36. Valid Sudoku

Medium Array Hash Table Matrix

Leetcode Link

Problem Description

In this problem, we are given a partially filled `9 x 9` Sudoku board. Our task is to determine whether the board is valid according to the rules of Sudoku. We do not need to solve the Sudoku; we just need to validate the existing filled-in cells. The rules are that each filled cell must meet three constraints:

1. Every row must have the digits `1-9` without any repetition.
2. Every column must have the digits `1-9` without any repetition.
3. Each of the nine `3 x 3` subgrids or boxes that compose the grid must have the digits `1-9` without any repetition.

A key point to note is that a valid Sudoku board does not necessarily mean that it can be successfully solved to completion. The question only asks for validation of the current state of the board, not solvability.

Intuition

To solve this problem effectively, we can follow an approach that checks all three Sudoku rules for validity as we traverse the board once. The intuition here is to keep track of the constraints to prevent any repetitions of the digits in rows, columns, and sub-boxes.

How we arrive at the solution approach:

- We initialize three lists to record the presence of each digit in rows, columns, and sub-grids (`row`, `col`, `sub`). Each of these lists will have 9 elements, with each element representing a row, column, or sub-grid, and each element is a `9` elements list to keep track of digits `1-9`.
- We then iterate over each cell in the board. Whenever we encounter a filled cell (not a '.'), we check for the digit's presence in the corresponding row, column, and sub-grid.
- The sub-grid (also called a box) can be identified by using index arithmetic. We use `i // 3 * 3 + j // 3` to obtain the index of the sub-grid where `i, j` is the cell's row and column indices.
- If the digit is already present in its corresponding row, column, or sub-grid, we return `False` since it breaks the Sudoku rule of no repetition.
- If the digit passes all checks, we mark it as present in the row, column, and sub-grid.
- After checking all cells, if no rule is broken, we return `True`, confirming that the board state is valid.

Solution Approach

The solution follows a straightforward approach to validate a Sudoku board. It utilizes data storage patterns with auxiliary space, specifically lists which lists to track the digits that have been seen in the rows, columns, and sub-grids. The algorithm implemented can be broken down into the following steps:

1. **Initialization**: We set up three lists of lists:
   - `row` to keep track of which numbers have been seen in each of the 9 rows.
   - `col` to keep track of which numbers have been seen in each of the 9 columns.
   - `sub` for the numbers seen in each of the 9 sub-grids. Each `[[False] * 9 for _ in range(9)]` represents a list of 9 elements initialized to `False`, indicating that no numbers have been encountered yet.
2. **Iteration over the board**: We traverse the board with two nested loops, one iterating over the rows (`i`) and the other over the columns (`j`).
3. **Skip empty cells**: We continue to the next iteration if the current cell (identified by `board[i][j]`) contains ".", which stands for an empty cell in Sudoku.
4. **Calculations**:
   - Ascertain the digit (converted to a zero-based index by `int(board[i][j]) - 1`) present in the cell to update the tracking lists.
   - Determine the correct sub-grid using the formula `k = i // 3 * 3 + j // 3`, which works by dividing the board into 3-row and 3-column segments.
5. **Validation Checks**:
   - If the number has already been seen in the current `row[i]`, `col[j]`, or `sub[k]`, return `False` since it violates the rules of Sudoku.
6. **Updating Structures**:
   - If the number hasn't been seen, we update `row[i][num]`, `col[j][num]`, and `sub[k][num]` to `True` to indicate that the digit is now accounted for in the respective row, column, and sub-grid.
7. **Final verdict**: If no repetitions are found throughout the entire board, we finish iterating with no issues and return `True`, confirming the board's validity.

By using these data structures, the solution achieves an efficient way to keep track of which numbers have appeared in different parts of the board without having to re-scan portions of the board multiple times. This results in a time complexity of O(1) for each cell check (since accessing and updating the elements in a list by index is O(1)), and a total time complexity of O(n^2) for the entire board, where n is the number of cells in a row or column.

Example Walkthrough

Let's illustrate the solution approach with a small example of a `9x9` Sudoku board section. The section we will walk through is a typical scenario with three cells filled:

```
5 3 .
6 . .
. 9 8
```

Here, we have filled digits '5', '1', and '2'. We need to validate if this section of the board is valid without considering the other cells of the entire board.

