NAME: ________________________________ NETID: ________________________

CS2110 Spring 2013 Prelim 2
April 16, 2013

Write your name and Cornell netid. *There are 5 questions plus one extra credit question on 7 numbered pages. Check now that you have all the pages. Write your answers in the boxes provided. Use the back of the pages for workspace. Ambiguous answers will be considered incorrect. The exam is closed book and closed notes. Do not begin until instructed. You have 90 minutes. Good luck!*

	1	2	3	4	5	ext	Total
Score	/20	/20	/20	/20	/20	/5	
Grader							

1. (20 points) A florist shop uses class Bouquet to manage a static ArrayList<Bouquet> in which all the bouquets currently on sale are listed. Here's the class (note: it uses an "enumerated" list of colors. This just means that the listed names (red, yellow…) can be used as "values" of objects of type Color).

```
class Bouquet {
   enum Color {red,yellow,orange,purple}; // The list of colors we use
   public Color primaryColor;       // primaryColor of this bouquet
   public int price;                // price in dollars
   public String name;              // Bouquet name in the catalog
   public long inventoryId;         // An inventory id number

   private static ArrayList<Bouquet> theInventory; // Bouquets in stock
}
```

(a) [5 points] Bouquet has several instance fields (primaryColor, price, etc) but theInventory, which lists all the Bouquets in the store, is static. Give one good reason that we might prefer that a store-wide inventory not be an instance field.
If there is just a single shared version of theInventory, making it a class (static) variable expresses that property nicely. Had it been declared to be an instance variable, a person reading the code might assume that there is a different inventory associated with each bouquet, and this could confuse them.

(b) [10 points] Write the method to implement its specification. Make your code match the comments.

```
/** Return the subset of the inventory for which c is the primary color */
public static ArrayList<Bouquet> floralOptions(Color c) {
 // Allocate a new ArrayList<Bouquet> called matching.
 ArrayList<Bouquet> matching = new ArrayList<Bouquet>();

// Iterate over the inventory, adding to "matching" the bouquets that match c
for(Bouquet bq: theInventory)
      if(bq.primaryColor == c)
           matching.Add(bq);

 // Return the new list.
 return matching;

}
```

(c) [5 points] If you modify this code to say ArrayList instead of ArrayList<Bouquet>, everything seems to work identically. Explain why cs2110 students use ArrayList<Bouquet> even though ArrayList also works.
By omitting the type, we are using an old notation in which Java automatically assumes that you intended "Object" as the generic type. But this weakens type checking and can even result in bugs, since ArrayList<Bouquet> and ArrayList<Object> are not really the same type.

2. (20 points) True or false?

a	[✓ T]	F	Even when working in a programming language other than Java, like Matlab or Python, no comparison-based sorting algorithm can achieve worst-case complexity better than O(n log n).
b	T	[✓ F]	The cost of the predefined operations offered by the abstract data types in the Java util package is always O(1).
c	[✓ T]	F	By overriding method compareTo, you can cause the Java util implementation of the priority queue to use a customized ordering method of your own design.
d	[✓ T]	F	Items pop from a stack in "last in, first out", or LIFO, order
e	T	[✓ F]	If you incorrectly implement the priority queue interface (your version of poll() has a bug and sometimes doesn't return the smallest element), Java's type checking will catch your mistake and the code won't compile.
f	[✓ T]	F	If the same element is inserted several times into a Java **Set**, the set will contain only a single instance of that element.
g	T	[✓ F]	If the same element is inserted several times into a Java **List**, the list will contain only a single instance of that element.
h	T	[✓ F]	If you implement an abstract data type that specifies that an operation should cost O(log N) time, the Java compiler will warn the programmer if the code might not have that property.
i	[✓ T]	F	The poll operation for a queue will return *null* if the queue is empty.
j	T	[✓ F]	If X is an object of type A, and B is neither a supertype of A nor a subtype of A, then (B)X will cause X to be converted into an object of type B. This is a form of "autoboxing".
k	[✓ T]	F	If a method takes a vector of length N as an input but always takes exactly 10 minutes to compute its result no matter what N was, we would say that it has complexity O(1).
l	[✓ T]	F	The worst-case complexity of removing an item from a heap containing N items is O(log(N))
m	[✓ T]	F	The worst-case complexity of inserting an item into a heap containing N items is O(log(N))
n	[✓ T]	F	If you use a very badly chosen hash function, and HashMap was implemented to use lists to resolve collisions, then looking up an item might have complexity O(n)
o	[✓ T]	F	The worst case complexity of QuickSort is O(N^2) but for most uses it runs in time O(N log N)
p	[✓ T]	F	If a method does something that requires exactly (N^2 + 3/2N) operations and your code calls that method N/2 times, the complexity of the resulting code is O(N^3)
q	T	[✓ F]	If the operation we are counting occurs in the body of a loop, then with *k* nested for-loops an algorithm would always have complexity at least O(N^k). Answer T if this is true for all code structures that fit this description, F if there might be ways to build code that fits the pattern and yet for which this complexity claim wouldn't be true.
r	T	[✓ F]	Suppose that X and Y are different objects of type A and that X.Equals(Y) is false. Then X.hashcode() != Y.hashcode(). That is, different objects have different hash codes.
s	T	[✓ F]	A hashcode() method must return a prime integer larger than 7.
t	[✓ T]	F	Javadoc is a tool that automatically produces manual pages from a special kind of comment you put on your methods and classes.

