NAME: ________________________________ NETID: ________________________

CS2110 Spring 2013 Final
10 May 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! And have a nice summer!*

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

1. (20 points) Your team is writing a program that searches for asteroids in photos collected by the Hubble Telescope. The program reads in images, scans for possible objects that might need more study, and prints details for each one it finds that go into a list for later examination.

An initial version of the program was able to process 6 images per second, but then you changed it into a multi-threaded version with 3 threads: (1) one reads images from a big database, (2) one scans each image for possible unknown asteroids, and (3) one prints information for anything found. Two bounded buffers were used between the stages: one holds images that have been read but not yet processed, and one holds descriptions of objects for further study until they have been printed.

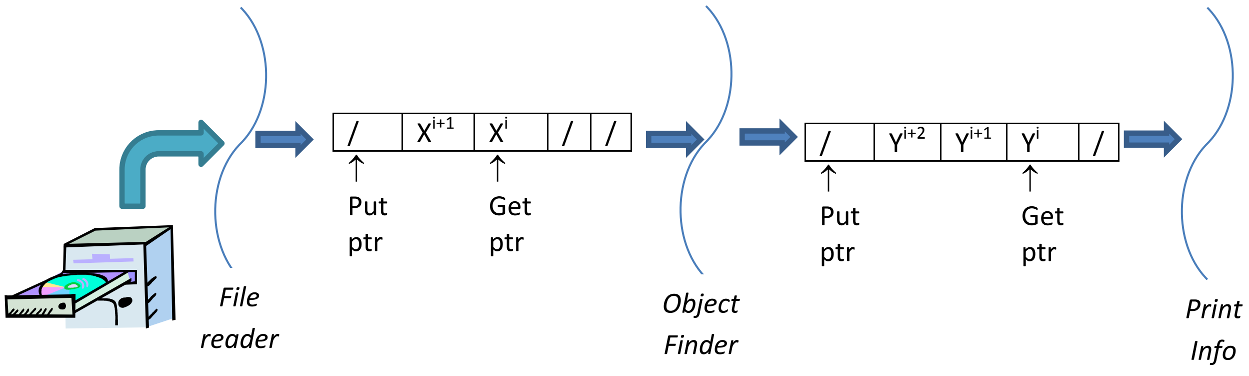
Everyone is excited because with this change, the program is now running at 23.7 images per second — almost four times faster. Your job is to explain why concurrency can give such a big benefit.
[5 points for the explanation, 5 more points for the nice graphic]. Explain what a thread is, why we use the term "producer/consumer" for threads used in the manner described here, and what a bounded buffer does. Also draw a picture showing the 3 threads and the 2 bounded buffers (make them 5 elements each), labeling each with its name. Design the graphic as if you intend to use it when explaining the program to a teammate.

A thread is a context within which execution occurs. Multiple threads can execute in a single process, sharing the objects accessible to them within the address space of that process. Thus a thread has private local variables but shares global variables with other threads.

A producer/consumer relationship arises when one thread creates or produces some sort of objects that will be analyzed or processed by a second thread, the consumer.

A bounded buffer is a form of queue that can hold some fixed maximum number of objects, allowing the producer to run slightly faster than the consumer. If the buffer fills up, the producer must wait. If the buffer becomes empty, the consumer must wait.

Picture: below are the three threads and two buffers. I've shown the first buffer with two out of a maximum of five objects in it, and the second with three objects, also from a maximum of five.



(a) [5 points]. With just 3 threads, you got nearly a 4x speedup. Explain why threads can speed up a program. Then explain why the speedup could be larger than the number of threads that were used. *Hint: think about a thread forced to wait, like when it reads data from a disk that needs to spin up and reach the right spot before the read can be performed.*

With threads, the program might be able to remain busy even while performing some task that requires waiting (like the disk I/O in the hint, but this is one of many things that might wait). Even with a single CPU (core), we can gain a speedup by having other things to do in such situations: the CPU utilization increases because during periods the CPU would have been idle, it can compute on behalf of one of the other runnable threads. With a modern multi-core machine, the speedup can be even greater because we can also exploit physical parallelism, allowing different cores to work on different tasks concurrently.

