Name:_____

CSE332, Spring 2012, Final Examination
June 5, 2012

Please do not turn the page until the bell rings.

Rules:

- The exam is closed-book, closed-note.
- **Please stop promptly at 4:20.**
- You can rip apart the pages, but please staple them back together before you leave.
- There are **100 points** total, distributed **unevenly** among **11** questions (many with multiple parts).

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.** Make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. (**9** points) For each of the following situations, name the best sorting algorithm we studied. (For one or two questions, there may be more than one answer deserving full credit, but you only need to give one answer for each.)

   (a) The array is mostly sorted already (a few elements are in the wrong place).

   (b) You need an $O(n \log n)$ sort even in the worst case *and* you cannot use any extra space except for a few local variables.

   (c) The data to be sorted is too big to fit in memory, so most of it is on disk.

   (d) You have many data sets to sort *separately*, and each one has only around 10 elements.

   (e) You have a large data set, but all the data has only one of about 10 values for sorting purposes (e.g., the data is records of elementary-school students and the sort is by age in years).

   (f) Instead of sorting the entire data set, you only need the k smallest elements where k is an input to the algorithm but is likely to be much smaller than the size of the entire data set.

   Solution:

   (a) insertion sort

   (b) heap sort

   (c) merge sort

   (d) insertion sort (or selection sort is probably okay too)

   (e) bucket sort

   (f) selection sort is the simplest most natural choice and fine if k is truly small – this is the expected answer. If k is not a small constant, we might prefer heap sort or a variant of quicksort with a cut-off like we used on a homework problem. So full credit for any of these answers.

Name:_____

2. (**9** points) Recall that the comparison-sorting problem is $\Omega(n \log n)$. This problem instead considers the problem of finding the *median* value of a collection assuming elements can only be compared (resolving ties for the median arbitrarily).

   (a) Consider this *bad argument* for showing that the median-finding problem is $\Omega(n \log n)$:

   > We can find the median of n elements by putting all elements in an array `arr`, sorting the array, and returning the middle element (`arr[n/2]` if n is odd). Since the sorting step is $\Omega(n \log n)$, the median-finding problem is $\Omega(n \log n)$.

   Explain what is *wrong* with this argument in 1-2 English sentences.

   (b) Give a simple argument that the median-finding problem is $\Omega(n)$.

   (c) Can we or can we not conclude from your answers to (a) and (b) that the median-finding problem is $\Omega(n \log n)$? *Explain your answer.*

Solution:

(a) This argument does not cover the possibility that there is another algorithm for median-finding that is faster than $\Omega(n \log n)$. Such an algorithm just cannot use a comparison sort for the n items.

(b) If the problem is not $\Omega(n)$, then there is an algorithm faster than linear. But such an algorithm cannot compare every element since doing so would take at least n comparisons. So there is some element never compared. Since this element might or not be the median and the algorithm cannot tell, no such algorithm can exist.

(c) We cannot conclude this: there may be a proof that the problem is $\Omega(n \log n)$ even though the argument in part (a) is not it.

(It turns out there is no proof that median-finding is $\Omega(n \log n)$ because the claim is false. An $O(n)$ algorithm exists. A nice description is at `http://en.wikipedia.org/wiki/Selection_algorithm`.)

Name:_____

3. (**10** points) In class we studied algorithms DFS for depth-first traversal of a graph and BFS for breadth-first traversal of a graph. Also recall $\Theta(f(n))$ just means $O(f(n))$ and $\Omega(f(n))$.

   Although DFS and BFS are for arbitrary graphs, we can use them for traversals starting from the root of a large, rooted binary tree with n nodes.

   (a) Use a picture to describe such a binary tree for which DFS from the root needs $\Theta(n)$ extra *space*.

   (b) Use a picture to describe such a binary tree for which BFS from the root needs $\Theta(n)$ extra *space*.

   (c) Use a picture to describe such a binary tree for which DFS from the root needs $\Theta(1)$ extra *space*.

   (d) Use a picture to describe such a binary tree for which BFS from the root needs $\Theta(1)$ extra *space*.

