1275. Find Winner on a Tic Tac Toe Game

Easy Array Hash Table Matrix Simulation

Problem Description

Tic-tac-toe is a classic game for two players, "A" and "B", played on a 3×3 grid. The objective is to be the first to get three of their own marks in a row either horizontally, vertically, or diagonally. In this LeetCode problem, we are given a list of moves where each move is represented by a pair of integers that correspond to a row and column on the grid. The task is to determine the outcome of the game based on the sequence of moves. The rules are as follows:

- Player "A" uses the mark 'X' and makes the first move, while player "B" uses the mark 'O'.
- Players alternate turns, placing their mark on an empty cell.
- The game ends when a player has three of their marks in a row or all cells are filled.
- If a player wins, their identity ("A" or "B") should be returned.
- If all cells are filled without a winner, the game is a draw and "Draw" should be returned.
- If the game has not ended, "Pending" should be returned.

The array `moves` represents the game sequence where `moves[i] = [row_i, col_i]` is the i-th move played on the grid at the cell defined by the row and column indices.

Intuition

The solution provided leverages a counting approach to keep track of the game's state without having to construct the entire grid after each move. Here's the intuition behind the solution:

- We create a count array `cnt` of length 8 to keep track of marks along the rows, columns, and diagonals. The first three positions are for the rows, the next three positions for the columns, and the last two for the two diagonals.
- We iterate over the moves inversely for both players, skipping moves alternatively (since players take turns).
- Each player's move is analyzed with respect to the potential winning conditions: filling a row, column, or diagonal. We increment the respective counter for the move made.
- After each move, we check if any counters have reached 3, indicating a winning condition.
- If there's a winner, we identify which player's turn it was based on whether the number of moves made so far is odd or even, and return "A" or "B" accordingly.
- If no winner is detected and the total moves made are 9, we declare the game a draw.
- If the game is not won and not all moves are played, we return "Pending" indicating unfinished status.

Solution Approach

Starting with the observation that a player wins if they achieve three of their symbols in a row, column, or diagonal, the code implements an elegant solution to identify the game's status after each move is made.

Here is a step-by-step approach to the implementation:

1. Create a counter array `cnt` with 8 zeros, where `cnt[0]` to `cnt[2]` correspond to the three rows, `cnt[3]` to `cnt[5]` to the three columns, and `cnt[6]` and `cnt[7]` to the two diagonals of the board.
2. Iterate over the moves in reverse order using the range `range(n-1, -1, -2)`, where `n` is the length of the moves list. This iteration goes backwards and skips every other move to alternate between players A and B as they play.
3. For each move, extract the row and column from `moves[k]`. Increment the respective counters in `cnt` for the row index `i` and column index `j+3`. This update simulates placing the 'X' or 'O' on the board.
4. Check for a diagonal match by:
   - Incrementing `cnt[6]` (the counter for the left-to-right diagonal) if `i == j`.
   - Incrementing `cnt[7]` (the counter for the right-to-left diagonal) if `i + j == 2`.
5. After updating the `cnt`, check if any value in `cnt` has reached 3 using the condition `any(v == 3 for v in cnt)`. This check is indicative of a winning condition.
6. If there is a winner, return "B" if `k` is odd (since the moves list includes player A's moves first, odd indexes will belong to player B during reverse iteration), or "A" if `k` is even, indicating it was player A's turn during the move that secured the win.
7. If no winner is found and the total number of moves `n` is 9 (meaning every cell has been filled without achieving a winning condition), the game ends in a Draw, and so we return `"Draw"`.
8. In the final case where no winner is found and `n` is less than 9, not all cells have been filled, thus the game is still ongoing, and the code returns `"Pending"`.

This method is efficient because it works incrementally, only tracking the essentials for determining the game outcome and not reconstructing the entire board's state after each move. It effectively makes use of a single-pass and counter-based technique, which is ideal for this array-based simulation of the tic-tac-toe game.

Example Walkthrough

Let's consider an example game represented by the sequence of moves:

```
moves = [[0,0],[2,0],[1,1],[2,1],[2,2]]
```

In this example, player A (using 'X') makes the first move at the top-left corner of the grid (0,0), followed by player B (using 'O') making a move at the bottom-left corner (2,0). The game proceeds with each player taking turns.