Step 1: Initialization

We create three lists, `row`, `col`, and `sub`, each initialized to `[[False] * 9 for _ in range(9)]`. This means we have a matrix for rows, columns, and sub-grids with `False` values, indicating that no numbers have been seen yet.

Step 2: Iteration over the board

We start traversing the board. In this case, we have:

```
1  Iteration 1:
2  i = 0, j = 0 -> cell (0,0) contains '5'
3
4  Iteration 2:
5  i = 0, j = 1 -> cell (0,1) contains '3'
6
7  Iteration 3:
8  i = 0, j = 2 -> cell (0,2) contains '.'
9
10 Iteration 4:
11 i = 1, j = 0 -> cell (1,0) contains '6'
12
13 Iteration 5:
14 i = 1, j = 1 -> cell (1,1) contains '1'
15
16 Iteration 6:
17 i = 1, j = 2 -> cell (1,2) contains '.'
18
19 Iteration 7:
20 i = 2, j = 0 -> cell (2,0) contains '.'
21
22 Iteration 8:
23 i = 2, j = 1 -> cell (2,1) contains '.'
24
25 Iteration 9:
26 i = 2, j = 2 -> cell (2,2) contains '2'
```

Step 3: Skip empty cells

As we encounter empty cells, indicated by '.', we continue to the next iteration and do not perform any operations for these cells.

Step 4: Calculations

- In the first iteration (`i = 0, j = 0`), we encounter the number '5'. We translate this to a zero-based index, which gives us `4`.
- We identify the sub-grid using the formula `k = i // 3 * 3 + j // 3`. Here, `k` is `0` since both `i // 3` and `j // 3` are `0`.

Step 5: Validation Checks

- We check if the number '5' has been seen in `row[0][4]`, `col[0][4]`, or `sub[0][4]`. Since all are `False`, we proceed.

Step 6: Updating Structures

- We mark `row[0][4]`, `col[0][4]`, and `sub[0][4]` as `True`, indicating that the number '5' has been accounted for in its row, column, and sub-grid.

- Similarly, for the cell (1,1) containing '1', we mark `row[1][0]`, `col[1][0]`, and `sub[0][0]` as `True`.

- For the cell (2,2) containing '2', we mark `row[2][1]`, `col[2][1]`, and `sub[0][1]` as `True`.

Step 7: Final Verdict

- Since none of the validation checks returned `False`, we conclude that no rules of Sudoku were violated, and thus the board section is considered valid.

In this small walkthrough, we can see that by maintaining tracking lists and updating them, we can efficiently validate the Sudoku board state without re-checking any cell. Once all cells are processed in this manner, if no issues arise, we can confidently return `True`.

Python Solution

```python
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        # Initialize tracking lists for rows, columns, and 3x3 sub-boxes.
        row = [[False] * 9 for _ in range(9)]
        col = [[False] * 9 for _ in range(9)]
        boxes = [[False] * 9 for _ in range(9)]

        # Iterate over each cell in the 9x9 board.
        for i in range(9):
            for j in range(9):
                cell_value = board[i][j]
                # Skip checking if the cell is empty.
                if cell_value == '.':
                    continue

                # Convert str digit to int and adjust index to zero-based.
                num = int(cell_value) - 1

                # Calculate box index for 3x3 sub-boxes using integer division.
                box_index = (i // 3) * 3 + j // 3

                # If the number has already been encountered in current
                # row, column or box, sudoku condition is violated.
                if row[i][num] or col[j][num] or boxes[box_index][num]:
                    return False

                # Mark current num as encountered in current row, column and box.
                row[i][num] = True
                col[j][num] = True
                boxes[box_index][num] = True

        # If no conditions are violated, then the board is a valid sudoku.
        return True
```

