2133. Check if Every Row and Column Contains All Numbers









Hash Table Matrix Array **Leetcode Link**

Problem Description

every row and every column contains all the integers from 1 to n (inclusive). This essentially means that, in a valid matrix, each number in the range from 1 to n should appear exactly once in each row and exactly once in each column without any repetition or omission. The function should return true if the matrix is valid, and false otherwise.

In this problem, we are given an n x n integer matrix and we need to determine if this matrix is "valid". A matrix is considered valid if

Intuition

behind our approach:

1. It appears exactly once in each row.

To determine if the given matrix is valid, we need to verify two conditions for each number from 1 to n:

- 2. It appears exactly once in each column.
- Since we need to check all numbers against both rows and columns, we can tackle each one independently. Here's the intuition

• For checking rows, we iterate over each row and create an array seen of size n where each element is initially set to False. As we encounter a number in a row, we calculate its index in the seen array by subtracting 1. If the corresponding index in the seen

- array is already True, it means we have seen this number earlier in this row, which violates our condition, so we return False. If not, we set the corresponding index to True. • For checking columns, we repeat the same process but iterate column-wise instead of row-wise, hence we'll have separate iterations for rows and columns.
- If both row-wise and column-wise iterations do not return False, it means that the matrix satisfies the condition of having unique numbers from 1 to n in all rows and columns, and thus, we return True, confirming that the matrix is valid.
- **Solution Approach** The implementation of the solution uses a straightforward approach to verify the conditions for a valid matrix. The checkValid

function is called with the matrix as input, and it is intended to return a boolean value indicating the validity of the matrix. Here's a

step-by-step explanation of the solution approach: 1. We determine the size of the matrix n by getting the length of the matrix, which is represented as the number of rows (or

columns since it is $n \times n$). 2. The function has two nested loops for each condition, i.e., one set for checking the rows and another for checking the columns:

- We iterate over each row with index i. ■ Inside this loop, we initialize an array seen of size n with all values set to False. This array keeps track of the numbers
- we have come across in the current row.
 - We then iterate over each element of the row with index j. ■ We calculate the value v by subtracting 1 from the matrix element matrix[i][j], which effectively translates the value

For columns:

return True.

• For rows:

- to an index that corresponds to the range [0, n-1]. ■ If seen[v] is True, it means the number v+1 was previously encountered in the same row, thus we should return False as the row does not contain unique numbers.
- The process is similar to rows but we iterate over each column with index j first. ■ For each column, we initialize a fresh seen array, and then iterate over each row i.
- We use the same logic as we did for rows to mark off seen values for the column elements. 3. If we successfully exit both nested loops without returning False, it means the matrix has passed all validation checks and we

■ If seen[v] is False, we mark it True, indicating that we've encountered v+1 for the first time in the current row.

- The algorithm uses an array-based implementation and checks for the presence of numbers using a boolean array. The time complexity of this algorithm is $0(n^2)$ since we have to check every element of an $n \times n$ matrix twice, once for row-wise and once
- for column-wise uniqueness.
- Example Walkthrough Let's walk through an example to understand how the solution approach works. We will use a 3 x 3 matrix for this example to

illustrate the validation process. Consider the following matrix:

We need to determine if this matrix is valid according to the specified conditions. The conditions are that the numbers 1 to n should

[[1, 2, 3],

[3, 1, 2]]

Step 1: Initialization

First, we identify n, the size of the matrix, which is 3.

Next, we see the number 2. Subtract 1 to get the index 1. Update the seen array at index 1 to True to indicate that 2 has been

• We look at the first element of the first row, which is 1. Subtract 1 to get the index 0. We update the seen array at index 0 to True to indicate that 1 has been seen: [True, False, False].

seen: [True, True, False]. • Lastly, we look at 3. We subtract 1 to get the index 2 and update seen at index 2 to True, indicating that 3 has been seen: [True,

appear exactly once in each row and once in each column.

True, True].

Step 2: Check Rows for Uniqueness

mark each number as seen, our seen array becomes [True, True, True].

Since we do not encounter any True value during this process, this row passes the uniqueness check.

We initialize a seen array for the first row and set all its values to False [False, False, False].

- We repeat this process for the second and third rows and in both cases, we don't find any duplicates. **Step 3: Check Columns for Uniqueness**
- We would then perform a similar check column-wise. • For the first column with values [1, 2, 3], we set a seen array as [False, False, False]. As we iterate through the column and

We do the same for the second and third columns, and in both cases, we find that each number from 1 to 3 appears only once.

Once both row-wise and column-wise checks are complete without encountering any True value in seen for already encountered elements, we can conclude that this matrix is indeed valid, and the function would return True.

Following the solution approach, our final output for the provided matrix [1, 2, 3], [2, 3, 1], [3, 1, 2] would be True, signaling that the matrix satisfies the required conditions of validity.

