794. Valid Tic-Tac-Toe State

Medium Array Matrix

Leetcode Link

Problem Description

In this problem, we are given the current state of a Tic-Tac-Toe board represented as an array of strings. Our goal is to determine whether this board configuration could have been reached in the course of a valid game. The board is of size `3 x 3`, and the characters `'X'`, `'O'`, and `' '` represent the moves by the first player, the second player, and an empty space, respectively.

Tic-Tac-Toe is played according to these rules:

- Two players take turns to make their move.
- The first player uses `'X'` and the second player uses `'O'`.
- Players can only place their marks in empty squares (`' '`).
- The game ends either when a player has filled a row, column, or diagonal with their character, or when all the squares are filled, rendering the game over.
- Once the game has ended, no further moves can be made.

We must verify if the given board could possibly be the result of playing a game following these rules.

Intuition

To provide a valid solution, a few simple checks must be implemented based on the game's rules.

1. Count the occurrences of `'X'` and `'O'`. Since players alternate turns starting with `'X'`, there should either be an equal number of both or one more `'X'` than `'O'`.
2. Check for winners. Since the game stops as soon as a winning condition is met:
   - If `'X'` has won, there must be exactly one more `'X'` than `'O'` on the board—because `'X'` plays first and would have just played to win the game.
   - If `'O'` has won, there must be an equal number of `'X'` and `'O'` on the board since `'O'` would have just played to win the game.
3. Understand that both `'X'` and `'O'` cannot win in a single game instance—it would against game rules.

The code given first defines a function, `win(c)`, which checks for winning conditions for the character `c`. It checks all rows, columns, and both diagonals to see if any of them are filled with the same character. The code then counts the number of `'X'`s and `'O'`s on the board. Lastly, it applies the previous two checks: it confirms the counts are appropriate given the rules, and it ensures that only one (or none) of the players could have possibly won the game. If any of these conditions fail, the board state is impossible, and the function returns `False`. Otherwise, it returns `True`.

By decomposing the problem into these logical steps, we can arrive at a clear solution approach that aligns with the rules of Tic-Tac-Toe.

Solution Approach

The solution is implemented in Python and uses simple iterative constructs to verify the validity of the given Tic-Tac-Toe board configuration.

Here are the key components of the solution approach:

1. **Counting `'X'` and `'O'` Characters**:
   - The solution starts by counting the number of `'X'` and `'O'` characters present on the board utilizing list comprehensions.
   - It's done by looping through each cell in the board using two nested loops (one for rows and one for columns) and incrementing the count appropriately for each character.
2. **Winning Check Function (`win(c)`)**:
   - The function `win(c)` takes a character `c` as an input and checks if that character has a win condition on the board.
   - It checks for a win in each row, in each column, and across both diagonals. The `all` function is used to simplify the check within each row, column, or diagonal.
   - If any win condition is met, it returns `True`. Otherwise, it returns `False`.
3. **Game Rules Logic**:
   - After counting `'X'`s and `'O'`s, the code checks if their counts are valid. According to the rules, there cannot be more `'O'`s than `'X'`s, and `'X'`s can have at most one more than `'O'` because they start the game. If these conditions are not met, the function returns `False`.
   - Then, it checks if `'X'` has won. If `'X'` has won, there must be one more `'X'` than `'O'`, as `'X'` would have just played the winning move. If there is no such count difference, the function returns `False`.
   - Similarly, it checks if `'O'` has won. If `'O'` has won, the counts of `'X'` and `'O'` must be equal, as `'O'` would have just played the winning move. If `does not equal` is in count, then it returns `False`.
4. **Final Validation**:
   - Lastly, the code ensures that both `'X'` and `'O'` have not won simultaneously. If either `'X'` or `'O'` is winning, their respective count checks must also pass, as previously described.

An understanding of how a Tic-Tac-Toe game progresses and ends is integral to the solution. The implementation leverages basic control structures and the inherent properties of the game rules to reach a conclusion. There's no use of advanced data structures since the problem can be solved with simple list iteration and condition checking.

By combining these logical steps, the code effectively validates if the final board state can be the result of a valid game of Tic-Tac-Toe.