Java Solution

```java
class Solution {
    // Function to check if a given Sudoku board is valid.
    public boolean isValidSudoku(char[][] board) {
        // Arrays to check the validity of rows, columns, and sub-grids.
        boolean[][] rows = new boolean[9][9];
        boolean[][] columns = new boolean[9][9];
        boolean[][] subgrids = new boolean[9][9];

        // Iterate over the board by rows and columns.
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                char currentChar = board[i][j];

                // If the current character is a dot, ignore and continue to the next iteration.
                if (currentChar == '.') {
                    continue;
                }

                // Convert char to its corresponding number (1-9).
                int number = currentChar - '0' - 1;

                // Calculate index for the subgrids.
                int subgridIndex = (i / 3) * 3 + j / 3;

                // Check if the number has already been recorded in the current row, column, or subgrid.
                if (rows[i][number] || columns[j][number] || subgrids[subgridIndex][number]) {
                    // If any is true, then the board is not valid.
                    return false;
                }

                // Mark the presence of the number in the current row, column, and subgrid.
                rows[i][number] = true;
                columns[j][number] = true;
                subgrids[subgridIndex][number] = true;
            }
        }

        // If no conflicts are found, the board is valid.
        return true;
    }
}
```

C++ Solution

```cpp
#include<vector>
using namespace std;
class Solution {
public:
    // Function to check if a Sudoku board is valid.
    bool isValidSudoku(std::vector<std::vector<char>>& board) {
        // Create three 9x9 matrices for rows, columns, and sub-boxes respectively.
        std::vector<std::vector<bool>> rowCheck(9, std::vector<bool>(9, false));
        std::vector<std::vector<bool>> colCheck(9, std::vector<bool>(9, false));
        std::vector<std::vector<bool>> subBoxCheck(9, std::vector<bool>(9, false));

        // Traverse the entire board to check each element.
        for (int row = 0; row < 9; ++row) {
            for (int col = 0; col < 9; ++col) {
                // Read the current character.
                char currentChar = board[row][col];

                // Skip if the cell is empty (denoted by '.').
                if (currentChar == '.') continue;

                // Convert char digit to integer index (0 to 8).
                int num = currentChar - '0' - 1;

                // Calculate sub-box index based on row and column.
                int subBoxIndex = (row / 3) * 3 + (col / 3);

                // Check if the number has already been used in the current row, column, or sub-box.
                // If it has, then the Sudoku board is invalid.
                if (rowCheck[row][num] || colCheck[col][num] || subBoxCheck[subBoxIndex][num]) {
                    return false;
                }

                // Mark the number as used in the current row, column, and sub-box.
                rowCheck[row][num] = true;
                colCheck[col][num] = true;
                subBoxCheck[subBoxIndex][num] = true;
            }
        }

        // If all checks pass, then the board is valid.
        return true;
    }
};
```

Typescript Solution

```typescript
function isValidSudoku(board: string[][]): boolean {
    // Initialize boolean arrays to track the presence of digits in rows, columns, and sub-boxes.
    const rowTracker: boolean[][] = Array.from({ length: 9 }, () => new Array<boolean>(9).fill(false));
    const colTracker: boolean[][] = Array.from({ length: 9 }, () => new Array<boolean>(9).fill(false));
    const subTracker: boolean[][] = Array.from({ length: 9 }, () => new Array<boolean>(9).fill(false));

    // Loop through each cell of the board.
    for (let i = 0; i < 9; i++) {
        for (let j = 0; j < 9; j++) {
            // Get the current value of the cell, if it's a number.
            const num = board[i][j].charCodeAt(0) - '0'.charCodeAt(0) - 1; // Adjust ASCII '1' to index 0

            // Skip the cell if it's not a valid number (i.e., '.').
            if (num < 0 || num > 8) {
                continue;
            }

            // Calculate the index for the sub-boxes.
            const subBoxIndex = Math.floor(i / 3) * 3 + Math.floor(j / 3);

            // Check whether the number has already been used in the current row, column, or sub-box.
            if (rowTracker[i][num] || colTracker[j][num] || subTracker[subBoxIndex][num]) {
                return false; // If the number has appeared, the board is not valid.
            }

            // Mark the number as seen in the current row, column, and sub-box.
            rowTracker[i][num] = true;
            colTracker[j][num] = true;
            subTracker[subBoxIndex][num] = true;
        }
    }

    // If no duplicates were found, the board is valid.
    return true;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(n^2)` where `n` is the length of one side of the Sudoku board. Since the board is always `9x9`, the complexity can be considered `O(1)` because it does not scale with input size.

The space complexity of the code is also `O(n^2)` for similar reasons. We initialize three 2D arrays (`row`, `col`, and `sub`) to keep track of the numbers present in each row, column, and 3×3 subgrid. Again, since `n = 9`, the space complexity is essentially `O(1)`.