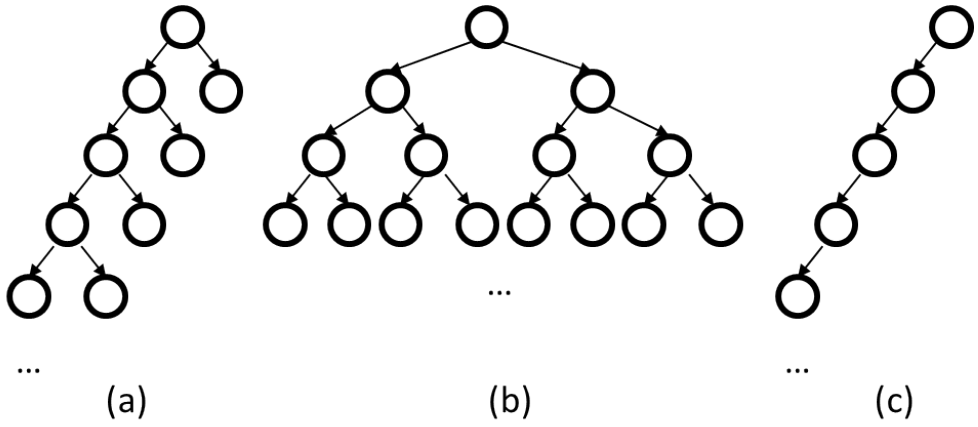
   Solution:
   There are various possible answers for each, but it is important the tree not have too many nodes for the bounds to hold.

   (a) See (a) below. We need a tall tree where we push two nodes as we descend each level of the tree. The tree must be unbalanced — a balanced tree would need only a stack of logarithmic size.

   (b) See (b) below. We need a balanced tree. For a complete tree, for example, when the traversal gets to the last full level of the tree, all the nodes in this level are in the tree and there are $O(n)$ such nodes (between roughly 1/4 and 1/2 of all the nodes).

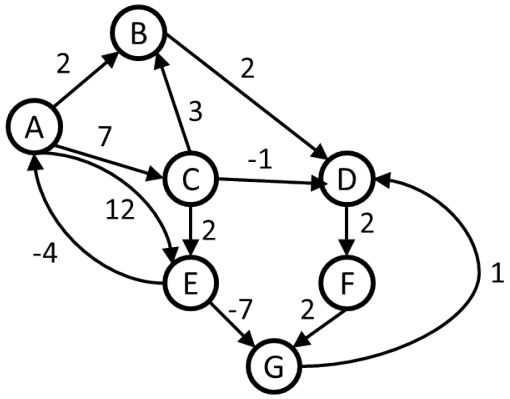
   (c) See (c) below: A binary tree does not require two children at internal nodes, so the simplest answer is a tree with one leaf despite many internal nodes.

   (d) The answer to (a) and the answer to (c) work fine, as done any graph where the number of nodes at any height in the tree is a constant.



Name:_____

4. (**10** points) Consider the following directed, weighted graph:



(a) Even though the graph has negative weight edges, step through Dijkstra's algorithm to calculate *supposedly* 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.

Solution:

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

Vertex	Known	Distance	Path
A	Y	0	
B	Y	2	A
C	Y	7	A
D	Y	4	B
E	Y	12 9	A C
F	Y	6	D
G	Y	8	F

(b) Dijkstra's algorithm found the wrong path to some of the vertices. For just the vertices where the wrong path was computed, indicate *both* the path that was computed and the correct path.

(c) What *single* edge could be removed from the graph such that Dijkstra's algorithm would happen to compute correct answers for all vertices in the remaining graph?

Solution:

(b) Computed path to G is A,B,D,F,G but shortest path is A,C,E,G.
Computed path to D is A,B,D but shortest path is A,C,E,G,D.
Computed path to F is A,B,D,F but shortest path is A,C,E,G,D,F.

(c) The edge from E to G.

Name:_____

5. (**7** points) Suppose we define a different kind of graph where we have weights on the vertices and not the edges.

   (a) Does the *single-source shortest-paths problem* make sense for this kind of graph? If so, give a precise and formal description of the *problem*. If not, explain why not. Note we are not asking for an algorithm, just what the problem is or that it makes no sense.

   (b) Does the *minimum spanning tree problem* make sense for this kind of graph? If so, give a precise and formal description of the *problem*. If not, explain why not. Note we are not asking for an algorithm, just what the problem is or that it makes no sense.

   Solution:

   (a) Yes, this problem makes sense: Given a starting vertex v find the lowest-cost path from v to every other vertex. The cost of a path is the sum of the weights of the vertices on the path.

   (b) No, this problem does not make sense: Any spanning tree would include every vertex, so they would all have the same weight. (Whether or not a spanning tree exists is just the question of whether the graph is connected and has nothing to do with the weight of vertices.)

   On the other hand, we gave full credit for an answer of "yes" that explained the cost of the tree to be the cost of its edges where an edge cost is the sum of its two nodes' costs. Equivalently, the spanning tree cost counts each internal node twice and each leaf once. Under this interpretation, the problem makes sense.