Example Walkthrough

Let's consider a Tic-Tac-Toe board with the following configuration:

```
["X", "O", "O"],
["X", "X", "O"],
["X", " ", " "]
```

Using the solution approach, we'll walk through the validation process:

1. **Counting `'X'` and `'O'` Characters**:
   - There are 3 `'X'` characters and 3 `'O'` characters on the board.
2. **Winning Check Function (`win(c)`)**:
   - We check if `'X'` has a win condition:
     - No row or column is entirely filled with `'X'`.
     - Diagonals do not have three `'X'` either.
     - So, `win('X')` would return `False`.
   - Next, we check if `'O'` has a win condition:
     - No row or column is entirely filled with `'O'`.
     - However, one diagonal (from the bottom right to the top left) is filled with `'O'`.
     - So, `win('O')` would return `True`.
3. **Game Rules Logic**:
   - Since `'O'` has a win condition, and the count of `'X'` and `'O'` is equal, this matches the rule that `'O'` must have won on the last move.
4. **Final Validation**:
   - Since only `'O'` has a win condition and the counts of `'X'` and `'O'` follow the rule associated with `'O'` winning, the game board is a possible result of a valid Tic-Tac-Toe game.

This illustrates how the code would verify the validity of the given board configuration by applying simple checks that are tied to the game rules. The analysis confirms that this particular board could indeed represent a game of Tic-Tac-Toe that has been played correctly according to the standard rules.

Python Solution

```python
class Solution:
    def validTicTacToe(self, board: List[str]) -> bool:
        # Function to check if player with mark 'X' or 'O' has won
        def has_won(mark):
            # Check horizontal and vertical wins
            for i in range(3):
                if all(board[i][j] == mark for j in range(3)):
                    return True
                if all(board[j][i] == mark for j in range(3)):
                    return True

            # Check for the two diagonal wins
            if all(board[i][i] == mark for i in range(3)):
                return True
            return all(board[i][2 - i] == mark for i in range(3))

        # Count the number of 'X's and 'O's on the board
        count_x = sum(row.count('X') for row in board)
        count_o = sum(row.count('O') for row in board)

        # Check for the correct number of 'X's and 'O's
        if count_x != count_o and count_x - 1 != count_o:
            return False

        # If 'X' has won, the count of 'X' and 'O' must be the same
        if has_won('X') and count_x != count_o:
            return False

        # If 'O' has won, the count of 'X' must be one more than 'O'
        if has_won('O') and count_x != count_o + 1:
            return False

        # The board is valid if it does not violate any of the above rules
        return True
```

Java Solution

```java
class Solution {
    private String[] board;

    // Checks if the given Tic-Tac-Toe board state is valid
    public boolean validTicTacToe(String[] board) {
        this.board = board;
        int countX = count('X'), countO = count('O');

        // 'X' goes first so there must be either the same amount of 'X' and 'O'
        // or one more 'X' than 'O'
        if (countX != countO && countX - 1 != countO) {
            return false;
        }

        // If 'X' has won, there must be one more 'X' than 'O'
        if (hasWon('X') && countX - 1 != countO) {
            return false;
        }

        // If 'O' has won, there must be the same number of 'X' and 'O'
        return !(hasWon('O') && countX != countO);
    }

    // Checks if the given player has won
    private boolean hasWon(char player) {
        // Check all rows and columns
        for (int i = 0; i < 3; ++i) {
            if (board[i].charAt(0) == player && board[i].charAt(1) == player && board[i].charAt(2) == player) {
                return true;
            }

            if (board[0].charAt(i) == player && board[1].charAt(i) == player && board[2].charAt(i) == player) {
                return true;
            }
        }
        // Check both diagonals
        if (board[0].charAt(0) == player && board[1].charAt(1) == player && board[2].charAt(2) == player) {
            return true;
        }
        return board[0].charAt(2) == player && board[1].charAt(1) == player && board[2].charAt(0) == player;
    }

    // Counts the number of times the given character appears on the board
    private int count(char character) {
        int count = 0;
        for (String row : board) {
            for (char cell : row.toCharArray()) {
                if (cell == character) {
                    ++count;
                }
            }
        }
        return count;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to check the validity of a tic-tac-toe game board state
    bool validTicTacToe(vector<string>& board) {
        // Lambda to count occurrences of a provided character ('X' or 'O') on the board
        auto countCharacter = [&](char character) {
            int count = 0;
            for (const auto& row : board) {  // Iterate through each row of the board
                for (char cell : row) {      // Iterate through each cell in the row
                    count += (cell == character); // Increment count if the cell matches the character
                }
            }
            return count;
        };

        // Lambda to check if a given player has won the game
        auto hasWon = [&](char player) {
            // Check rows and columns for a win condition
            for (int i = 0; i < 3; ++i) {
                if (board[i][0] == player && board[i][1] == player && board[i][2] == player) return true;
                if (board[0][i] == player && board[1][i] == player && board[2][i] == player) return true;
            }
            // Check both diagonals for a win condition
            if (board[0][0] == player && board[1][1] == player && board[2][2] == player) return true;
            return board[0][2] == player && board[1][1] == player && board[2][0] == player;
        };

        // Count the number of 'X' and 'O' on the board
        int countX = countCharacter('X');
        int countO = countCharacter('O');

        // Check the game's rules:
        // Rule 1: The number of 'X's must either be equal to or one more than the number of 'O's
        if (countX != countO && countX - 1 != countO) return false;

        // Rule 2: If 'O' has won, there must be one more 'X' than 'O'
        if (hasWon('X') && countX - 1 != countO) return false;

        // Rule 3: If 'O' has won, the number of 'X's and 'O's must be equal
        if (hasWon('O') && countX != countO) return false;

        // If all rules are satisfied, the board state is valid
        return true;
    }
};
```

