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.

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.*

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*.

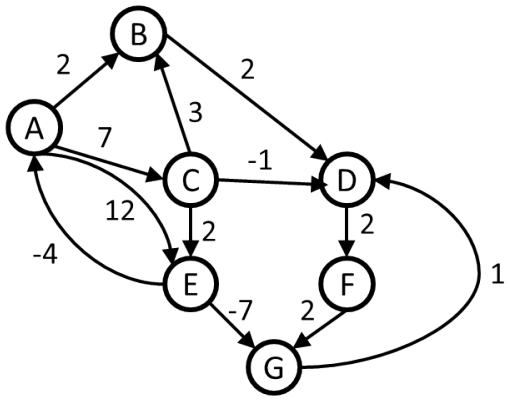
   (b) Use a picture to describe such a binary tree for which BFS from the root needs $\Theta(n)$ extra *space*.

   (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*.

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.

**Known vertices (in order marked known):** ___ ___ ___ ___ ___ ___ ___

| Vertex | Known | Distance | Path |
|---|---|---|---|
| A | | | |
| B | | | |
| C | | | |
| D | | | |
| E | | | |
| F | | | |
| G | | | |

(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?

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.

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
```

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.

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.

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()`.

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).

Name:_____

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

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.