Name:_____

6. (**12** points) In Java using the ForkJoin Framework, write code to solve the following problem:

   - Input: An `int[]` (though the element type happens to be irrelevant)
   - Output: A new `int[]` where the elements are in the reverse order. For example, the element in the last array index of the input will be at index 0 in the output.

   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 array element.

   We have provided some of the code for you.

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class Main {
    static final ForkJoinPool fjPool = new ForkJoinPool();
    static int[] reverse(int[] array) {
        // ADD A LINE HERE
        fjPool.invoke(new Reverse(answer,array,0,array.length)); // DO NOT CHANGE
        // ADD A LINE HERE
    }
}

// DEFINE A CLASS HERE
```

   Solution:

```
    static int[] reverse(int[] array) {
        int [] answer = new int[array.length];
        fjPool.invoke(new Reverse(answer,array,0,array.length));
        return answer;
    }
class Reverse extends RecursiveAction {
    int[] out;
    int[] in;
    int lo;
    int hi;
    Reverse(int[] o, int[] i, int l, int h) {
        out = o; in = i; lo = l; hi = h;
    }
    public void compute() {
        if(hi==lo+1) {
            out[in.length-lo-1] = in[lo];
        } else {
            Reverse left  = new Reverse(out,in,lo,(hi+lo)/2);
            Reverse right = new Reverse(out,in,(hi+lo)/2,hi);
            left.fork();
            right.compute();
            left.join();
        }
    }
}
```

Name:_____

7. (**7** points) Suppose a program uses your solution to the previous problem to reverse an array containing 2^{27} elements.

   (a) In English, about how big is 2^{27}? For example, "one thousand" is an answer in the right form, but is the wrong answer.

   (b) When the program executes, how many fork-join threads will your solution to the previous problem create? Give both an exact answer and an approximate English answer.

   (c) Suppose you modify your solution to use a sequential cut-off of 1000. When this modified program executes, how many fork-join threads will it create? Give both an exact answer and an approximate English answer.

   Solution:

   (a) one hundred million (or one hundred twenty-eight million)

   (b) 2^{27}, one hundred million (If your answer to the previous question calls `right.fork` instead of `right.compute`, then the answer is $2^{28} - 1$, which is about two hundred fifty (six) million.)

   (c) 2^{18}, two hundred fifty (six) thousand (Note that most answers were twice as small, diving by 2^{10} instead of 2^9. Since 1000 is a bit less than 2^{10}, the code will solve problems of size 512 sequentially. This "off by 2x" error was worth only one point.)

Name:_____

8. (**10** points) In class we learned an algorithm for parallel prefix sum with $O(n)$ work and $O(\log n)$ span. In this problem, you will develop similar algorithms for parallel *suffix* sum, where element `i` of the output array holds the sum of the elements in indices greater than or equal to `i` in the input.

   (a) Describe in English or high-level pseudocode a two-pass algorithm to solve the parallel suffix sum problem. Assume the reader is familiar with the parallel prefix sum problem, so for any parts that are *exactly* the same, you can just say they are the same. Any parts that are different need a full explanation.

   (b) Suppose you needed an implementation of parallel suffix sum, but you already had available a library for parallel prefix sum. Describe an easier-to-implement algorithm for parallel suffix sum that uses parallel prefix sum and one or more other algorithms as subroutines while maintaining $O(n)$ work and $O(\log n)$ span.

   Solution:

   (a) Use an "up pass" and a "down pass" just like for parallel prefix. The "up pass" is identical, creating a binary tree with the sum for a range of the array at each node. For the "down pass" we pass a "fromRight" argument to each node. For the root, this value is 0. An internal node passes this value to its children as follows: The right child gets the same fromRight value and the left child gets the sum of the same fromRight value and the stored sum at the right child. At the leaf for range [lo,hi+1) (a one-element range), the fromRight value is the output for index lo.

   (b) Use three steps. First, reverse the array using your solution to problem 4. Second, perform parallel prefix sum on this array. Third, reverse the output of the second step and use this as the result. Each step is $O(n)$ work and $O(\log n)$ span, so the overall asymptotic work and span is the same.

   There is an alternate unanticipated solution for which we gave full credit if explained well: First do parallel prefix sum. Now perform a parallel map over the result where we take the total sum of the array and subtract from it the value in the array from the prefix sum (and then add back the value in the original input to produce an inclusive suffix sum). Note the total sum can be computed via its own reduce operation or just as the rightmost value from the prefix sum. Also note that for the map step, we can produce an inclusive sum by subtracting just the item one to the left (with a special case for index 0).

Name:_____

9. (**9** points) Answer "always" or "sometimes" or "never" for each of the following.

   (a) A program with data races also has bad interleavings.

   (b) A program with bad interleavings also has data races.

   (c) A program that correctly uses consistent locking (all thread-shared data is associated with a lock that is always held when accessing the data) has data races.

   (d) A program that correctly uses consistent locking (all thread-shared data is associated with a lock that is always held when accessing the data) has bad interleavings.

   (e) Java's `synchronized` statement will block if the thread executing it already holds the lock that is being acquired.

   (f) This code snippet using a condition variable `foo` is a bug: `if(someCondition()) foo.wait()`.

Solution:

(a) Sometimes

(b) Sometimes

(c) Never

(d) Sometimes

(e) Never

(f) Always

Name:_____

10. (**12** points) Consider an adjacency-list representation of an undirected graph where if edge (u, v) is in the graph then v is in u's adjacency list and u is in v's adjacency list. (This supports efficient operations for "find all nodes adjacent to a given node.") Suppose we wish to support multiple threads accessing the graph concurrently such that every thread always sees a consistent state of the graph. Further suppose we use a locking strategy where each adjacency list uses a different reentrant lock (one lock for the whole adjacency list).

    (a) This pseudocode for deleting an edge from the graph — and doing nothing if the edge is not already present — has a concurrency error. Write down an interleaving that exhibits a concurrency error and describe what happens when the interleaving occurs. Assume `array` holds the adjacency lists, vertices are represented by ints, and `contains` and `delete` are methods on lists (the latter throwing an exception if the item is not present).

