Solution: Design Tic-Tac-Toe

Let's solve the Design Tic-Tac-Toe problem using the Knowing What to Track pattern.

We'll cover the following

- Statement
- Solution
  - Naive approach
  - Optimized approach using mark counting
    - Step-by-step solution construction
    - Just the code
    - Solution summary
    - Time complexity
    - Space complexity

Statement

Suppose that two players are playing a tic-tac-toe game on an $n \times n$ board. They're following specific rules to play and win the game:

- A move is guaranteed to be valid if a mark is placed on an empty block.
- No more moves are allowed once a winning condition is reached.
- A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Implement a **TicTacToe** class, which will be used by two players to play the game and win fairly.

Keep in mind the following functionalities that need to be implemented:

- **Constructor**, the constructor, which initializes an object of `TicTacToe`, allowing the players to play on a board of size $n \times n$.
- **move(row, col, player)** indicates that the player with the ID, `player`, places their mark on the cell (`row`, `col`). The move is guaranteed to be a valid move. At each move, this function returns the player ID if the current player wins and returns 0 if no one wins.

Constraints:

- $3 \leq n \leq 9$
- `player` should be either `1` or `2`.
- $0 \leq$ `row`, `col` $< n$
- Every call to `move()` will be with a unique `row`, `col` combination.
- The `move()` function will be called at most n^2 times.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time

complexity and any implementation constraints.

Naive approach

The naive approach to this problem is to check, after every move, whether the current player has won the game. Either player can win the game if the following conditions are met:

- They are able to mark an entire row.
- They are able to mark an entire column.
- They are able to mark all the cells of one of the two diagonals.

We mark the cell identified by the given row and column indexes with the current player's marker. Then, we check the following conditions to determine whether the current player wins:

1. Check the current row to see whether the rest of the cells in the row are also occupied by the current player.
2. Check the current column to see whether the rest of the cells in the column are also occupied by the current player.
3. In case the current cell is on one of the diagonals, check to see whether the rest of the cells in that diagonal are also occupied by the current player.

After every move, we iterated up to four times over n cells to check for each condition, row, column, diagonal (top-left to bottom-right corner), and anti-diagonal (top-right to bottom-left corner). Therefore, the time complexity is $O(n)$. The space complexity of this naive approach is $O(n^2)$, because we are using a board of size $n \times n$.

Let's see if we can use the mark-counting technique to reduce the time and space complexity of our solution.

Optimized approach using mark counting

The following are the three kinds of win scenarios in tic-tac-toe:

- Player 1 wins
- Player 2 wins
- No player wins

A player can win by marking all the cells in a row or a column, or along the diagonal, or, along the anti-diagonal. To identify whether either of the two players wins or if it's a tie between the two players, we can efficiently count the marks made on the tic-tac-toe board.

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

The constructor creates two arrays, `rows` and `cols`, each of size n, and initializes both arrays with `0`. These arrays are used to count marks placed in a particular row and column.

Java

main.java

PrintTicTacToe.java

```
1  import java.util. * ;
2  class TicTacToe {
```

```
  List < Integer > rows;
  List < Integer > cols;
  List < String > board;

  // TicTacToe class contains rows, cols, diagonal,
  // and anti_diagonal to create a board.
  // Constructor is used to create board of size n * n.
  public TicTacToe(int n) {
    this.rows = new ArrayList < Integer > (Collections.nCopies(n, 0));
    this.cols = new ArrayList < Integer > (Collections.nCopies(n, 0));

    // this.board is only used for printing purposes
    this.board = new ArrayList < >();
    this.board = PrintTicTacToe.initializeBoard(this.board, n);

    PrintTicTacToe.printTicTacToe(this.board, n);
    System.out.println("Initial state of counters: ");
    PrintTicTacToe.printBoardStates(rows, cols);
  }

  // move function will allow the players to play the game
  // by placing their mark at the row and col of their choice.
  public int move(int row, int col, int player)
  {
    return 0;
```

Design-Tic-Tac-Toe

Next, we implement the `move` function that will be used to play the game and also to check who wins the game. It takes three parameters. The first two parameters, `row` and `col`, identify the cell to mark. The third parameter, `player`, represents the player making this move.

Since there are n rows and n columns on a board, at each move, we need a way to check if the player has already marked all n cells in that row or column.

We increment the count when Player 1 marks a cell and decrement the count when Player 2 marks a cell. To implement this, we set the `currentPlayer` to `1` for Player 1's move and set it to `-1` for Player 2's move. We add this to the corresponding row and column in the `rows` and `cols` arrays, respectively. For example, if in a certain move, Player 2 places their marker at cell $(1, 2)$, we decrement the values at `rows[1]` and `cols[2]`.

Purely for the purpose of printing, we also set **O** for Player 1 and **X** for Player 2 on the board.

