Name:_____

CSE 332, Spring 2010, Final Examination
8 June 2010

Please do not turn the page until the bell rings.

Rules:

- The exam is closed-book, closed-note. You may use a calculator *for basic arithmetic only.*
- **Please stop promptly at 4:20.**
- You can rip apart the pages, but please staple them back together before you leave.
- There are **10** questions (many with multiple parts) worth **10 points each** for a total of 100 points.

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly circle your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. Suppose we sort an array of numbers, but it turns out every element of the array is the same, e.g., `{17, 17, 17, ..., 17}`. (So, in hindsight, the sorting is useless.)

   (a) What is the asymptotic running time of insertion sort in this case?

   (b) What is the asymptotic running time of selection sort in this case?

   (c) What is the asymptotic running time of merge sort in this case?

   (d) What is the asymptotic running time of quick sort in this case?

   Solution:

   (a) $O(n)$

   (b) $O(n^2)$

   (c) $O(n \log n)$

   (d) $O(n^2)$

Name:_____

2. Consider this code. Assume any client passes an appropriate `Comparator` object for comparing elements. These are just the assumptions we made throughout the course when computing with `Comparator` objects:

   - It returns 0 for equal elements.
   - It returns a negative number if the first element is "less than" the second and a positive number if the first element is "greater than" the second.
   - Elements are totally orderd (transitive, anti-symmetric).

```
interface Comparator<E> {
  int compare(E e1, E e2);
}
```

```
class C {
  static <E> boolean mystery(E[] array,
                             Comparator<E> c) {
    for(int i=0; i < array.length-1; i++) {
      for(int j=i+1; j < array.length; j++) {
        if(c.compare(array[i],array[j])==0)
          return false;
      }
    }
    return true;
  }
}
```

   (a) In English, *what* does `mystery` compute? Do not explain *how* it computes it.

   (b) What is the worst-case running time of `mystery` in terms of n, the length of `array`?

   (c) Describe in English and/or pseudocode an algorithm for the same problem with running time $O(n \log n)$. You do not need to explain the implementation of any standard algorithms you use as part of your solution. Assume it is acceptable to mutate the contents of the array.

   (d) Now assume it is *not* acceptable to mtuate the contents of the array. How can you modify your algorithm to use extra space but not change the asymptotic running time?

Solution:

   (a) It returns `true` if no two elements of the array are equal and `false` otherwise. In other words, it determines if all elements of the array are distinct.

   (b) $O(n^2)$

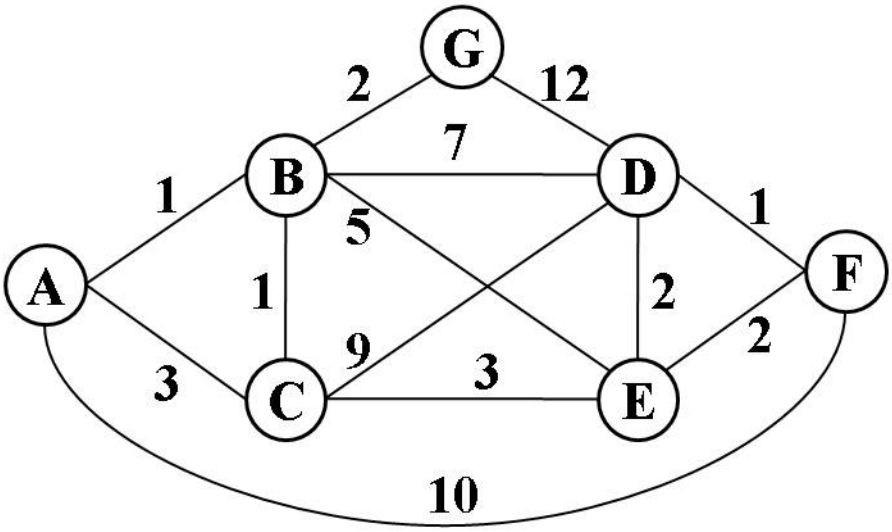
   (c) Sort the array, which takes time $O(n \log n)$. Then make one $O(n)$ pass through the array as follows:

```
for(int i=0; i < array.length-1; i++)
  if(c.compare(array[i],array[i+1]))
    return false;
return true;
```

   (d) Simply make a copy of the array first in time $O(n)$ and then employ the algorithm from part (c) on the copy.

Name:_____

3. Consider the following undirected, weighted graph:



Step through Dijkstra's algorithm to calculate the single-source shortest paths from A to every other vertex. Show your steps in the table below. Cross out old values and write in new ones, from left to right within each cell, as the algorithm proceeds. Also list the vertices in the order which you marked them known. Finally, indicate the lowest-cast path from node A to node F.