3. (20 points) Suppose we create a min-heap (of maximum size seven) by inserting five elements 71, 3, 8 15, -7 in the order shown (leaving room for two more). Draw the resulting heap first as a tree (exactly as seen in class) and as a vector (again, using the vector representation of a heap seen in class).

(a) 5 points. The heap:

$$\begin{tikzcd}
 & & -7 \arrow[dl] \arrow[dr] & \\
 & 3 \arrow[dl] \arrow[dr] & & 8 \\
71 & & 15 &
\end{tikzcd}$$

(b) 5 points. The corresponding vector (put a slash through “empty” elements):

-7	3	8	71	15	/	/

(c) 5 points. Now show the vector after we call poll() once, to extract the smallest element.

3	15	8	71	/	/	/

(d) 5 points. Suppose that you are looking at element *k* in the vector representation of a heap. Write a public static method that computes the index of the element containing the *parent* of *k*. (By index we mean that k is an offset into the vector representing the heap). The root has no parent; return 0 in this case. *Note: this has a simple answer. Very complex but correct answers won't get a lot of partial credit.*

```
/** Given a heap in vector representation, computes the index of the element
containing the parent of k. Returns 0 if called for the root.   */
public static int parent(int k) {
       // We know that the children of node n are at locations 2n+1 and 2n+2.
       // Below, we "invert" this rule:

       // Node 0 is the root and has no parent, return 0
       if(k == 0) return 0;

       // Child with an even index will have a parent at k/2-1
       if(k % 0 == 0) return k/2-1;

       // Child with an odd index: parent was at k/2
       return k/2;
}
```

A second possible solution:

```
public static int parent(int k) {
       // We know that the children of node n are at locations 2n+1 and 2n+2.
       return max(k-1, 0)/2;
}
```

4. (20 points) Suppose that we are given a binary tree in which nodes have public instance fields *left* and *right* pointing to the left and right children (or **null**, if there is no child on that side), and with user-defined equals(), compareTo(), and hashcode() methods. Write method treeEquality to implement the specification given below. You can assume that the input is a valid tree.

a. [15 points] *Hint: this can be done very easily! We'll deduct for inelegant, complex code.*