Now, let's walk through the moves according to the solution approach:

1. We initialize a counter array `cnt` with 8 zeros: `cnt = [0,0,0,0,0,0,0,0]`.
2. The length of the `moves` list is 5, and we start iterating from the last move to the first move, skipping every other one to simulate the alternating turns:
   - For `k = 4`, which is the move `[2,2]` by player A:
     - The row index is 2, so we increment `cnt[2]`.
     - The column index is 2, so we increment `cnt[5]`.
     - Since the row and column indices are equal, indicating a diagonal, we increment `cnt[6]`.
   - For `k = 2`, which is the move `[1,1]` by player A:
     - The row index is 1, so we increment `cnt[1]`.
     - The column index is 1, so we increment `cnt[4]`.
     - The indices are the same, indicating the diagonal again, so we increment `cnt[6]`.

   At this point, after the even-indexed moves by player A, `cnt` looks like this: `[0,1,1,0,1,1,2,0]`.
3. Then we iterate for player B:
   - For `k = 3`, which is the move `[2,1]` by player B:
     - The row index is 2, so we increment `cnt[2]`.
     - The column index is 1, so we increment `cnt[4]`.

(Note: Player B does not contribute to diagonal counters in these moves.)

4. After these iterations, we check if any counters reached the value of 3, which would indicate a win. Since none have, we proceed.

The current state of `cnt` is `[0,1,2,0,2,1,2,0]`. No values are 3, so no winner yet.

5. Since only 5 moves are made and the total number of moves is less than 9, we cannot have a 'Draw', so we return 'Pending'.

As a result, the output for this sequence will be `"Pending"`, as the game has not concluded yet.

Solution Implementation

Python

```python
class Solution:
    def tictactoe(self, moves):
        # Initialize the number of moves
        num_moves = len(moves)
        # '\counters' is a list that holds the count of marks for each row, column and diagonals
        # indexes [0,1,2] are for rows, [3,4,5] are for columns, [6] is for diagonal, [7] is for anti-diagonal
        counters = [0] * 8

        # Start from the last move and check if there is a winner after every move
        for idx in range(num_moves - 1, -1, -2):
            # Get the row and column index from the last move
            row, col = moves[idx]

            # Increment the corresponding row and column counters
            counters[row] += 1
            counters[col + 3] += 1

            # If the move is on the main diagonal, increment the corresponding counter
            if row == col:
                counters[6] += 1

            # If the move is on the anti-diagonal, increment the corresponding counter
            if row + col == 2:
                counters[7] += 1

            # Check if any counters reached 3, which would indicate a win
            if any(value == 3 for value in counters):
                # Return 'B' if the current index is odd (indicating player "B"'s turn), otherwise 'A'
                return "B" if idx % 2 else "A"

        # After all moves, if no winner is found, check if the board is full. If yes, it's a draw
        # Otherwise, the game is still pending
        return "Draw" if num_moves == 9 else "Pending"
```

Java

```java
class Solution {
    public String tictactoe(int[][] moves) {
        int totalMoves = moves.length; // Total number of moves made.
        int[] counts = new int[8]; // Array to keep track of the counts across rows, columns, and diagonals.

        // Iterate from the last move to the first, decrementing by 2 to alternate between players.
        for (int moveIndex = totalMoves - 1; moveIndex >= 0; moveIndex -= 2) {
            int row = moves[moveIndex][0]; // Row of the current move.
            int col = moves[moveIndex][1]; // Column of the current move.

            // Increment the count for the current row and column.
            counts[row]++;
            counts[col + 3]++;

            // Check for diagonal win condition (top-left to bottom-right).
            if (row == col) {
                counts[6]++;
            }

            // Check for anti-diagonal win condition (top-right to bottom-left).
            if (row + col == 2) {
                counts[7]++;
            }

            // Check if the current player has won (if any count reaches 3).
            if (counts[row] == 3 || counts[col + 3] == 3 || counts[6] == 3 || counts[7] == 3) {
                return moveIndex % 2 == 0 ? "A" : "B"; // Return winner "A" or "B" based on move index.
            }
        }

        // If all 9 moves are made and no winner, it is a draw.
        return totalMoves == 9 ? "Draw" : "Pending"; // If not a draw, the game is still pending.
    }
}
```