Solution:
Known vertices (in order marked known): A B C G E D or F D or F

Vertex	Known	Cost	Path
A	Y	0	
B	Y	1	A
C	Y	3 2	A B
D	Y	8 7	B E
E	Y	6 5	B C
F	Y	10 7	A E
G	Y	3	B

Lowest-cost path from A to F: A to B to C to E to F

Name:_____

4. In Java using the ForkJoin Framework, write code to solve the following problem:

   - Input: A `String[]`
   - Output: A `Pair` of (a) the number of words starting with the letter `'c'` and (b) the length of the longest word starting with the letter `'c'`. (If no words start with `'c'`, the length is irrelevant.)

   Your solution should have $O(n)$ work and $O(\log n)$ span where n is the array length. Do *not* employ a sequential cut-off: the base case should process one `String`.

   We have provided some of the code for you. You only need to provide one class definition:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
class Pair {
  int count;
  int longest;
  Pair(int c, int l) { count = c; longest = l; }
}
class Main{
  static final ForkJoinPool fjPool = new ForkJoinPool();
  Pair processCWords(String[] array) {
     return fjPool.invoke(new ProcessCWords(array,0,array.length));
  }
}
```

   Solution:

```
class ProcessCWords extends RecursiveTask<Pair> {
   String[] arr;
   int lo;
   int hi;
   ProcessCWords(String[] a, int l, int h) { arr=a; lo=l; hi=h; }
   public Pair compute() {
     if(hi==lo+1) {
       if(arr[lo].charAt(0)=='c')
         return new Pair(1,arr[lo].length());
       else
         return new Pair(0,0); // length < any word starting with 'c'
     } else {
        ProcessCWords left = new ProcessCWords(arr,lo,(hi+lo)/2);
        ProcessCWords right = new ProcessCWords(arr,(hi+lo)/2,hi);
        left.fork();
        Pair ans1 = right.compute();
        Pair ans2 = left.join();
        return new Pair(ans1.count+ans2.count,
                        Math.max(ans1.longest,ans2.longest));
     }
   }
}
```

   It's also fine to mutate one `Pair` from a subproblem and return it. The code above will throw an exception if the array contains empty strings. We didn't take off for not noticing this, but we did give +0.5 for explicitly checking this case, without allowing more than a total of 10 points for the problem.

Name:_____

5. You are at a summer internship working on a program that currently takes 10 minutes to run on a 4-processor machine. Half the execution time (5 minutes) is spent running sequential code on one processor and the other half is spent running parallel code on all 4 processors. Assume the parallel code enjoys perfectly linear speedup for any number of processors.

   Note/hint/warning: This does *not* mean half the work is sequential. Half the *running time* is spent on sequential code.

   Your manager has a budget of $6,000 to speed up the program. She figures that is enough money to do only one of the following:

   - Buy a 16-processor machine.
   - Hire a CSE332 graduate to parallelize more of the program under the highly dubious assumptions that:
     - Doing so introduces no additional overhead
     - Afterwards, there will be 1 minute of sequential work to do and the rest will enjoy perfect linear speedup.

   Which approach will produce a faster program? Show your calculations, including the total *work* done by the program and the expected running time for both approaches.

   Solution:
   This is an application of Amdahl's Law. Let S be the running time for the sequential portion, L be the running time for the parallel portion on 1 processor and P be the number of processors. Then $T_p = S + L/P$. Initially T_P is 10 minutes, S is 5 minutes and $P = 4$, which means L is 20 minutes. So the total work is 20+5=25 minutes.

   Therefore, $T_{16} = 5 + 20/16$ is 6.25 minutes and that's the "buy a 16-processor" option.

   Under the "hire an intern" option S becomes 1 and L becomes 24 minutes. So the total time is $1+24/4$ which is 7 minutes.

   So the better choice is to buy the computer — apologies to those of you hoping for the job.

Name:_____

6. (a) On our midterm we saw a somewhat complicated algorithm for finding the second-smallest element in a binary search tree. It descendend to the correct node without making any modifications to the tree. Suppose this algorithm is implemented in a method `secondSmallestMidterm`. Would it be correct for two threads to execute `secondSmallestMidterm` concurrently? Explain briefly.

   (b) Here is a simpler algorithm for finding the second smallest element in a binary search tree in terms of some other operations:

