matrix, respectively.

The reshaped matrix need to be filled with all the elements of the original matrix in the same **row-traversing** order as they were.

If the 'reshape' operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

Example 1:

```
Input:
nums =
[[1,2],
 [3,4]]
r = 1, c = 4
Output:
[[1,2,3,4]]
Explanation:
The row-traversing of nums is [1,2,3,4]. The new reshaped matrix is a 1 * 4 matrix, fi
```

Example 2:

```
Input:
nums =
[[1,2],
 [3,4]]
r = 2, c = 4
Output:
[[1,2],
 [3,4]]
Explanation:
There is no way to reshape a 2 * 2 matrix to a 2 * 4 matrix. So output the original ma
```

Note:

1. The height and width of the given matrix is in range [1, 100].
2. The given *r* and *c* are all positive.

Solution

Approach 1: Using Queue

Algorithm

The simplest method is to extract all the elements of the given matrix by reading the elements in a row-wise fashion. In this implementation, we use a queue to put the extracted elements. Then, we can take out the elements of the queue formed in a serial order and arrange the elements in the resultant required matrix in a row-by-row order again.

The formation of the resultant matrix won't be possible if the number of elements in the original matrix isn't equal to the number of elements in the resultant matrix.

Java

Copy

```
class Solution {
    public int[][] matrixReshape(int[][] nums, int r, int c) {
        int[][] res = new int[r][c];
        if (nums.length == 0 || r * c != nums.length * nums[0].length)
            return nums;
        Queue<Integer> queue = new LinkedList();
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < nums[0].length; j++) {
                queue.add(nums[i][j]);
            }
        }
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                res[i][j] = queue.remove();
            }
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(m * n)$. We traverse over $m * n$ elements twice. Here, m and n refer to the number of rows and columns of the given matrix respectively.
- Space complexity : $O(m * n)$. The queue formed will be of size $m * n$.

Approach 2: Without Using Extra Space

Algorithm

Instead of unnecessarily using the queue as in the brute force approach, we can keep putting the numbers in the resultant matrix directly while iterating over the given matrix in a row-by-row order. While putting the numbers in the resultant array, we fix a particular row and keep on incrementing the column numbers only till we reach the end of the required columns indicated by c. At this moment, we update the row index by incrementing it and reset the column index to start from 0 again. Thus, we can save the space consumed by the queue for storing the data that just needs to be copied into a new array.

Java

Copy

```
public class Solution {
    public int[][] matrixReshape(int[][] nums, int r, int c) {
        int[][] res = new int[r][c];
        if (nums.length == 0 || r * c != nums.length * nums[0].length)
            return nums;
        int rows = 0, cols = 0;
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < nums[0].length; j++) {
                res[rows][cols] = nums[i][j];
                cols++;
                if (cols == c) {
                    rows++;
                    cols = 0;
                }
            }
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(m * n)$. We traverse the entire matrix of size $m * n$ once only. Here, m and n refers to the number of rows and columns in the given matrix.
- Space complexity : $O(m * n)$. The resultant matrix of size $m * n$ is used.

Approach 3: Using division and modulus

Algorithm

In the last approach, we needed to keep a track of when we reached the end of columns for the resultant matrix and needed to update the current row and column number for putting the extracted elements by checking the current indices every time. Instead of doing these limit checks at every step, we can make use of maths to help ease the situation.

The basic behind this approach is as follows. Do you know how a 2-D array is stored in the main memory(which is 1-D in nature)? It is internally represented as a 1-D array only. The element $nums[i][j]$ of $nums$ array is represented in the form of a one dimensional array by using the index in the form: $nums[n * i + j]$, where m is the number of columns in the given matrix. Looking at the same in the reverse order, while putting the elements in the elements in the resultant matrix, we can make use of a $count$ variable which gets incremented for every element traversed as if we are putting the elements in a 1-D resultant array. But, to convert the $count$ back into 2-D matrix indices with a column count of c, we can obtain the indices as $res[count/c][count\%c]$ where $count/c$ is the row number and $count\%c$ is the column number. Thus, we can save the extra checking required at each step.

Java

Copy

```
public class Solution {
    public int[][] matrixReshape(int[][] nums, int r, int c) {
        int[][] res = new int[r][c];
        if (nums.length == 0 || r * c != nums.length * nums[0].length)
            return nums;
        int count = 0;
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < nums[0].length; j++) {
                res[count / c][count % c] = nums[i][j];
                count++;
            }
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(m * n)$. We traverse the entire matrix of size $m * n$ once only. Here, m and n refers to the number of rows and columns in the given matrix.
- Space complexity : $O(m * n)$. The resultant matrix of size $m * n$ is used.

Rate this article:

Previous Next