(b) [5 points]. The bounded-buffer class we saw in lecture uses the Java "synchronized" keyword and the "wait" and "notifyAll" methods. Explain "synchronized," "wait" and "notify".

Synchronized: tells Java that a lock on the specified object (or "this" if none is specified) must be acquired and held while executing some protected block of code (a "critical section").

Wait: releases a synchronization lock and causes this task to pause on the wait queue of the specified object. Used to wait until some desired condition becomes true. You can also specify a timeout, in which case the wait runs for at most the specified period of time.

NotifyAll: wakes up all the threads waiting on the specified object. They will contend to reacquire the synchronization lock and then will resume execution on the line after the wait, but only one at a time (because only one can actually hold the lock at a time). The thread that wakes up will normally need to recheck to see if the desired condition now holds.

2. (20 points) True or false?

a	[✓ T]	F	If we think of the Internet as a graph with vertices corresponding to routers and weighted edges corresponding to network links and their delays, we could use Dijkstra's algorithm to compute a lower bound on the delay from a source to each of a set of destinations.
b	[✓ T]	F	If a graph G is connected, then the minimum spanning tree for G will also be connected.
c	[✓ T]	F	We can test whether a directed graph G is a DAG by repeatedly removing vertices with in-degree 0. G is a DAG if eventually every vertex is removed.
d	T	[✓ F]	If your program is crashing because of deadlocks but your computer only has one core, running it on a multicore machine might guarantee that deadlocks can no longer occur.
e	T	[✓ F]	With a quantum computer, operations always have O(1) complexity.
f	T	[✓ F]	Technologies like JQL allow us to express complex operations on large collections of data very concisely. A disadvantage is that these approaches are so slow: by writing the same operations by hand, we would almost always get much better performance.
g	[✓ T]	F	If we use Java to perform operations on a database *<u>stored on disk</u>*, we often would need to explicitly tell Java when to save updates back into the database.
h	[✓ T]	F	Consider classes A and B, with neither being a subclass of the other. If you create an ArrayList<A>() and attempt to insert an object declared to have type B into it, **you will get a compile-time error**.

i	T	[✓ F]	If a function includes a try { *something;* } catch(IOException e) { handle-IOExceptions; } then the function must return null in the event that an exception does occur.
j	T	[✓ F]	Suppose method X contains an exception handler for NullReferenceException, and inside the try X calls method Y, which doesn't catch NullReferenceException. Now suppose that on line of 10 of method Y, a null reference occurs. *Then after X handles the exception, Y will resume on line 11: the line after the one that caused the exception.*
k	T	[✓ F]	Dijkstra's algorithm computes the "all pairs" shortest paths in a directed graph with non-negative edge weights. That is, after running the algorithm a single time, starting from some single vertex, it will have computed *MinDist(x,y)* for all vertices x and y in the graph.
l	T	[✓ F]	Suppose class A implements interface T. If you pass an object of type A to a method that expects an argument of type T, the code compiles but Java will throw a runtime exception.
m	T	[✓ F]	Suppose B is a subclass of A and you create a new A object. Then B's constructor will be executed automatically by Java because when a superclass is instantiated, the constructors for all the subclasses are executed too.
n	T	[✓ F]	Given graphs G and H, suppose that g_0 is a randomly picked vertex in G, and h_0 is a randomly picked vertex in H. Then *any* correct implementation of the .equals() method for graph vertices *must* return true for g_0.equals(h_0) if G and H represent the same graph, and false if G and H differ in any way (e.g. different vertices or edges, different weights, etc).
o	[✓ T]	F	A hashcode method that often has collisions would be legal, but could cause terrible performance for Java utilities such as HashMap and HashSet.
p	[✓ T]	F	If objects x, y, and z are of type A and x.equals(y), and x.equals(z) then y.equals(z). You may assume that the implementation of "equals" is correct.
q	[✓ T]	F	An invariant is a property of a data structure that is always true.
r	[✓ T]	F	A recurrence relation is an equation that defines a sequence. One or more base cases are given, and then each subsequent value is defined as a function of the preceding ones.
s	[✓ T]	F	One important form of inductive proof starts by proving the base case(s) and then shows that if the desired property holds for all $N_0 < k < N$, it also holds for k=N.
t	[✓ T]	F	Concurrent threads that both read and update some shared object can avoid race conditions by using Java synchronization correctly.

