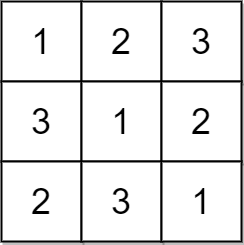
<https://leetcode.com/problems/check-if-every-row-and-column-contains-all-numbers/description/>

An n x n matrix is **valid** if every row and every column contains **all** the integers from 1 to n (**inclusive**).

Given an n x n integer matrix matrix, return true *if the matrix is****valid****.* Otherwise, return false.

**Example 1:**

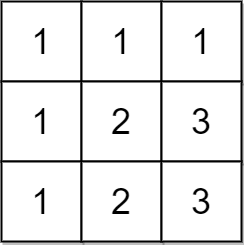


**Input:** matrix = [[1,2,3],[3,1,2],[2,3,1]]

**Output:** true

**Explanation:** In this case, n = 3, and every row and column contains the numbers 1, 2, and 3.Hence, we return true.

**Example 2:**



**Input:** matrix = [[1,1,1],[1,2,3],[1,2,3]]

**Output:** false

**Explanation:** In this case, n = 3, but the first row and the first column do not contain the numbers 2 or 3.Hence, we return false.

**Constraints:**

n == matrix.length == matrix[i].length

1 <= n <= 100

1 <= matrix[i][j] <= n

**Attempt 1: 2025-08-03**

**Solution 1: Hash Table (10 min)**

class Solution {

    public boolean checkValid(int[][] matrix) {

        int n = matrix.length;

        for(int i = 0; i < n; i++) {

            Set<Integer> row = new HashSet<>();

            Set<Integer> col = new HashSet<>();

            for(int j = 0; j < n; j++) {

                if(!row.add(matrix[i][j]) || !col.add(matrix[j][i])) {

                    return false;

                }

            }

        }

        return true;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

LeetCode 2133 - Check Valid Matrix Solution

This problem requires checking if an n x n matrix contains all integers from 1 to n in every row and every column without any duplicates.

**Approaches**

**Approach 1: HashSet Validation**

**Time Complexity**: O(n²)

**Space Complexity**: O(n)

**Idea**: Use HashSet to check for duplicates in each row and column. For each row and column, we'll verify it contains exactly the numbers 1 to n.

**Approach 2: Boolean Array Validation**

**Time Complexity**: O(n²)

**Space Complexity**: O(n)

**Idea**: Use boolean arrays instead of HashSets for slightly better performance, marking seen numbers.

**Approach 3: In-place Validation**

**Time Complexity**: O(n²)

**Space Complexity**: O(1)

**Idea**: Use the matrix itself to mark seen numbers by negating values (only works if all numbers are positive).

**Solution Code (Approach 1 - HashSet)**

class Solution {

public boolean checkValid(int[][] matrix) {

int n = matrix.length;

for (int i = 0; i < n; i++) {

Set<Integer> rowSet = new HashSet<>();

Set<Integer> colSet = new HashSet<>();

for (int j = 0; j < n; j++) {

// Check row

if (matrix[i][j] < 1 || matrix[i][j] > n || !rowSet.add(matrix[i][j])) {

return false;

}

// Check column

if (matrix[j][i] < 1 || matrix[j][i] > n || !colSet.add(matrix[j][i])) {

return false;

}

}

}

return true;

}

}

**Solution Code (Approach 2 - Boolean Array)**

class Solution {

public boolean checkValid(int[][] matrix) {

int n = matrix.length;

for (int i = 0; i < n; i++) {

boolean[] rowSeen = new boolean[n + 1];

boolean[] colSeen = new boolean[n + 1];

for (int j = 0; j < n; j++) {

int rowVal = matrix[i][j];

int colVal = matrix[j][i];

if (rowVal < 1 || rowVal > n || rowSeen[rowVal]) {

return false;

}

rowSeen[rowVal] = true;

if (colVal < 1 || colVal > n || colSeen[colVal]) {

return false;

}

colSeen[colVal] = true;

}

}

return true;

}

}

**Solution Code (Approach 3 - In-place)**

class Solution {

public boolean checkValid(int[][] matrix) {

int n = matrix.length;

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

int val = Math.abs(matrix[i][j]);

if (val < 1 || val > n) return false;

// Mark row

if (matrix[i][val - 1] < 0) return false;

matrix[i][val - 1] \*= -1;

// Mark column

if (matrix[val - 1][j] > 0) {

matrix[val - 1][j] \*= -1;

} else {

return false;

}

}

}

// Restore matrix (optional)

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

matrix[i][j] = Math.abs(matrix[i][j]);

}

}

return true;

}

}

**Explanation**

**HashSet Approach**:

Uses two HashSets per iteration to track seen numbers in rows and columns

Checks for invalid numbers (outside 1-n range) and duplicates

Efficient but uses more space due to HashSet overhead

**Boolean Array Approach**:

Uses boolean arrays instead of HashSets for better performance

Similar logic but with direct array access which is faster

Still maintains O(n) space complexity

**In-place Approach**:

Most space efficient (O(1) space)

Marks seen numbers by negating values in the matrix

Requires restoring the matrix afterward if needed

Tricky to implement correctly due to overlapping marks

The HashSet approach is the most straightforward and readable, while the boolean array approach offers better performance. The in-place solution is most space efficient but more complex to implement correctly. Choose based on your specific needs (readability vs performance vs space).

**Refer to**

[L36.Valid Sudoku (Ref.L37,L2133)](note://WEBa794bfed438a7ef4e6df229bd1d854d1)

[L37.Sudoku Solver (Ref.L36)](note://AEF375E022AF421AB858359AF9045537)

[L1572.Matrix Diagonal Sum (Ref.L2133,L2319)](note://WEB4f166767aa668b346d883f3d30593f17)

[L2661.First Completely Painted Row or Column (Ref.L2133)](note://WEB83715ac07871669cd4154fc3cf5af82d)