```
void deleteEdge(int u, int v) {
  synchronized(array[u]) {
    if(!array[u].contains(v))
      return;
  }
  synchronized(array[u]) { array[u].delete(v); }
  synchronized(array[v]) { array[v].delete(u); }
}
```

    (b) This pseudocode also has a concurrency error. Repeat the previous problem with this code:

```
void deleteEdge(int u, int v) {
  synchronized(array[u]) {
    synchronized(array[v]) {
       if(!array[u].contains(v))
         return;
       array[u].delete(v);
       array[v].delete(u);
    }
  }
}
```

    (c) Write a correct version of `deleteEdge` in the same style as the broken versions above. For simplicity, you can assume no other operations modify the graph (or that they are written using the same approach as your method).

Solution:
See next page.

Name:_____

This page left blank so you have more room for problem 9 if necessary.

Solution:

(a) (Other answers are possible.) Here a bad interleaving causes an exception because we try to remove an edge that has already been removed. The exception occurs when Thread 1 does `array[u].delete(v)`.

```
Thread 1 (deleteEdge(u,v))          Thread 2 (deleteEdge(u,v))
--------------------------          --------------------------
  synchronized(array[u]) {
    if(!array[u].contains(v))
      return;
  }
                                       all of deleteEdge(u,v)
  synchronized(array[u]) {
    array[u].delete(v);
  }
```

(b) Here a bad interleaving causes a deadlock, so neither thread will run again and no other threads can get the two locks for these adjacency lists.

```
Thread 1 (deleteEdge(u,v))          Thread 2 (deleteEdge(v,u))
--------------------------          --------------------------
synchronized(array[u]) {            synchronized(array[v]) {
  synchronized(array[v]) { // blocks  synchronized(array[u]) { // blocks
```

(c)
```
void deleteEdge(int u, int v) {
  int a = u < v ? u : v;
  int b = u > v ? u : v;
  synchronized(array[a]) {
    synchronized(array[b]) {
       if(!array[a].contains(b))
         return;
       array[a].delete(b);
       array[b].delete(a);
    }
  }
}
```

Name:_____

11. (**5** points) Consider using a simple linked list as a dictionary. Assume the client will never provide duplicate elements, so we can just insert elements at the beginning of the list. Now assume the peculiar situation that the client may perform any number of insert operations but will only ever perform at most one lookup operation.

    (a) What is the worst-case running-time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

    (b) What is the worst-case amortized running-time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

    Solution:

    (a) inserts are $O(1)$ (push on the front of a linked list), but the lookup is $O(n)$ where n is the number of inserted items since the lookup may be last and be for one of the earliest inserted items

    (b) amortized all operations are now $O(1)$. Inserts are still $O(1)$. And the lookup can take at most time $O(n)$ where n is the number of previously inserted items. So the total cost of any n operations is at most $O(n)$, which is amortized $O(1)$.