**566. Reshape the Matrix**

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.

我的答案：

|  |
| --- |
| class Solution {  public:  vector<vector<int>> matrixReshape(vector<vector<int>>& nums, int r, int c) {  if(nums.size() \* nums[0].size() != r \* c)  return nums;  if(r == nums.size())  return nums;    int n = nums[0].size();  vector<vector<int>> B(r);    for(int i = 0; i < r; i++)  for(int j = 0; j < c; j++)  if(n != 1)  B[i].push\_back(nums[(i\*c+j)/n][(i\*c+j)%n]);  else  B[i].push\_back(nums[i\*c+j][0]);  return B;  }  }; |

答案思路：

#### Approach #1 Using queue [Accepted]

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.

#### Approach #2 Without using extra Space [Accepted]

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*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.

#### Approach #3 Using division and modulus [Accepted]

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 idea 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 array is represented in the form of a one dimensional array by using the index in the form: nums[n\*i + j], where m*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 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*c*, we can obtain the indices as res[count/c][count\%c] where count/c is the row number and *count*%*c* is the coloumn number. Thus, we can save the extra checking required at each step.