# 2661. First Completely Painted Row or Column

```
Medium
        Array
                Hash Table
                            Matrix
```

### **Problem Description**

the sequence provided in arr. Both arr and mat contain all integers in the range [1, m \* n]. The process involves iterating over arr and painting the cell in mat that corresponds to each integer arr[i]. The goal is to determine the smallest index i at which either a whole row or a whole column in the matrix mat is completely painted. In other words, find the earliest point at which there's either a full row or a full column with all cells painted in the matrix.

Given a 0-indexed integer array arr and an m x n integer matrix mat, the task is to process arr and paint the cells in mat based on

ntuition

The solution revolves around tracking the painting progress in mat as we process each element in arr. The core thought is that we don't need to modify the matrix mat itself but can instead track the number of painted cells in each row and column with auxiliary data structures. This leads us to the idea of using two arrays row and col to maintain the count of painted cells for each respective row and column.

each number in the matrix to its coordinate (i, j). This allows us to quickly access the correct row and column to increment our counts. Solution Approach

Since we need to find the positions in the matrix mat that correspond to the integers in arr, we can set up a hash table idx to map

### Firstly, we create a hash table named idx to store the position of each element in the matrix mat. For each element in mat, we make an entry in idx where the key is the element mat[i][j] and the value is a tuple (i, j) representing its row and column

```
indices.
```

 $idx = \{\}$ for i in range(m): for j in range(n): idx[mat[i][j]] = (i, j)

The implementation consists of several key steps that use algorithms and data structures to efficiently solve the problem.

```
This hash table allows us to have constant-time lookups for the position of any element later when we traverse arr.
```

Next, we define two arrays, row and col, with lengths corresponding to the number of rows m and the number of columns n in the matrix mat. These arrays are used to count how many cells have been painted in each row and column, respectively.

```
row = [0] * m
col = [0] * n
```

(i, j) in the matrix mat using the hash table idx.

The main part of the solution involves iterating through the array arr. For each element arr[k], we get its corresponding position

```
for k in range(len(arr)):
   i, j = idx[arr[k]]
```

We increment the counts in row[i] and col[j] since arr[k] marks the cell at (i, j) as painted.

```
row[i] += 1
col[j] += 1
```

first index where a row or column is completely painted. if row[i] == n or col[j] == m: return k

Then, we check if we have completed painting a row or a column. In other words, if either row[i] equals the number of columns n

or col[j] equals the number of rows m, we have found our solution. At that point, we can return the current index k as this is the

```
This approach efficiently determines the solution by using a hash table for fast lookups and simple arrays for counting, thus
avoiding the need to make any modifications to the matrix mat itself.
```

**Example Walkthrough** 

mat = [

[1, 2],

[3, 4] arr = [3, 1, 4, 2]

The matrix mat is of size  $2 \times 2$ , so m = 2 and n = 2. Our goal is to paint the cells in mat in the order specified by arr and find the

Let's illustrate the solution with a small example. Assume we are given the following matrix mat and array arr:

## smallest index i at which either a whole row or a whole column is completely painted.

```
Step 2: Initialize counts for rows and columns:
row = [0, 0]
```

Since col[0] is now equal to the number of rows m, we've painted an entire column. The smallest index at this point is i = 1.

rows\_count, cols\_count = len(matrix), len(matrix[0])

# Create a dictionary to map each number to its position in the matrix.

# Iterate through the sequence and update row and column counters.

row\_index, col\_index = number\_position\_index[number]

Step 1: Create a hash table idx mapping matrix values to coordinates:

 $idx = \{1: (0, 0), 2: (0, 1), 3: (1, 0), 4: (1, 1)\}$ 

```
So, the earliest index in arr at which a whole row or column is painted is 1.
```

col = [0, 0]

```
Python
```

from typing import List

# solution\_instance = Solution()

# Example usage:

class Solution {

Java

C++

public:

#include <vector>

class Solution {

#include <unordered\_map>

// according to the order given in 'arr'.

for (int k = 0; k < order.size(); ++k) {</pre>

auto [i, j] = numberToPosition[order[k]];

// Function to find the first index at which all numbers in either the

function firstCompleteIndex(arr: number[], mat: number[][]): number {

for (let index = 0; index < arr.length; ++index) {</pre>

for col\_index in range(cols\_count):

return index

# The List type needs to be imported from the typing module.

return -1

from typing import List

// Get the position of the current element from the map

// Increment completion count for the row and column

// Check if the current row or column is completed

const [row, col] = positionMap.get(arr[index])!;

// same row or the same column of the matrix have appeared in the array.

// from the hash map.

// Get the dimensions of the matrix

++rowCompletionCount[row];

++colCompletionCount[col];

const rowCount = mat.length;

const colCount = mat[0].length;

number position index = {}

for row\_index in range(rows\_count):

for col\_index in range(cols\_count):

for index, number in enumerate(sequence):

if number in number\_position\_index:

row\_counters[row\_index] += 1

# This would then allow you to use the Solution class and its method.

# result = solution\_instance.first\_complete\_index(sequence, matrix)

Solution Implementation

Step 3: Start iterating through arr:

```
class Solution:
   def first_complete_index(self, sequence: List[int], matrix: List[List[int]]) -> int:
       # Get the dimensions of the matrix.
```

• For arr[0] = 3, the position in mat is idx[3] = (1, 0). Increment row[1] and col[0]. Now row = [0, 1], col = [1, 0].

