NAME : _____ NETID : _____

CS2110 Quiz 3

Problem 1: Ten T/F questions:

T	**F**	**A program calls buildTree and treeAnalysis on *n* objects. buildTree performs $1.5n^2$ operations, while treeAnalysis performs *nlog(n)+sqrt(n)* operations. If these are the only parts of the program that we're interested in, the overall complexity is $O(n^2)$.**
T	**F**	**If we create an ArrayList<? super Frog>, then since everything is an Object, anything can be stored into the ArrayList provided that we first cast it to Object.**
T	**F**	**A *postorder* traversal of a sorted data set stored in a balanced binary search tree will visit nodes in *reverse alphabetic order*.**
T	**F**	**In the Quicksort algorithm, performance will be *O(n log(n))* provided that the *pivot* for each step is the median for the values being sorted.**
T	**F**	**The worst performance for Bubblesort occurs when the input is already sorted in reverse order.**
T	**F**	**No matter what order elements are inserted into a priority queue, they can be extracted in sort order.**
T	**F**	**A stack has the property that elements pop out in a "most recently pushed first" order**
T	**F**	**A single instance of an object can simultaneously be referenced from many places, for example from a tree and a stack. If that object is modified, no matter which place it is accessed from, the caller will see the new value.**
T	**F**	**One reason that SWING users sometimes employ anonymous classes for in-line event handlers is because this style of code allows them to place the logic for an event handler at the same place in a program where other aspects of the functionality of the control are defined.**
T	**F**	**If multiple keys hash to the same value, a HashMap will still function correctly, but performance may be degraded because lookups will involve searching a list.**

Problem 2: The SeenAt class is defined this way: `public class SeenAt { String whenSeen; Frog whichFrog; }` and implements the *comparable* interface (first compares frogs, then for identical frogs, breaks ties by when that frog was seen). Define a method seenOften which takes an integer *n* and an unsorted ArrayList of SeenAt objects and returns a second ArrayList of frogs that were seen *n* or more times.

```
public static ArrayList<Frog> seenOften(int n, ArrayList<SeenAt> obs)
{
        Frog last = null;                                   // Most recently seen, if any
        int count = 0;                                      // How many times we've seen it
        ArrayList<Frog> rv = new ArrayList<Frog>;           // List of "frequently seen" frogs
        Collections.sort(obs);                              // Sort by frog
        for(SeenAt saw: obs)
        {
                if(last != null && last.equals(saw.whichFrog) == true) {
                        // Each time we see this same frog, we increment the counter
                        // When it hits the threshold, n, we add it to the result list
                        if(++count == n) rv.add(saw.whichFrog);
                } else {
                        // First encounter of a new, not previously seen frog
                        last = saw.whichFrog;
                        count = 1;
                        // Special case for n=0 or n=1
                        if(n <= 1) rv.add(saw.whichFrog);
                }
        }
        return rv;
}
```

Problem 3: What was the complexity of your solution to problem 2?

Same as the complexity of Collections.sort() or Arrays.sort(): *n log(n)*