3. (20 points) In class we learned about graph *colorings*: an assignment of "colors" to the vertices in a graph such that no two adjacent vertices have the same color. Suppose a vertex is defined by the class shown below. Your job is to provide code for some methods that would be used to find out how many colors are in use in a graph and whether or not the assignment of colors is a graph coloring. We are *not* asking you to write the code that computes the coloring: you can assume each node already has a color when these methods are called. Moreover, you may assume that the methods are *never* called with null arguments, and that the graph itself is undirected and fully connected.

```
public class Vertex {
    public enum Color { red, yellow, blue, black, green, … };   // A long list of colors
    public Color vColor;                                        // Color of this node
    public Set<Vertex> neighbors;                               // Neighbors of this node
}
```

a. [5 points] Write the body of the method shown below

```
/** Return true if the neighbors of v have colors different from v's color.
  *   Return false if some neighbor of v has the same color as v. */
public static boolean colorCheck(Vertex v) {
    for(Vertex neighbor: v.neigbors)
```

```
            if(neighbor.vColor == v.vColor)
                return false;
        return true;
}
```

b. [5 points] Write the body of the method shown below.

```
/** Return true if the current vertex coloring of graph g is a legal graph
coloring, false otherwise */
private static boolean isALegalGraphColoring(Set<Vertex> g) {
// You can use colorCheck.
    for(Vertex v: g)
        if(!colorCheck(v))
            return false;
    return true;
}
```

c. [5 points] Write the body of the recursive method shown below

```
/** Precondition: v is not in set setV of visited vertexes. Add to setC the colors in use by v and
      vertexes reachable from v along paths of vertexes not in setV. Add to setV all visited vertexes. */
private static void setOfColors(Vertex v, Set<Vertex> setV, Set<Color> setC) {
     setC.Put(v.vColor) ;
     setV.put(v) ;
     for(Vertex neighbor : v.neighbors)
        if( !setV.contains(neighbor)
            setOfColors(neigbor, setV, setC) ;
}
```

d. [5 points] Write the body of the method shown below

```
/** Return the number of distinct colors in use in v and vertexes of the graph that are reachable from v.
 *   (You are expected to use function setOfColors, above). */
public static int numberOfColors(Vertex v) {
          HasSet<Vertex> vset = new HashSet<Vertex>();
          HashSet <Color> cSet = new HashSet<Color>();

          setOfColors(v, vSet, cSet);
          return cSet.size();
}
```

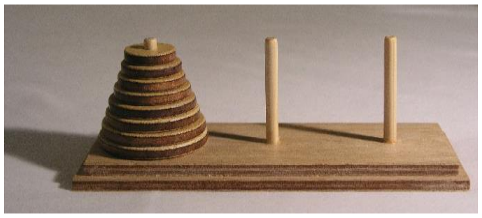
4. (20 points)

You and your CS2110 friends have decided to get rich by selling the ultimate iPhone and Android app for rock climbers. A rock-climbing location might have a number of possible routes. A route starts at some point on the cliff, then has a series of pitches (sections), each rated by difficulty and by the amount of time required, and finally ends at the top of the cliff. *A pitch goes from some point on the cliff to some other point on the cliff; it may involve climbing sideways or even back downward, not just up, and there are routes that climb up, circle around, and then return to a point previously visited.*

a. [5 points] How do you represent a great cliff for rock climbing as a Java data structure? Give a short and clear description of the data structure you would use. If your data structure makes use of some standard Java utility or interface (FIFO queue, priority queue, set, stack, list, set, etc) you don't need to tell us how that standard Java utility works. Just tell us what you would use and why.