C++

```cpp
class Solution {
public:
    string tictactoe(vector<vector<int>>& moves) {
        int movesCount = moves.size(); // Total number of moves made

        // Array to keep track of the count of marks for rows, columns, and diagonals
        // Rows:    indices 0-2
        // Columns: indices 3-5
        // Diagonals: index 6 (left-top to right-bottom), index 7 (left-bottom to right-top)
        int count[8] = {0};

        // We start checking the game status from the last move backwards
        for (int k = movesCount - 1; k >= 0; k -= 2) {
            int row = moves[k][0];
            int col = moves[k][1];

            count[row]++; // Increment row count
            count[col + 3]++; // Increment column count

            // Check for diagonal - top-left to bottom-right
            if (row == col) {
                count[6]++;
            }

            // Check for anti-diagonal - bottom-left to top-right
            if (row + col == 2) {
                count[7]++;
            }

            // Check if player made 3 marks in a row, column, or diagonal
            if (count[row] == 3 || count[col + 3] == 3 || count[6] == 3 || count[7] == 3) {
                // If the index is even, it is player A's move; if odd, player B's move.
                return k % 2 == 0 ? "A" : "B";
            }
        }

        // If all 9 moves are made and no winner, it is a draw. Otherwise, game is pending.
        return movesCount == 9 ? "Draw" : "Pending";
    }
};
```

TypeScript

```typescript
function tictactoe(moves: number[][]): string {
    // Total number of moves made, corresponds to the number of turns played
    const totalMoves = moves.length;

    // An array to count the moves for rows, columns, and diagonals
    // Indices 0-2 for rows, 3-5 for columns, 6-7 for diagonals
    const moveCounters = new Array(8).fill(0);

    // Iterate through the moves in reverse, starting with the last move
    for (let moveIdx = totalMoves - 1; moveIdx >= 0; moveIdx -= 2) {
        const [row, col] = moves[moveIdx];

        // Increment the counters for the current row and column
        moveCounters[row]++;
        moveCounters[col + 3]++;

        // If the move is on the main diagonal, increment the counter
        if (row === col) {
            moveCounters[6]++;
        }

        // If the move is on the secondary diagonal, increment the counter
        if (row + col === 2) {
            moveCounters[7]++;
        }

        // Check if any counter has reached 3, indicating a win
        if (moveCounters[row] === 3 || moveCounters[col + 3] === 3 || moveCounters[6] === 3 || moveCounters[7] === 3) {
            // Determine the winner based on the index of the move
            return moveIdx % 2 === 0 ? 'A' : 'B';
        }
    }

    // If all 9 spaces were played and no winner, it's a draw
    return totalMoves === 9 ? 'Draw' : 'Pending';
}
```


```python
class Solution:
    def tictactoe(self, moves):
        # Initialize the number of moves
        num_moves = len(moves)
        # '\counters' is a list that holds the count of marks for each row, column and diagonals
        # indexes [0,1,2] are for rows, [3,4,5] are for columns, [6] is for diagonal, [7] is for anti-diagonal
        counters = [0] * 8

        # Start from the last move and check if there is a winner after every move
        for idx in range(num_moves - 1, -1, -2):
            # Get the row and column index from the last move
            row, col = moves[idx]

            # Increment the corresponding row and column counters
            counters[row] += 1
            counters[col + 3] += 1

            # If the move is on the main diagonal, increment the corresponding counter
            if row == col:
                counters[6] += 1

            # If the move is on the anti-diagonal, increment the corresponding counter
            if row + col == 2:
                counters[7] += 1

            # Check if any counters reached 3, which would indicate a win
            if any(value == 3 for value in counters):
                # Return 'B' if the current index is odd (indicating player "B"'s turn), otherwise 'A'
                return "B" if idx % 2 else "A"

        # After all moves, if no winner is found, check if the board is full. If yes, it's a draw
        # Otherwise, the game is still pending
        return "Draw" if num_moves == 9 else "Pending"
```


Time and Space Complexity

The time complexity of the given code is `O(1)`, as the function processes a fixed number (at most 9) of moves, regardless of the input size. The logic within the loop has a constant number of operations (checking the status of the game board, incrementing counters, and checking for a win condition), and thus does not grow with input size.

The space complexity of the code is also `O(1)` because the amount of memory used does not depend on the input size either. The `cnt` list has a constant size of 8, and no additional memory is allocated that would depend on the input.