```
synchronized E deleteMin() { ... }
synchronized E findMin() { ... }
synchronized void insert(E x) { ... }
E secondSmallestFinal() {
  E min = this.deleteMin();
  E ans = this.findMin();
  this.insert(min);
  return ans;
}
```

   Notice `deleteMin`, `findMin`, and `insert` are synchronized methods.

   i. Suppose two threads call `secondSmallestFinal` concurrently. Demonstrate how one of them can get the wrong answer.

   ii. Does the code above have any *data races*? Explain briefly.

   iii. What is the easiest way to fix `secondSmallestFinal`?

Solution:

(a) Yes, `secondSmallestMidterm` does no writes to shared memory, so multiple concurrent executions would not interfere. (Interleaving with other tree operations like `insert` would be a problem, but that is not the question here.)

(b) i. If thread 1 deletes the minimum element and then thread 2 runs `secondSmallestFinal`, it will actually get the 3rd smallest element (or raise an exception if there were only 2 elements to begin with).

ii. No, there is never unsynchronized access to the same field by two threads because all field access is by synchronized helper methods.

iii. Make it `synchronized` as well.

Name:_____

7. You are designing a new social-networking site to take over the world. To handle all the volume you expect, you want to support multiple threads with a fine-grained locking strategy in which each user's profile is protected with a different lock. At the core of your system is this simple class definition:

```
class UserProfile {
   static int id_counter;
   int id; // unique for each account
   int[] friends = new int[9999]; // horrible style
   int numFriends;
   Image[] embarrassingPhotos = new Image[9999];
   UserProfile() { // constructor for new profiles
     id = id_counter++;
     numFriends = 0;
   }
   synchronized void makeFriends(UserProfile newFriend) {
      synchronized(newFriend) {
        if(numFriends == friends.length
           || newFriend.numFriends == newFriend.friends.length)
          throw new TooManyFriendsException();
        friends[numFriends++] = newFriend.id;
        newFriend.friends[newFriend.numFriends++] = id;
      }
   }
   synchronized void removeFriend(UserProfile frenemy) {
      ...
   }
}
```

(a) The constructor has a concurrency error. What is it and how would you fix it? A short English anwser is enough – no code or details required.

(b) The `makeFriends` method has a concurrency error. What is it and how would you fix it? A short English anwser is enough – no code or details required.

(c) Rather than throwing an exception in `makeFriends` if an array is full, give two alternatives. Describe them only at a high level – a sentence or two is enough – without getting into any code details. One alternative should be easy and have nothing to do with concurrency. The other should involve concurrency.

Solution:

(a) There is a data race on `id_counter`. Two accounts could get the same `id` if they are created simultaneously by different threads. Or even stranger things could happen. You could synchronize on a lock for `id_counter` or mark `id_counter` `volatile`.

(b) There is a potential deadlock if there are two objects `obj1` and `obj2` and one thread calls `obj1.makeFriends(obj2)` when another thread calls `obj2.makeFriends(obj1)`. The fix is to acquire locks in a consistent order based on the `id` fields, which are unique.

(c) First, you could resize the array. Second, you could use *condition variables* to `wait` until there is room in the array and have `removeFriend` call `notifyAll`.

Note from instructor, not necessary for the exam: Getting this right is rather difficult since you need room in *both* arrays but it would be inefficient and deadlock-prone to wait on one object while still holding the lock for the other. Something like this pseudocode should work:

```
while(true) {
   synchronized(acct1) { if(acct1.friends is full) acct1.wait(); }
   synchronized(acct2) { if(acct2.friends is full) acct2.wait(); }
   synchronized(acct1) {
   synchronized(acct2) {
      if(acct1.friends is full || acct2.friends is full) continue;
      ... add to arrays ...
      break;
   }
 }
```

Name:_____

8. Note: This problem is doable before or after the previous problem. This problem does not involve concurrency.

   Consider again the `UserProfile` class from the previous problem. Suppose we have an array of type `UserProfile[]` that contains every user profile. Let n be the length of the array and m be the number of "friendships" in the entire system.

   (a) This array is essentially a representation of a graph.
      i. What are the nodes and what are the edges?
      ii. Is it more like an adjacency-list or an adjacency-matrix (e.g., in how efficiently one can perform graph operations on it)?

   (b) Suppose we want to implement a method `isConnected` that takes two numbers and determines if the accounts with these id numbers are connected by any sequence of friendships. What would be the asymptotic worst-case running time of an efficient algorithm for this problem?

   (c) Suppose you are looking over some profile information and you realize that almost all `isConnected` queries are for profiles that are actually very close to each other, meaning only a small number of friendships are needed to connect them. How should this information guide your choice for how to traverse the data such that your algorithm is more efficient in practice (even though it will not improve the asymptotic worst-case)? Be specific about how you would traverse the data.

   Solution:

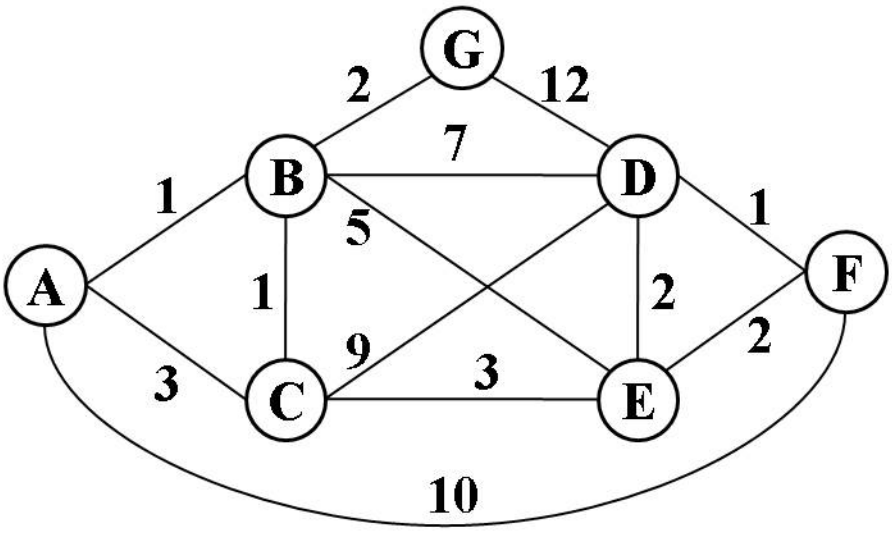
   (a) The nodes are profiles and the edges are friendships. The graph is undirected since the friend relationship is symmetric. It is more like an adjacency-list — it does not require $O(n^2)$ space and its running times for all operations are the same as for an adjacency list.

   (b) $O(m)$, i.e., proportional to the number of edges.

   (c) You should use a *breadth-first* traversal of the graph because it will visit all friends close to one of the ids first, which will usually suffice to answer the query.

Name:_____

9. Consider the following undirected, weighted graph (seen previously in the exam):



Apply Kruskal's algorithm to compute a minimum spanning tree (MST). In the designated spaces below, write down the edges in the order they are considered by the algorithm. If the edge is part of the MST found by the algorithm, write it down in the first list of edges that form the MST. Write down the other edges considered in the order you considered them. Assume that the algorithm terminates as soon as an MST has been found. Don't forget part (c).

(a) Edges that form part of the MST, in the order considered:

Solution:

In any order: (A,B), (B,C), (D,F) Then in either order: (B,G) and ((D,E) **xor** (E,F)) Then: (C,E)

(b) Other edges considered, but not included in the MST, in the order considered:

Solution:

((D,E) **xor** (E,F)) Then one can *optionally* list (A,C)

(c) Is there another MST in addition to the one you listed in (a)? Explain.

Solution:

Yes, we can add (D,E) or (E,F) to the solution but not both. Our choice connects D, E, and F but has no further impact on how the algorithm proceeds. We gave both answers above.

Name:_____

10. Suppose we have an undirected, connected, weighted graph with n vertices such that:

    - Every weight is an integer greater than zero.
    - No two edges have the same weight.

    (a) In terms of n, what is the lowest cost a minimum spanning tree (MST) could have for a graph with n nodes that meets the description above? (The cost of an MST for some n-node graphs meeting the description will be higher. The question here is what the lowest possible MST cost could be — you get to pick a graph that minimizes MST cost subject to the criteria.)

    Give a closed-form solution, which just means that, for full credit, your answer should not be in terms of a series.

    (b) Prove your answer to part (a) is correct for all $n \geq 1$.

    Solution:

    (a) $n(n+1)/2 - n$ or equivalently $n(n-1)/2$. This is the closed form of the series $\Sigma_{i=1}^{n-1} i$.

    (b) Any spanning tree for a graph with n nodes has $n-1$ edges. Given the criteria, the lowest weights these edges could have are $1, 2, ..., n-1$ so the lowest possible cost for an MST is $\Sigma_{i=1}^{n-1} i$. An example with this cost would be a list where the i^{th} edge has weight i. We prove by induction on n that $\Sigma_{i=1}^{n-1} i = n(n-1)/2$ for all $n \geq 1$.

    - Base case $n = 1$: The sum has zero terms, so it is 0. And $1(1-1)/2 = 0$.
    - Inductive case $n = k+1$: We need to show $\Sigma_{i=1}^{k+1-1} i = (k+1)k/2$. We know $\Sigma_{i=1}^{k+1-1} i = (\Sigma_{i=1}^{k-1} i) + k$. By induction, $\Sigma_{i=1}^{k-1} i = k(k-1)/2$. So $\Sigma_{i=1}^{k+1-1} i = k(k-1)/2 + k$. And $k(k-1)/2 + k = k^2/2 - k/2 + 2k/2 = k^2/2 + k/2 = (k+1)k/2$ which is what we need.