I would represent a cliff as a set of routes, and each route would be a directed graph. There would need to be a set of vertices representing start points, and perhaps also a set representing exit locations on top. Each route would be a list of vertices: climb from here to here. Those should be adjacent in the graph (corresponding to a pitch) and labeled with the difficulty and time required.

b. [5 points] Define: *directed acyclic graph* (*DAG*). Would the data structure you recommended in part 4a be a DAG? Explain why or why not.



A cycle is a connected path in the graph that returns to some vertex. While some climbs could be acyclic, the definition makes it clear that the potential for cycles does exist: there are climbs that go to some point, climb around in a little circle on the face of a cliff, and then descend. Represented in the graph, these will be cycles.

c. [5 points] Define: *connected graph*. Would the data structure you recommended in part 4a be connected? Explain why or why not.

Again, while there may be particular cliffs for which all the climbs do start at some single point, in general one will see locations at which there are many points of "entry". Each of these different routes could be completely disjoint from the other routes. Thus there might be multiple independent subgraphs, with no connectivity between them.

d. [5 points] A climber might want the app to recommend the fastest way to the top of the cliff that doesn't involve any pitches with difficulty greater than some limit. Tell us what problem is solved by Dijkstra's algorithm. Then explain how we can adapt this algorithm to solve the problem of recommending a route for this climber. *Note: Recall that a given cliff could have many possible starting points. The recommended route would need to include a recommended starting point and ending point.*

Dijkstra's algorithm would be a very good choice for this kind of climber. We would run the algorithm once for each possible starting point, and it would compute the shortest path to each of the exit points accessible from that starting point, but not including any edges that exceed the limits in terms of difficulty or expected time. Then over the set of (entry, exit) pairs, we would identify the one that has the shortest cost and recommend that route. Dijkstra's algorithm normally computes the distance from entry to exit but as presented by Professor Gries, doesn't actually track the exact sequence of vertices in order. However, adding that extra information wouldn't be at all difficult, and this would allow us to also print the route pitch by pitch – a necessary step, because without that pitch-by-pitch information, there may be ambiguity in the solution (e.g. think of an entry point that has multiple possible routes to the same exit at the top, some easier and faster than others).

5. (20 points) To the right is a picture of a game called "Towers of Hanoi". The game starts with a pile of disks on one of the three rods, as seen in the picture. Notice that the disks are arranged with the smallest on top and the largest on the bottom. The game involves moving the disks from the rod on the left to the rod on the right, one at a time, but without ever putting a large disk on top of a smaller one. Let's name the rods from left to right: A, B and C.

Towers of Hanoi can be solved in a very elegant way using recursion: To move N disks from rod A to rod C, first move N-1 disks from rod A to rod B, then move one disk (this will be the biggest of them) from rod A to rod C, and finally move N-1 disks from rod B to rod A. The nice thing about this method is that because the biggest disk was moved last, it automatically satisfies the rule about never putting a big disk on a smaller disk. Moreover, as seen below, the instructions don't need to say "which" disk to move, since the one to move is always on top. Your job is to write a method that implements this recursion, printing instructions as it does so. Here's what it should print for a few values of N. *Hint: notice how the "move N-1 disks to the middle tower" step is performed in these three examples. Draw little pictures if that might help you build up intuition into how this works.*