• For arr[1] = 1, the position in mat is idx[1] = (0, 0). Increment row[0] and col[0]. Now row = [1, 1], col = [2, 0].

```
number = matrix[row_index][col_index]
        number_position_index[number] = (row_index, col_index)
# Initialize counters for each row and column.
row_counters = [0] * rows_count
col counters = [0] * cols count
```

```
col_counters[col_index] += 1
                # Check if a row or a column is complete.
                if row_counters[row_index] == cols_count or col_counters[col_index] == rows_count:
                    return index
       # If no row or column has been completed, return -1 as a default case.
       return -1
# The List type needs to be imported from the typing module.
```

/\*\* \* Finds the first index in the input array where a complete row or column is found in the input matrix. \* @param arr Single-dimensional array of integers. \* @param mat Two-dimensional matrix of integers.

valueToIndexMap.put(mat[row][col], new int[]{row, col});

\* @return The earliest index at which the input array causes a row or a column in the matrix to be filled.

```
int colCount = mat[0].length;
// Mapping from the value to its coordinates in the matrix
Map<Integer, int[]> valueToIndexMap = new HashMap<>();
for (int row = 0; row < rowCount; ++row) {</pre>
```

// Dimensions of the matrix

int rowCount = mat.length;

public int firstCompleteIndex(int[] arr, int[][] mat) {

```
// Arrays to keep track of the number of values found per row and column
int[] rowCompletion = new int[rowCount];
int[] colCompletion = new int[colCount];
```

for (int col = 0; col < colCount; ++col) {</pre>

```
// Iterate through the array `arr`
for (int k = 0; ; ++k) {
   // Get the coordinates of the current array value in the matrix
    int[] coordinates = valueToIndexMap.get(arr[k]);
    int rowIndex = coordinates[0];
    int colIndex = coordinates[1];
    // Increment the counters for the row and column
```

rowCompletion[rowIndex]++; colCompletion[colIndex]++; // Check if the current row or column is completed if (rowCompletion[rowIndex] == colCount || colCompletion[colIndex] == rowCount) { // Return the current index if a row or column is complete return k;

```
int firstCompleteIndex(vector<int>& order, vector<vector<int>>& matrix) {
    int rows = matrix.size(), cols = matrix[0].size();
   unordered_map<int, pair<int, int>> numberToPosition;
   // Populate a hash map with the number as the key and its position (i, j)
   // in the matrix as the value.
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
           numberToPosition[matrix[i][j]] = {i, j};
   // Create a vector to keep track of the count of filled numbers in each row and column.
   vector<int> rowCount(rows, 0), colCount(cols, 0);
```

// Returns the first index at which all numbers in a row or column are filled

// Iterate through the order vector to simulate filling the matrix.

// Get the position of the current number in the order array

```
++rowCount[i];
            ++colCount[j];
           // If a row or a column is completely filled, return the current index.
            if (rowCount[i] == cols || colCount[j] == rows) {
                return k;
       // The code prior guarantees a result, so this return statement might never be reached.
       // However, it is here as a fail-safe.
       return -1;
};
TypeScript |
```

// Increment the filled numbers count for the respective row and column.

```
// Map to store the position for each value in the matrix
const positionMap: Map<number, number[]> = new Map();
// Fill the map with positions of each value
for (let row = 0; row < rowCount; ++row) {</pre>
    for (let col = 0; col < colCount; ++col) {</pre>
        positionMap.set(mat[row][col], [row, col]);
// Arrays to keep track of the count of numbers found in each row and column
const rowCompletionCount: number[] = new Array(rowCount).fill(0);
const colCompletionCount: number[] = new Array(colCount).fill(0);
// Iterate through the array elements to find the complete row/col index
```

```
if (rowCompletionCount[row] === colCount || colCompletionCount[col] === rowCount) {
              // Return the current index if a complete row or column is found
              return index;
      // In case no complete row or column is found,
      // the function will keep running indefinitely due to the lack of a stopping condition in the for loop.
class Solution:
   def first_complete_index(self, sequence: List[int], matrix: List[List[int]]) -> int:
       # Get the dimensions of the matrix.
        rows_count, cols_count = len(matrix), len(matrix[0])
       # Create a dictionary to map each number to its position in the matrix.
       number_position_index = {}
        for row_index in range(rows_count):
```

if row\_counters[row\_index] == cols\_count or col\_counters[col\_index] == rows\_count:

```
number = matrix[row index][col index]
        number_position_index[number] = (row_index, col_index)
# Initialize counters for each row and column.
row_counters = [0] * rows_count
col_counters = [0] * cols_count
# Iterate through the sequence and update row and column counters.
for index, number in enumerate(sequence):
    if number in number_position_index:
        row_index, col_index = number_position_index[number]
        row counters[row index] += 1
        col counters[col index] += 1
```

```
# This would then allow you to use the Solution class and its method.
# Example usage:
# solution_instance = Solution()
# result = solution_instance.first_complete_index(sequence, matrix)
```

# Check if a row or a column is complete.

# If no row or column has been completed, return -1 as a default case.

```
Time and Space Complexity
```

elements and updating the row and col counts takes O(k) time. However, each element's index is accessed in constant time due to the dictionary. Therefore, the overall time complexity is the sum of both parts, 0(m \* n + k). The space complexity involves the storage used by the idx dictionary, which holds one entry for each element in the m x n matrix, hence 0(m \* n) space. Additionally, the row and col arrays utilize 0(m) and 0(n) space, respectively. Consequently, the

The time complexity of the given code can be broken down into two main parts. Firstly, populating the idx dictionary requires

iterating through each element of the m  $\times$  n matrix, which takes  $0(m \times n)$  time. Secondly, iterating over the arr array with k

total space complexity is 0(m \* n) since 0(m + n) is subsumed under 0(m \* n) when m and n are of the same order.