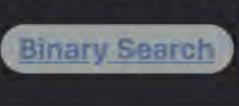
## 1198. Find Smallest Common Element in All Rows







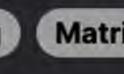


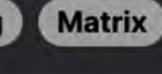






In this problem, we are given a matrix mat with dimensions m x n, where m represents the number of rows and n represents the





Leetcode Link

#### Problem Description

number of columns. Each row in the matrix is sorted in strictly increasing order. We are tasked with finding the smallest element that is common to all rows in the matrix. The question specifies that if no such common element exists, we should return -1.

every row, indicating that it is a score that every test taker has achieved.

To visualize, imagine a matrix where each row represents sorted test scores, and we want to find the lowest score that appears in

#### When approaching this problem, the key observation we can make is based on two factors:

Intuition

Each row is sorted in strictly increasing order.

- Given that each row is sorted, it might be tempting to look for the intersection of all rows, but with a large matrix, this can be

2. We want to find the smallest common element across all rows.

track the occurrences of each element across all rows. The intuition for the solution can be broken down into the following steps:

inefficient. Instead, we leverage the fact that we want to find the smallest common element. We can employ a frequency counter to

 Count the frequency of each number using a counter (a Python dictionary in the provided solution). As soon as the count of an element equals the number of rows, meaning the element has appeared in every row, we have found

our smallest common element.

Iterate through each element x in every row in the matrix.

- Return this element.
- By checking for the condition where the count equals the number of rows, we can short-circuit as soon as we find our answer. Since the rows are sorted and we are iterating row-wise, the first common element we encounter will naturally be the smallest due to the
- sort order.

 It leverages the sorted nature of the rows to ensure the first common element we find is the smallest. It avoids unnecessary comparisons since it will return immediately once a common element is found in all rows. It does not require additional space for sorting or storing large portions of the matrix beyond the counter, which at most stores n

keys where n is the number of elements in a row.

This approach is both effective and efficient because:

- **Solution Approach**
- The solution approach for this problem utilizes a Counter from Python's collections module to track the occurrences of each element across all rows in the matrix.

## Here's an in-depth explanation of the steps involved in the given solution:

 Create an instance of Counter which will hold elements as keys and their frequencies as values. Iterate through each row of the matrix using a for loop. Within each row, iterate through each element x.

Import the Counter class from the collections module. The Counter is a subclass of the dictionary and is used for counting

- Check if the count for the current element x has reached the total number of rows len(mat). If yes, this means the current element x has appeared in each row, and by the nature of the iteration over sorted rows, x is the smallest element meeting this
- criterion. As soon as such an element x is found, return x from the function, as we have found our smallest common element.

The algorithms and data structures used in this approach include:

hashable objects (elements of the matrix in this case).

- If the loop completes and no common element is found, return -1.
- Hash Table/Counter: To keep track of the number of times an element appears across all rows. Nested Loops: To iterate over all elements in all rows.

Update the count for the current element x in the Counter. This is achieved by cnt [x] += 1.

- Early Exit/Greedy Approach: By returning the first element that matches our condition (appears in all rows), we are using a greedy approach that ensures the algorithm is efficient.
- This approach works well because:
  - efficient in terms of both time and space complexity.
- number of columns. The space complexity is O(k) where k is the range of elements since we need to count occurrences of each distinct element.

The greedy component of the algorithm (returning as soon as we find the smallest common element) ensures that the solution is

Let's walk through an example to illustrate the solution approach. Imagine we have the following matrix mat with m = 3 rows and n = 4 columns:

from collections import Counter

return -1

return -1;

C++ Solution

class Solution {

def smallestCommonElement(self, mat):

# Go through each row in the matrix.

return element

# If no common element is found, return -1.

public int smallestCommonElement(int[][] matrix) {

// Iterate through each row of the matrix

elementCount[element]++;

return element;

int[] elementCount = new int[10001];

for (int element : row) {

for (int[] row : matrix) {

// Initialize an array to hold the count of each element

// Iterate through each element in the row

// Increment the count for this element

// If no common element is found in all rows, return -1

if (elementCount[element] == matrix.length) {

element\_counter = Counter()

class Solution:

16

17

18

19

20

23

24

11

15

16

20

21

22

23

24

26

24

25

26

28

29

31

30 };