With N=1: *tower("A", "B", "C", 1) prints*

Move one disk from A to C

With N=2: *tower("A", "B", "C", 2) prints*

Move one disk from A to B
Move one disk from A to C
Move one disk from B to C

With N=3: *tower("A", "B", "C", 3) prints*

Move one disk from A to C
Move one disk from A to B
Move one disk from C to B
Move one disk from A to C
Move one disk from B to A
Move one disk from B to C
Move one disk from A to C

a. [5 points].

```
/** Print instructions to move N disks from rod startRod to rod targetRod using rod otherRod as
  *  as the "middle one". (Note: the base case is when N = 0; there is nothing to print.)
  *  Precondition: No disk is on top of a smaller one. The N disks to be moved are smaller than the
  *  ones on rods targetRod and otherRod. */
public static void tower(String startRod, String otherRod, String targetRod, int N) {
    if(N > 0)  {
        Tower(startRod, otherRod, N-1);
        System.out.println("Move one disk from " + startRod + " to " + targetRod);
        Tower(otherRod, targetRod, N-1);
    }
}
```

b. [5 points]. We wish to figure out the cost of solving this problem, using a recurrence relation T(N). What would be the base case for T(N)?

T(0) = 0 or T(1) = 1

c. [5 points]. Write T(N) as a more general recurrence function that expresses the cost of moving N disks in terms of the cost of moving N-1 disks.

T(N) = 2T(N-1)+1

d. [5 points]. Show how this recurrence relation can be solved to obtain a simple equation for the cost of moving N disks using method tower(). You can solve it in any way you wish –by algebraic manipulation, by looking at examples and inferring from them, by drawing a tree of recursive calls, or anything else.

$T(N) = 2T(N-1)+1$

$= 4T(N-2)+2+1$

$= 8T(N-3)+4+2+1$

. . .

$= 2^{N-1} + 2^{N-2} + \dots + 2^0$

$= 2^N - 1$

ext. [5 points extra credit]
Suppose that an Internet provider has a graph representing all the network routers and the network links that it owns. The provider uses Prim's algorithm to compute a minimum spanning tree and provides you with this tree. The original class TreeNode is shown below. Now the provider obtains a list of (source, destination, traffic) triples. The class is also shown below. Each object of type Triple lists a node in the graph (the source for some flow of network traffic), a second node (the destination), and an integer giving the data rate, measured in megabits/second.

Write the body of the static method given below. The method is defined within the class TreeNode, and you can extend TreeNode with additional fields if you wish, use helper methods, etc.

```
public class Triple {
      public TreeNode source, destination;
      public int traffic;
}

public class TreeNode {
     public  Set<TreeNode> neighbors;      // The vertices adjacent to this one in the spanning tree

   /** v is a node of a spanning tree of a network and tL a list of traffic triples. Compute and print
     *  the traffic load on each network link, sorted from maximum to minimum traffic.  Hint: if vertex
    *   v has a neighbor x, then x will list v as a neighbor too.  Your algorithm will need to be careful
    *  not to get stuck in a loop, visiting v, then x, then v, then x….
    */
   public static void printLinkLoads(TreeNode v, Triple[] tL) {
         Array.Sort(tL);
         for(Triple tp: tL)
             if(containsEdge(v, new HashSet<TreeNode>(),  tp.source, tp.destination))
                   System.out.println("There is an edge ("+v+","+neightbor+") with load="+t.traffic);
   }

   /** Returns true if (from,to) is in the spanning tree, false if not.
     *  The algorithm is just a DFS in which the set "visited" is used to avoid loops
     */
   public static boolean edgeInTree(TreeNode root, HashSet<TreeNode> visited, Treenode from,
Treenode to) {
         if(visited.contains(v))
```

```
            return false;
        if((v == from && v.neighbors.contains(to)) || (v == to && v.neighbors.contains(from))
            return true;
        visited.add(v);
        foreach(Treenode n: v.neigbors)
            if(containsEdge(n, visited, from, to))
                return true;
        return false;
    }

}
```