# Lab 08: Counting Tic-Tac-Toe Boards

COSC 290 - Fall ’21

<table>
<thead>
<tr>
<th>Starter File(s)</th>
<th>Lab08.zip (2 .java files)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Submission</td>
<td>Upload only the following file(s) to Moodle:</td>
</tr>
<tr>
<td></td>
<td>• BoardCounting.java</td>
</tr>
<tr>
<td></td>
<td>• Lab08Tester.java</td>
</tr>
<tr>
<td>Due Date</td>
<td>Monday, November 8th at 11:59PM for all lab sections</td>
</tr>
</tbody>
</table>

## 1 Overview

In this lab you will write functions to count various Tic-Tac-Toe board scenarios via recursive algorithms. To represent the different permutations of Tic-Tac-Toe boards, we will use a tree-like (or, more specifically, a trie-like) data structure.

## 2 The Tic-Tac-Toe Tree

Tic-tac-toe is a simple game played between two players, on a board like the one below:

```
    X
   ___
  |   |
  |   |
  |   |
```

The players, one represented by X’s and one represented by O’s, take turns filling in their respective character in a blank spot on the board (we’ll say the player going first is always X). The first person to have three of their symbols in the same row, column, or diagonal wins the game. Below is an example of a game where X has won:

```
    X
   ___
  | X |
  | O |
  | O |
```

In this lab, we will count different possible Tic-Tac-Toe boards using a tree-like data structure. In our tree, each "node" represents some Tic-Tac-Toe board, with the children of that node representing the possible boards that could result from that board. More specifically, each Node’s immediate children represent all of the possible resulting boards after the next move is made.

On the following page is a visualization of such a tree (partially showing the first three levels):
The root of the entire tree is our initial empty board. Each level down the tree, the X or O in green represents the next move on the board. The root’s children represent the moves the first player (X) can make on the very first turn.

Look at the first child of the root (with an X in the upper-left corner). All of the children of this node represent the possible board outcomes after the second turn, where on the first turn Player 1 put their X in the upper-left corner.

3 Your Task

Provided to you are two starter files:

- Board.java: an object representation of a Tic-Tac-Toe board.
- BoardCounting.java: a collection of methods to count the number of boards given certain scenarios.

Of these files, you can only modify BoardCounting.java. You will implement a series of counting methods outlined below. Each of these methods utilize recursion in some capacity. You can and should create helper functions (you may want to perform the recursion in a separate function for some of these algorithms), however you should not add any additional class or instance variables.

3.1 Count Total Nodes

First you must finish the implementation of the countTotalNodes function. The majority of this function is already implemented for you, you only need to correct one line (the base case).

This function counts and returns the number of boards in a tree where the argument Board object is its root. This count will ultimately be the size of the tree with the argument Board as its root – again, count includes duplicate boards and assumes the game is played until the board is full, even if a player wins.

3.2 Count Total Leaves

Implement countTotalLeaves. This function is similar to 3.1, but only counts the total leaves in the tree with the argument Board as its root. Leaf nodes are nodes with no children – in this case, each leaf will be a board that is completely filled in (since games don’t stop when if a player wins). The returned count may include duplicates.

3.3 Count Unique Nodes

Implement countUniqueNodes. This function is similar to 3.1, but only counts the unique nodes in the tree.

A declaration for a recursive helper function is provided here. Trace this code and pay special attention to the return type(s) – how will these data structure(s) be helpful here?
3.4 **Count Unique Leaves**

Implement `countUniqueLeaves`. This function is similar to 3.2, but only counts the **unique** leaves in the tree.

3.5 **Count Unique Leaves with Wins**

Implement `countUniqueLeavesWithWins`. This function is similar to 3.4, but now assumes that any tree stops when a player wins the game. This means that some leaf nodes will **not** be a completely filled board.

3.6 **Count Unique Nodes with Symmetry**

Implement `countUniqueNodesWithSymmetry`. This function counts the number of **unique nodes** in the tree (not just leaves), but also accounts for symmetry. This means that it won’t count two nodes that are the same given any number of **rotations** and/or **flips**.

4 **Pre-Lab Questions**

After reading this document and provided code, answer the following **before** we meet (we will discuss in-lab):

1. Given this tree representation of a tic-tac-toe board, what is the maximum number of children any node could have? In any given tree, is there more than one node that could have this number of children?
2. If we complete the example tree outlined in Section 2, would there exist any nodes in the tree that would be duplicates of one another?
3. There are only two public methods in Board which modify the Board object they are called on – what are they?
4. What would `nextPlayer()` return given the following Board?:

```
X
X O
X O
```

5. The four boards below are all **symmetries** of one another (relevant to the function you will implement for 3.6):

```
X | O | X
O X | X O
O X O | O X
```

Assume we have a variable `myBoard` which references a Board object representing the leftmost board in the example above. How would you transform `myBoard` to recreate the other three above? *Hint: review the functions provided to you in Board.java*

5 **Submission**

See the top of this document for the lab’s due date and time. When submitting your code, include **only** the files listed below:

- `BoardCounting.java`
- `Lab08Tester.java`