With this mechanism in place, we can see that if Player 1 marks all n cells in the i^{th} row, the value of `rows[i]` will be n. Similarly, if Player 2 marks all n cells in the i^{th} row, the value of `rows[i]` will be $-n$. The same holds true for any value in the `cols` array, for either player.

This property gives us a simple way to check winning conditions after every move. If, after the current player has marked the cell in the i^{th} row and the j^{th} column, the absolute value of either `rows[i]` or of `cols[j]` becomes equal to n, it means that this player has won the game and we can return the value of `player` as the winner of the game.

Let's see how we will count the marks for two moves played in sequence: `move(0, 0, 1)` and `move(0, 2, 2)`.

1 of 3

Let's look at the implementation of the approach discussed above:

Java

main.java

PrintTicTacToe.java

```java
import java.util.*;

class TicTacToe {
  List < Integer > rows;
  List < Integer > cols;
  List < String > board;

  // TicTacToe class contains rows, cols, diagonal,
  // and anti_diagonal to create a board.
  // Constructor is used to create board of size n * n.
  public TicTacToe(int n) {
    this.rows = new ArrayList < Integer > (Collections.nCopies(n, 0));
    this.cols = new ArrayList < Integer > (Collections.nCopies(n, 0));

    // this.board is only used for printing purposes
    this.board = new ArrayList < >();
    this.board = PrintTicTacToe.initializeBoard(this.board, n);

    PrintTicTacToe.printTicTacToe(this.board, n);
    System.out.println("Initial state of counters: ");
    PrintTicTacToe.printBoardStates(rows, cols);
  }

  // move function will allow the players to play the game
  // by placing their mark at the row and col of their choice.
  public int move(int row, int col, int player)
  {
      int currentPlayer = 1;
```

Design-Tic-Tac-Toe

Let's look at the output carefully to see how we did. We were able to correctly identify the winning condition in Game 1, but not in Game 2.

In Game 2, Player 2 actually won in the sixth move, but we didn't detect it because we were not keeping track of the marks on the diagonals. Recall that a player can also win a game if they mark all the cells along the diagonal (from the top-left corner to the bottom-right corner) or along the anti-diagonal (from the top-right

corner to the bottom-left corner). So, after every move, we must check the diagonal and the anti-diagonal. Keep in mind that regardless of the board size, there can only be one diagonal and one anti-diagonal.

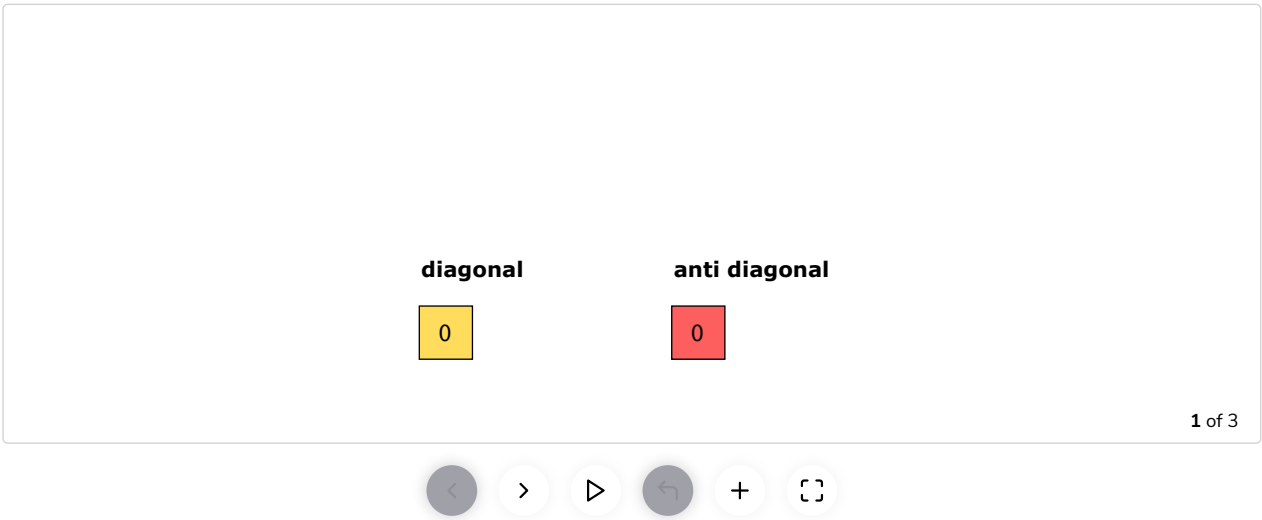
Since there are always n cells on the diagonal or anti-diagonal, the player must mark the cells on the diagonal or anti-diagonal n times to win the game. A cell (i, j) is on the diagonal if $i = j$ and on the anti-diagonal if $j = n - 1 - i$. To confirm this, you can check whether these cells on a 3×3 board are on the diagonal, the anti-diagonal, or on neither: $(0, 0)$, $(2, 0)$, $(1, 0)$, and $(2, 2)$.