First, a comment about this solution: the TreeNode "equals" method doesn't know about recursion and only does equality checking on a single node at a time. This is standard in Java for many reasons, but the main ones are that otherwise, equality checking could easily crash while you are modifying a tree because there could periods when the child pointers aren't properly set. Thus node equality doesn't simply solve tree equality on its own. People who did assignments A3 and A4 were exposed to this issue. Just the same, we saw some solutions where people assumed that equality operates on the whole tree, not just a single node. There were also some people who were confused about the idea that we are given the root of some subtree and need to do a recursive check. We deducted for those mistakes because this is standard Java and matches many of the examples seen in class.

```
/** Return true iff ta and tb are identical trees —they are both null or both
trees with the same shape and corresponding nodes are equal. */
public static boolean treeEquality(TreeNode ta, TreeNode tb) {
    // Check for null trees
    if(ta == null || tb == null)
         return ta == null && tb == null;

    // Check that ta and tb have the same "value"
    if(!ta.equals(tb))
          return false;

    // Recursively check equality for left and right subtrees
    return treeEquality(ta.left, tb.left) && treeEquality(ta.right, tb.right);

}
```

A shorter answer that behaves the same way.

```
return ta==tb||(ta!=null&&tb!=null&&ta.equals(tb)&&
treeEquality(ta.left,tb.left)&&treeEquality(ta.right,tb.right));
```

… Some people wrote it this way to make it even more concise. Guess which version we preferred?

~~return~~ return ta==tb||(ta!=null&&tb!=null~~&&tb!=null~~&&ta.equals(tb)&&treeEquality(ta.left,tb.left)&&treeEquality(ta.right,tb.right));

b. [5 points] We overload treeEquality with this method:

```
/**  sets is a set of M balanced BSTs, each containing N nodes (for some M ≥ 0, N ≥ 0).
     Return true iff for all x, y in sets, treeEquality(x, y) is true */
public static Boolean treeEquality(Set<BST> sets) { … }
```

In English, describe an efficient way to use the overload of treeEquality() from part a to solve this problem (without changes to the method from part (a), and without giving the code for this new version of treeEquality). Keeping your answer short, now tell us what will be the worst-case O() time complexity of the new method, expressed in terms of N and M. Justify your answer.

We should start by emphasizing that although an object of type Set<BST> has no duplicates, by definition, the Set<T> ADT doesn't know about treeEquality() policy, and won't use it. So even though by definition a Set<BST> can't have two identical BST objects in it, it could still have two or more objects for which treeEquality is true. They could simply be different objects that happen to represent trees that are equal "in the eyes of the user-defined treeEquality method".

If the set has just one element, then this extended treeEquality is true. Otherwise, it suffices to take the first element and compare it item by item against the remaining M-1 elements. So we do M-1 invocations of the treeEquality method written in part a.

This is because we know that if A=B and A=C then B=C too, by transitivity. So there is no need to compare B and C, which avoids what would otherwise be an N^2 comparison task. Many people didn't realize this, so many solutions had a kind of nested for-loop structure: for all A,B in the set, make sure that treeEquality(A, B) is true. That works, but would be slower.

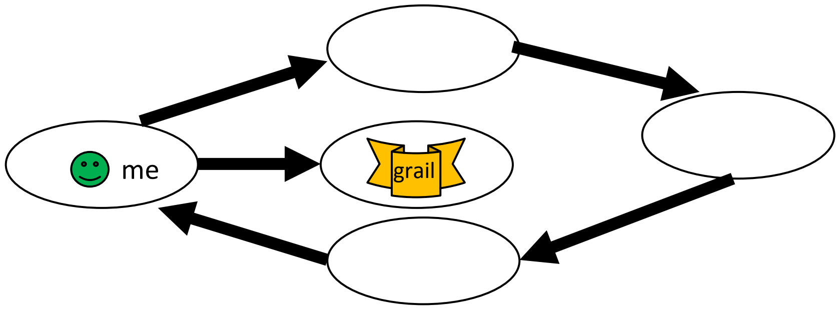
In the worst case, the treeEquality method of part a must visit every node of both trees. Thus if a tree has N nodes, treeEquality from part a may do N comparison operations (calls to .equals).

Thus (M-1)*N calls to .equals are done, and because M-1 is O(M), the worst-case complexity is O(M*N);

5. (20 points) You are writing code for a computer game in which the user explores a cave system consisting of caverns linked by tunnels, gathering weapons and treasures while fighting monsters. The game ends when the user is killed, or wins by discovering a special room containing the Grail of Hwynza. The cave is represented as an undirected graph. The Grail is within a finite distance of the entrance and each cavern is connected to finitely many other caverns.

a. [5 points] There is a famous way of escaping from any maze: Keep your hand in contact with the left wall and just keep walking. Can this method be used to hunt for the Grail? If yes, give a small proof that justifies your answer; if no, draw a cave system in which this method would fail.