25 }

[1, 2, 3, 4],

[1, 3, 5, 7],

[1, 6, 8, 9],

Example Walkthrough

1 Counter after processing first row: {1: 1, 2: 1, 3: 1, 4: 1}

 Next, we move to the second row and update the counts: 1 Counter after processing second row: {1: 2, 2: 1, 3: 2, 4: 1, 5: 1, 7: 1}

Finally, we iterate through the third row and again update the counts:

We start iterating through the first row, updating the counts in our Counter:

1 Counter after processing third row: {1: 3, 2: 1, 3: 2, 4: 1, 5: 1, 7: 1, 6: 1, 8: 1, 9: 1}

case, number '1' - is the smallest element. Once we hit this condition, we return '1'.

We notice that number 1 now has a count of 2, but since there are 3 rows, we must continue.

Python Solution

# Initialize a counter to keep track of the occurrences of each element.

# Since we are iterating in order, the first element

// If the count for the current element equals the number of rows

// it means we have found a common element present in all rows

// Return the first smallest common element found

# that satisfies this condition is the smallest common element.

# If the count of this element is equal to the number of rows, # it means this element is present in each row. if element\_counter[element] == len(mat):

Since we are looking for the smallest common element and the rows are sorted, the first element to reach the count of 3 - in this

In the given example, the smallest element common to all rows is 1, and the algorithm will return this result without having to iterate

through all elements in the matrix, showcasing its efficiency. If no common element was found, we would return -1 instead.

- The hash table operation (checking and updating the count of each element) is on average a constant time operation (0(1)). Since the matrix is pre-sorted, we can be confident that the first element found in all rows is the smallest possible, eliminating

the need for additional comparison or search algorithms.

The solution uses a single pass through all elements with a time complexity of 0(m \* n) where m is the number of rows and n is the

- Each row is sorted in strictly increasing order. We want to find the smallest element that is common to all rows. First, we create a Counter to keep track of the counts of each element across the rows.

At this point, we see that the number 1 has been encountered in each row (count == number of rows == 3), which means it is common to all of them.

for row in mat: # Go through each element in the row. 11 for element in row: # Increase the count for this element. 13 element\_counter[element] += 1 14

```
Java Solution
```

```
1 #include <vector>
2 using namespace std;
   class Solution {
   public:
       // Function to find the smallest common element in all rows of 'mat'
       int smallestCommonElement(vector<vector<int>>& mat) {
           // Array to keep track of the count of each number across all rows.
           int count[10001] = {};
           // Iterate over each row of the matrix 'mat'.
11
12
           for (auto& row : mat) {
13
               // Iterate over each element in the row.
               for (int element : row) {
14
                   // Increment the count of the current element.
                   ++count[element];
17
                   // If the current element's count equals the number of rows in 'mat',
18
                   // it means the element is present in all rows and is the smallest
19
20
                   // common element found so far. Return it.
                   if (count[element] == mat.size()) {
22
                        return element;
23
```

// If no common element is found in all rows, return -1.

#### function smallestCommonElement(matrix: number[][]): number { // Create an array to count occurrences of each element const elementCount: number[] = new Array(10001).fill(0);

return -1;

Typescript Solution

// Iterate through each row in the matrix

elementCount[element]++;

for (const element of row) {

// Iterate through each element in the current row

// If the current element's count matches the number of rows,

// Increment the count for this element

for (const row of matrix) {

```
// it means the element is common to all rows, thus return it
13
              if (elementCount[element] === matrix.length) {
14
15
                  return element;
16
17
18
19
20
       // If no common element was found in all rows, return -1
21
       return -1;
Time and Space Complexity
Time Complexity
The given Python code iterates through all elements in the matrix mat, which has a size of m * n, with m being the number of rows
and n being the number of columns. The time complexity is therefore 0(m * n) because each element is visited exactly once. The
check for the count cnt[x] to equal len(mat) is O(1) since it's a simple dictionary access.
```

# Hence, the overall time complexity is 0(m \* n).

Space Complexity The space complexity is dominated by the Counter dictionary used to store the counts of each element. In the worst case, if all elements in the matrix are distinct, the counter would need to store n elements from each of m rows, which could lead to storing up to m \* n key-value pairs.

Thus, the space complexity is 0(m \* n) in the worst case. However, if there are common elements among the rows, the number of keys in the dictionary will be less than m \* n. Still, the worst-case space complexity remains 0(m \* n).