For this, we introduce two new variables, `diagonal` and `antiDiagonal`, initialized with 0, to count the marks placed along the diagonal and along the anti-diagonal.

Similar to our method in the previous step, we increment the count when Player 1 marks a cell, and decrement the count when Player 2 marks a cell.

Based on this additional counting, we have a new winning condition. If the absolute value of `diagonal` or of `antiDiagonal` is equal to n, we return `player` as the winner.

Let's see how we will count marks for two moves involving the diagonals, played in the sequence, `move(0, 0, 1)` and `move(2, 0, 2)`.



Let's add this logic to our solution:

Java

main.java

PrintTicTacToe.java

```java
import java.util.*;

class TicTacToe {
  List < Integer > rows;
  List < Integer > cols;
  List < String > board;
  int diagonal;
  int antiDiagonal;

  // TicTacToe class contains rows, cols, diagonal,
  // and anti_diagonal to create a board.
  // Constructor is used to create board of size n * n.
  public TicTacToe(int n) {
    this.rows = new ArrayList < Integer > (Collections.nCopies(n, 0));
    this.cols = new ArrayList < Integer > (Collections.nCopies(n, 0));
    diagonal = 0;
    antiDiagonal = 0;
```

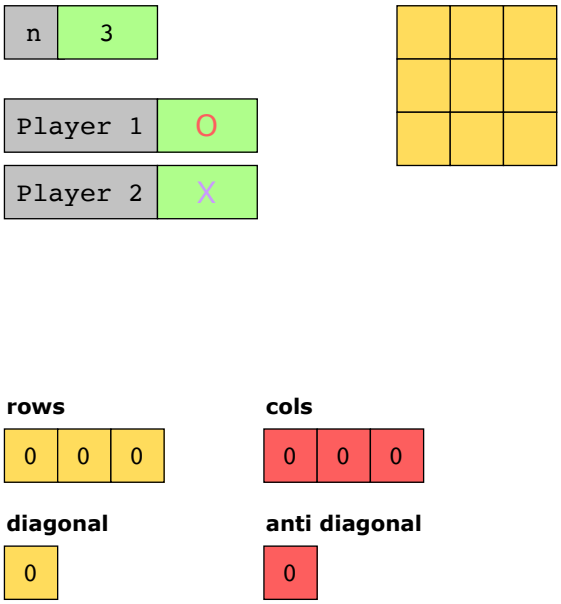
```
    antiDiagonal = 0;
    // this.board is only used for printing purposes
    this.board = new ArrayList < >();
    this.board = PrintTicTacToe.initializeBoard(this.board, n);

    PrintTicTacToe.printTicTacToe(this.board, n);
    System.out.println("Initial state of counters: ");
    PrintTicTacToe.printBoardStates(rows, cols, diagonal, antiDiagonal);
  }

  // move function will allow the players to play the game
  // by placing their mark at the row and col of their choice
```

Design-Tic-Tac-Toe

The following illustration presents a complete example:



1 of 8

Just the code

Here's the complete solution to this problem:

Java

```
import java.util.*;

class TicTacToe {
  List < Integer > rows;
  List < Integer > cols;
  int diagonal;
  int antiDiagonal;

  // TicTacToe class contains rows, cols, diagonal,
  // and anti_diagonal to create a board.
  // Constructor is used to create a board of size n * n.
```

```
12    public TicTacToe(int n) {
13      this.rows = new ArrayList < Integer > (Collections.nCopies(n, 0));
14      this.cols = new ArrayList < Integer > (Collections.nCopies(n, 0));
```

```
18
19    // move function will allow the players to play the game
20    // for given row and col.
21    public int move(int row, int col, int player)
22    {
23        int currentPlayer = -1;
24        if (player == 1)
25        {
26          currentPlayer = 1;
27        }
```

Design-Tic-Tac-Toe

Solution summary

Let's recap the solution:

1. Initialize two arrays to the track counts of marks made in rows and columns. Initialize both arrays to 0.
2. Initialize two variables to count the marks made along the diagonal and the anti-diagonal.
3. For every move (i, j), made by Player 1, increment `rows[i]` and `cols[j]` as well as `diagonal` and `antiDiagonal`, when applicable. For moves by Player 2, decrement the relevant counts.
4. If, at any point, the updated value in `rows` or in `cols` equals n, or if either of two variables counting marks on the diagonals equals n, return the current player as the winner.
5. Otherwise, the game is a tie, and we return `0`.

Time complexity

In the `move()` function, we update `rows[i]`, `cols[j]`, and, if applicable, `diagonal`, and `antiDiagonal` in every move. We then check all four values to see if any of the win conditions has been met. Since these are a constant number of operations, the time complexity of `move()` is $O(1)$.

For a board of size $n \times n$, `Constructor` takes $O(n)$ time to allocate space for $2n + 2$ counters and to initialize them to 0.

Space complexity

The solution uses $O(n)$ space, where n is the length of the arrays, `rows` and `cols`.