Initialize a list to keep track of seen numbers in the current row

Initialize a list to keep track of seen numbers in the current column

// Retrieve the number and convert it to an index (0 to n-1)

// Check if the number has already been seen in the current column

// If the number is seen before, the matrix is not valid

int numberIndex = matrix[row][col] - 1;

if (seenColumn[numberIndex]) {

seenColumn[numberIndex] = true;

// Mark the number as seen

return false;

Mark the number as seen in the current row

Calculate the correct index for the value considering 1-based to 0-based indexing 12 value_index = matrix[row][col] - 1 13 14 # If we have already seen this number in the current row, the matrix is not valid if seen_row[value_index]: 15

Calculate the correct index for the value considering 1-based to 0-based indexing

If we have already seen this number in the current column, the matrix is not valid

```
class Solution:
    def checkValid(self, matrix: List[List[int]]) -> bool:
        # Find the size of the matrix
```

size = len(matrix)

for row in range(size):

Check every row in the matrix

seen_row = [False] * size

return False

Check every column in the matrix

seen_col = [False] * size

for row in range(size):

for col in range(size):

seen_row[value_index] = True

value_index = matrix[row][col] - 1

for col in range(size):

Python Solution

16

17

18

19

20

21

24

25

26

27

28

29

```
30
                    if seen_col[value_index]:
                        return False
31
32
33
                    # Mark the number as seen in the current column
34
                    seen_col[value_index] = True
35
36
           # If we reach this point, the matrix has unique numbers in all rows and all columns
37
            return True
38
Java Solution
   class Solution {
       public boolean checkValid(int[][] matrix) {
           // Retrieve the size of the matrix, which is a square matrix (n \times n)
           int size = matrix.length;
           // Check each row in the matrix
            for (int row = 0; row < size; ++row) {</pre>
                // Create a tracker for seen numbers in the current row
                boolean[] seenRow = new boolean[size];
                for (int col = 0; col < size; ++col) {</pre>
10
                    // Retrieve the number and convert it to an index (0 to n-1)
                    int numberIndex = matrix[row][col] - 1;
                    // Check if the number has already been seen in the current row
14
                    if (seenRow[numberIndex]) {
15
                        // If the number is seen before, the matrix is not valid
                        return false;
16
17
18
                    // Mark the number as seen
                    seenRow[numberIndex] = true;
19
20
21
22
23
           // Check each column in the matrix
24
           for (int col = 0; col < size; ++col) {</pre>
25
                // Create a tracker for seen numbers in the current column
                boolean[] seenColumn = new boolean[size];
26
27
                for (int row = 0; row < size; ++row) {</pre>
```

37 38 // If all rows and columns contain all numbers exactly once, matrix is valid 39 40 return true; 41

28

29

30

31

32

33

34

35

36

42 }

4

*/

```
43
C++ Solution
 1 class Solution {
 2 public:
       // Function to check if the given matrix is valid
       bool checkValid(vector<vector<int>>& matrix) {
            int size = matrix.size(); // Get the size of the matrix
           // Check each row for validity
           for (int rowIndex = 0; rowIndex < size; ++rowIndex) {</pre>
                vector<bool> seenInRow(size, false); // Initialize seenInRow to keep track of numbers in row
 9
                for (int columnIndex = 0; columnIndex < size; ++columnIndex) {</pre>
                    int value = matrix[rowIndex][columnIndex] - 1; // Zero-based index for checking
11
                    if (seenInRow[value]) return false; // If the number is already seen, matrix is invalid
12
13
                    seenInRow[value] = true; // Mark the number as seen for current row
14
15
16
17
           // Check each column for validity
           for (int columnIndex = 0; columnIndex < size; ++columnIndex) {</pre>
18
                vector<bool> seenInColumn(size, false); // Initialize seenInColumn to keep track of numbers in column
19
20
                for (int rowIndex = 0; rowIndex < size; ++rowIndex) {</pre>
21
                    int value = matrix[rowIndex][columnIndex] - 1; // Zero-based index for checking
22
                    if (seenInColumn[value]) return false; // If the number is already seen, matrix is invalid
                    seenInColumn[value] = true; // Mark the number as seen for current column
23
24
25
26
27
           // If all rows and columns pass the checks, return true indicating the matrix is valid
28
           return true;
29
30 };
31
Typescript Solution
```

```
const size = matrix.length;
10
       // Create arrays to track the presence of numbers in each row and column.
11
       let rows = Array.from({ length: size }, () => new Array(size).fill(false));
12
       let columns = Array.from({ length: size }, () => new Array(size).fill(false));
13
14
       // Iterate over each cell in the matrix.
15
       for (let i = 0; i < size; i++) {
16
17
           for (let j = 0; j < size; j++) {
               // Get the current value, subtracting 1 to convert it to a 0-based index.
18
               let currentValue = matrix[i][j] - 1;
19
20
               // Check if the current value has already been seen in the current row or column.
21
22
               if (rows[i][currentValue] || columns[j][currentValue]) {
23
                   return false; // If the value has been seen, return false as the matrix is invalid.
24
25
26
               // Mark the value as seen for the current row and column.
27
               rows[i][currentValue] = true;
28
               columns[j][currentValue] = true;
29
30
31
       // If we've checked all cells and have not returned false, the matrix is valid.
32
33
       return true;
34 }
35
Time and Space Complexity
The time complexity of the code is 0(n^2). This is because there are two nested loops, each iterating n times, where n is the size of
```

each row and column, but since it is reused, the space complexity does not exceed O(n).

* Checks if a given square matrix is valid. A matrix is valid if every row and every column

* @param {number[][]} matrix - The square matrix to check for validity.

function checkValid(matrix: number[][]): boolean {

* @returns {boolean} - Returns true if the matrix is valid, false otherwise.

* contains all numbers from 1 to n (where n is the number of rows/columns in the matrix) exactly once.

column for validating the matrix, ensuring all numbers from 1 to n appear exactly once in each row and column. The space complexity of the code is O(n). This is because additional space is used to create the seen array which has a size of n. This array is used to track whether a number has been seen before in the current row or column. The seen array is reset to False for

the matrix's dimension (both width and height). The first loop iterates through each row while the second inner loop checks each