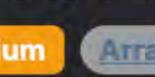
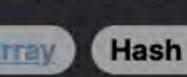
# 2661. First Completely Painted Row or Column







**Problem Description** 

Medium Array Hash Table Matrix Leetcode Link

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 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.

# Intuition

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. 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

The solution revolves around tracking the painting progress in mat as we process each element in arr. The core thought is that we

counts. With this setup, we iterate through arr, using the hashed mapping to find where arr[k] should be painted in mat. Each time we increment the counts in row and col, we check if either of them has reached the size of their respective dimension (n for rows, m for

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

columns). If they have, it means that a complete row or column has been painted, and we can return the current index k as the result. This approach leads to an efficient solution since it avoids the need to modify the matrix directly and leverages fast lookups and updates in the auxiliary arrays. Solution Approach

### 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.

for i in range(m):

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

for j in range(n):
 idx[mat[i][j]] = (i, j)

```
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.
```

1 row = [0] \* m2 col = [0] \* n

The main part of the solution involves iterating through the array arr. For each element arr[k], we get its corresponding position (i, j) in the matrix mat using the hash table idx.

```
1 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. 1 row[i] += 12 col[j] += 1

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 first

```
Then, we check if we have completed painting a row or a column. In other words, if either row[1] equals the number of columns n or
```

return k

if row[i] == n or col[j] == m:

index where a row or column is completely painted.

the need to make any modifications to the matrix mat itself. Example Walkthrough

This approach efficiently determines the solution by using a hash table for fast lookups and simple arrays for counting, thus avoiding

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

[1, 2], [3, 4]

1 mat = [

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

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

```
Step 1: Create a hash table idx mapping matrix values to coordinates:
1 idx = \{1: (0, 0), 2: (0, 1), 3: (1, 0), 4: (1, 1)\}
```

1 row = [0, 0]2 col = [0, 0]

• 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].

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.

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

```
• 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].
```

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

for index, number in enumerate(sequence):

if number in number\_position\_index:

row\_counters[row\_index] += 1

col\_counters[col\_index] += 1

col\_counters = [0] \* cols\_count

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

Step 2: Initialize counts for rows and columns:

```
Python Solution
```

class Solution:

9

10

15

16

17

18

19

20

21

22

18

19

20

23

24

25

26

29

30

31

32

33

35

36

37

38

39

41

42

43

45

26

28

29

31

32

33

34

35

36

37

42

44

43 };

44 }

Step 3: Start iterating through arr:

```
# 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):
```

def first\_complete\_index(self, sequence: List[int], matrix: List[List[int]]) -> int:

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

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

```
number = matrix[row_index][col_index]
                    number_position_index[number] = (row_index, col_index)
11
12
13
           # Initialize counters for each row and column.
           row_counters = [0] * rows_count
14
```

```
23
24
                   # Check if a row or a column is complete.
                   if row_counters[row_index] == cols_count or col_counters[col_index] == rows_count:
26
                        return index
27
28
           # If no row or column has been completed, return -1 as a default case.
29
           return -1
30
31 # The List type needs to be imported from the typing module.
   from typing import List
33
34 # This would then allow you to use the Solution class and its method.
35 # Example usage:
36 # solution_instance = Solution()
37 # result = solution_instance.first_complete_index(sequence, matrix)
Java Solution
   class Solution {
       /**
        * 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.
        * @return The earliest index at which the input array causes a row or a column in the matrix to be filled.
        */
 8
       public int firstCompleteIndex(int[] arr, int[][] mat) {
 9
           // Dimensions of the matrix
10
           int rowCount = mat.length;
11
           int colCount = mat[0].length;
12
13
           // Mapping from the value to its coordinates in the matrix
14
           Map<Integer, int[]> valueToIndexMap = new HashMap<>();
           for (int row = 0; row < rowCount; ++row) {</pre>
16
                for (int col = 0; col < colCount; ++col) {</pre>
17
```

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

// Arrays to keep track of the number of values found per row and column

// Get the coordinates of the current array value in the matrix

// Return the current index if a row or column is complete

if (rowCompletion[rowIndex] == colCount || colCompletion[colIndex] == rowCount) {

int[] coordinates = valueToIndexMap.get(arr[k]);

// Increment the counters for the row and column

// Check if the current row or column is completed

int[] rowCompletion = new int[rowCount];

int[] colCompletion = new int[colCount];

int rowIndex = coordinates[0];

int colIndex = coordinates[1];

rowCompletion[rowIndex]++;

colCompletion[colIndex]++;

// Iterate through the array `arr`

for (int k = 0; ; ++k) {

return k;

// from the hash map.

++rowCount[i];

++colCount[j];

return k;

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

if (rowCount[i] == cols || colCount[j] == rows) {

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

// If a row or a column is completely filled, return the current index.

```
C++ Solution
 1 #include <vector>
2 #include <unordered_map>
   class Solution {
   public:
       // Returns the first index at which all numbers in a row or column are filled
       // according to the order given in 'arr'.
       int firstCompleteIndex(vector<int>& order, vector<vector<int>>& matrix) {
           int rows = matrix.size(), cols = matrix[0].size();
           unordered_map<int, pair<int, int>> numberToPosition;
10
11
12
           // Populate a hash map with the number as the key and its position (i, j)
           // in the matrix as the value.
13
           for (int i = 0; i < rows; ++i) {
14
15
               for (int j = 0; j < cols; ++j) {
                   numberToPosition[matrix[i][j]] = {i, j};
16
17
18
19
20
           // Create a vector to keep track of the count of filled numbers in each row and column.
21
           vector<int> rowCount(rows, 0), colCount(cols, 0);
~~
           // Iterate through the order vector to simulate filling the matrix.
           for (int k = 0; k < order.size(); ++k) {</pre>
24
25
               // Get the position of the current number in the order array
```

```
38
           // The code prior guarantees a result, so this return statement might never be reached.
39
           // However, it is here as a fail-safe.
           return -1;
```

```
Typescript Solution
  // Function to find the first index at which all numbers in either the
2 // same row or the same column of the matrix have appeared in the array.
   function firstCompleteIndex(arr: number[], mat: number[][]): number {
       // Get the dimensions of the matrix
       const rowCount = mat.length;
       const colCount = mat[0].length;
       // 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
10
       for (let row = 0; row < rowCount; ++row) {
           for (let col = 0; col < colCount; ++col) {
               positionMap.set(mat[row][col], [row, col]);
14
15
16
       // Arrays to keep track of the count of numbers found in each row and column
17
       const rowCompletionCount: number[] = new Array(rowCount).fill(0);
18
       const colCompletionCount: number[] = new Array(colCount).fill(0);
19
20
       // Iterate through the array elements to find the complete row/col index
21
22
       for (let index = 0; index < arr.length; ++index) {</pre>
23
           // Get the position of the current element from the map
           const [row, col] = positionMap.get(arr[index])!;
24
           // Increment completion count for the row and column
25
           ++rowCompletionCount[row];
26
           ++colCompletionCount[col];
28
29
           // Check if the current row or column is completed
           if (rowCompletionCount[row] === colCount || colCompletionCount[col] === rowCount) {
30
               // Return the current index if a complete row or column is found
31
               return index;
33
34
35
       // In case no complete row or column is found,
36
37
       // the function will keep running indefinitely due to the lack of a stopping condition in the for loop.
38 }
39
```

Time and Space Complexity

and updating the row and col counts takes 0(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 total space

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 elements

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

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