Typescript Solution

```typescript
/**
 * Checks if a given tic-tac-toe board state is valid.
 * @param {string[]} board - An array of strings representing the tic-tac-toe board.
 * @return {boolean} - Returns true if the board state is valid, false otherwise.
 */
var validTicTacToe = function(board) {
    // Helper function to count occurrences of 'X' or 'O' on the board.
    const countOccurrences = (char: string): number => {
        return board.reduce((accumulator, currentRow) => {
            return accumulator + [...currentRow].filter(c => c === char).length;
        }, 0);
    };

    // Helper function to check if a player has won the game.
    const hasWon = (player: string): boolean => {
        // Check rows and columns for a win.
        for (let i = 0; i < 3; ++i) {
            if (board[i][0] === char && board[i][1] === char && board[i][2] === char) {
                return true;
            }
            if (board[0][i] === char && board[1][i] === char && board[2][i] === char) {
                return true;
            }
        }
        // Check both diagonals for a win.
        return board[0][0] === player && board[1][1] === player && board[2][2] === player ||
               board[0][2] === player && board[1][1] === player && board[2][0] === player;
    };

    // Count occurrences of 'X' and 'O'.
    const countX = countOccurrences('X');
    const countO = countOccurrences('O');

    // Ensure 'X' goes first and there's at most one more 'X' than 'O'.
    if (countX !== countO && countX - 1 !== countO) {
        return false;
    }

    // Check for a win by 'X'. If 'X' has won, there must be one more 'X' than 'O'.
    if (hasWon('X') && countX - 1 !== countO) {
        return false;
    }

    // Check for a win by 'O'. If 'O' has won, there must be an equal number of 'X' and 'O'.
    if (hasWon('O') && countX !== countO) {
        return false;
    }

    // If none of the invalid conditions were met, the board is valid.
    return true;
};
```

Time and Space Complexity

Time Complexity

The time complexity of the `validTicTacToe` function is $O(1)$ because the size of the board is fixed at 3×3, and the algorithm iterates over the 9 cells of the board a constant number of times to count occurrences of 'X' and 'O', and to check for wins.

To calculate a and b, we have two double loops that go through the 3×3 board, which would normally result in a time complexity of $O(n^2)$. However, since the board size is constant and does not grow with input, it results in a fixed number of operations that do not depend on any input size variable, so it is $O(1)$.

The `win` function is called at most two times (once for 'X' and once for 'O'). Within each call, it goes through each row, each column, and both diagonals to check for a win condition, which again, since the board is a fixed 3×3 size, results in a constant number of operations that are $O(1)$.

Space Complexity

The space complexity of the `validTicTacToe` function is also $O(1)$. No additional space is used that grows with the input size. The function only uses a fixed number of variables (a, b, and the board itself) and the space taken by the recursive stack during the calls to `win` doesn't depend on the input size since the depth of recursion is not affected by the input but by the fixed size of the 3×3 board.