No:



b. [5 points] Suppose that the cave is of infinite extent (but that any cavern has finitely many adjacent caverns). Would a depth-first search strategy be certain to find the Grail? Again, don't just say yes or no: give a proof of your answer.

No: There could be an infinitely long chain of empty caverns on the left in which case a depth-first strategy follows that chain forever. If the grail is to my right it never finds the grail.

c. [5 points]. Again, suppose that the cave is of infinite extent. Would a breadth-first search strategy be certain to find the Grail? Prove your answer.

Yes. A depth-first search explores all caverns at distance 0, then all at distance 1, etc. Since the grail is at finite distance, say D, then we will eventually explore distance-D caverns and will find it.

d. [5 points]. In a finite cave system with N caverns in which the Grail is hidden in a random cavern, what would be the average O() complexity of finding the Grail, given that "visiting a room" is the operation we wish to count (e.g. "visiting" has cost 1)? Again, justify your answer.

No matter what strategy we use we will need to visit every cavern because in the worst case, the last one we check will contain the Grail. Here the cave system is of finite size, so we can use depth-first search; the cost is thus O(N).

ext. [3+2 points extra credit] A *clique* of a graph G is a subgraph C such that for every two distinct vertices v1 and v2 in C, (v1, v2) is an edge of C.

A graph is maintained in adjacency-list form: it is given by variable G of type List<Vertex>, where class Vertex has a public instance field List<Vertex> neighbors, giving the vertices to which it has an edge.

For 3 points of extra credit, correctly complete the body of method cliqueTest, below.

For 2 additional points of extra credit write a second method maxCliqueSize, specified below.

Assume there is an existing method public static Set<T> intersect(Set<T> a, Set<T> b), and a second public static Set<T> union(Set<T> a, Set<T> b); the first returns a new Set<T> that is the intersection of it arguments, and the latter returns the union.

```
/** Return true iff the vertices in C form a clique */
public static boolean cliqueTest(List<Vertex> C) {
    // Check each vertex v in vertices to confirm that each other vertex
    // c in vertices is in v's neighbor set
    for(Vertex v1: C)
        for(Vertex v2: C)
             if(v1 != v2 && !v1.neighbors.contains(v2))
                  return false;
    return true;
}
```

```
/** Return the largest integer k such that G contains a clique of size k.
    Precondition: G has at least one vertex. */
public static int maxCliqueSize(List<Vertex> G) {
   return maxCliqueSize(G, new ArrayList<Vertex>());
}
/* For each unvisited vertex in C, intersect its neighbor set against the
   neighbor sets of its neighbors.  Search the resulting subsets for cliques.
   Any clique would need to be within one of these subsets.
   The list "visited" is used to avoid "rechecking" the same vertex again
   and again, which would cause infinite recursion. */
public static int maxCliqueSize(List<Vertex> C, List<Vertex> visited) {
   // Base cases: if argument set is a clique, return the set size
   if(C.size() == 0 || cliqueTest(C))
      return C.size();
   int mc = 1;
   for(Vertex v1: C)
   {
      if(!visited.contains(v1))
      {
        // Compute a new version of visited that includes v1 too.  We'll use
        // vp in the recursive call, but compute it just one for efficiency.
        Set<vertex> vp = union(visited, new Set<Vertex>() {v1});
        // nodes aren't neighbors of themselves, so we need to add each node to
        // its own neighbor set before calling intersect in the code that follows.
        for(Vertex v2: C)
           if(!visited.contains(v2))
              mc = max(mc,maxCliqueSize(
                    intersect(union(v1.neighbors, new Set<Vertex>() {v1}),
                              union(v2.neighbors, new Set<Vertex>() {v2})),
                    vp);
      }
   }
   return mc;
```

```
    }
```