CSE 373, Winter 2013
Final Exam, Tuesday, March 19, 2012

Name: ______________________________

Quiz Section: ____________ **TA/Grader:** ____________

Student ID #: ____________

Rules:

- You have **110 minutes** to complete this exam.
  You may receive a deduction if you keep working after the instructor calls for papers.
- This test is open-book/notes. You may use any paper resources or textbooks you like.
- You may *not* use any computing devices, including calculators, cell phones, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- Please do not abbreviate code, such as writing ditto marks ("") or dot-dot-dot marks (...).
- If you enter the room, you must turn in an exam and will not be permitted to leave without doing so.
- You must show your **Student ID** to a TA or instructor for your submitted exam to be accepted.

Good luck! You can do it!

Problem	Description	Earned	Max
1	Big-Oh		10
2	Sort Tracing		10
3	Sorting Algorithm Selection		10
4	Sort Algorithm Implementation		15
5	Graph Properties		15
6	Graph Paths		15
7	Graph Implementation		15
8	Parallelism / Concurrency		10
TOTAL	**Total Points**		**100**

1. Big-Oh

Give a tight bound of the runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of the variable *N*. Write your answer on the right side.

Question	Answer
a)	O(____________)

```
Map<Integer, String> map =
        new TreeMap<Integer, String>();

for (int i = 1; i <= N; i++) {
    map.put(i, "foo");
    if (map.containsKey(i / 100)) {
        map.remove(i - 1);
        map.put(i, "bar");
    }
}
```

Question	Answer
b)	O(____________)

```
List<String> list = new ArrayList<String>();
for (int i = N; i >= 1; i--) {
    for (int j = 1; j <= i; j++) {
        if (!list.contains(i * j)) {
            list.add(i * j);
        }
    }
}
for (int j = 1; j <= N; j++) {
    list.add(j);
}
System.out.println(list);
```

Question	Answer
c)	O(____________)

```
Map<Integer, Integer> map =
        new HashMap<Integer, Integer>();
for (int i = 1; i <= N; i++) {
    map.put(i, i+1);
}

int sum = 0;
for (int i : map.keySet()) {
    Set<Integer> copy =
            new HashSet<Integer>(map.values());
    if (copy.contains(i * 2)) {
        sum++;
    }
}
System.out.println(sum);
```

2. Sort Tracing

a)
```
// index  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
       { 38, 36, 51, 45, 66, 58, 53,  9, 90, 91, 85, 48, 13, 40, 58, 45, 23}
```
Trace the execution of the **shell sort** algorithm over the array above. Use gaps of *N*/2, *N*/4, ..., 2, 1. Show each pass of the algorithm and the state of the array after the pass has been performed, until the array is sorted. Please clearly label each intermediate array so we know what gap is being used at each step.

2. Sort Tracing

b)
```
// index  0   1   2   3   4   5   6   7   8   9  10
       { 16, 21, 45,  8, 11, 53,  3, 26, 49, 31, 12}
```
Trace the execution of the **quick sort** algorithm over the array above, using the *first* element as the pivot. Show each pass of the algorithm, with the pivot selection and partitioning, and the state of the array as/after the partition is performed, until the array is sorted. You do not need to show details of partitioning calls over ranges of only 1 or 2 elements.

3. Sorting Algorithm Selection

For each of the following situations, choose the sorting algorithm we studied that will perform the best. Choose one of: bogo, stooge, bubble, selection, insertion, shell, heap, merge, quick, or bucket sort. (For some questions, more than one answer might be acceptable, but you only need to list one.)

a) I am sorting data that is stored over a network connection. Based on the properties of that connection, it is extremely expensive to "swap" two elements. But looping over the elements and looking at their values is very inexpensive. I want to minimize swaps above all other factors.

A good choice would be: ________________________ sort.

b) I have a fast computer with many processors and lots of memory. I want to choose a sorting algorithm that is fast and can also be parallelized easily to use all of the processors to help sort the data.

A good choice would be: ________________________ sort.

c) I have an array that is already sorted. Periodically, some new data comes in and is added to the array at random indexes, messing up the ordering. I need to re-sort the array to get it back to being fully ordered. I do not want to use very much additional memory during the sort.

A good choice would be: ________________________ sort.

d) We are working on an unstable system. When we try to run our sort on large arrays, sometimes the sorting algorithm will be forced to stop running in the middle of execution. (The array will be left in whatever state it was in at the time the algorithm stopped.) But we can start the algorithm over and run it again later. So we want to choose an algorithm that will make progress toward the goal even if it does not always finish executing.

A good choice would be: ________________________ sort.

4. Sorting Algorithm / Guava Collection Programming

Write a method named **`guavaSort`** that accepts an array of strings as a parameter and arranges the strings in the array into sorted ascending order. Specifically, your sorting algorithm should use a Guava `Multiset` or `Multimap` to implement a variation of the bucket sort algorithm that will work on strings. Use the Guava collection to count occurrences of strings, similarly to what is done in bucket sort, and then place those strings back into the array in sorted order. For example, suppose your method is passed the following array:

```
[Farm, Zoo, Car, Apple, Bee, Golf, Bee, Dog, Golf, Zoo, Zoo, Bee, Bee, Apple]
```

Your collection should store the following occurrences of the strings:

```
[Apple x 2, Bee x 4, Car, Dog, Farm, Golf x 2, Zoo x 3]
```

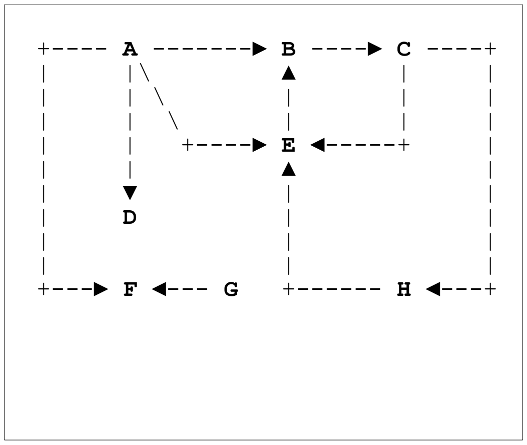
Which you should use to put the strings back into the array in sorted order:

```
[Apple, Apple, Bee, Bee, Bee, Bee, Car, Dog, Farm, Golf, Golf, Zoo, Zoo, Zoo]
```

Your code should run in O(*N* log *N*) time and use O(*N*) memory, where *N* is the number of elements in the array.

You may assume that the array passed, and all of the strings in it, are not `null`.
Do not construct any other auxiliary collections other than the single `Multiset` or `Multimap`.

5. Graph Properties
Parts a) - c) refer to the graph shown at right.



a) **Circle the choice** on each line that correctly describes this graph; circle one of the two choices on each of the four lines. You do not need to explain your answer.

- This graph is: **[directed ,** **undirected]**.

- This graph is: **[weighted ,** **unweighted]**.

- This graph is: **[connected ,** **unconnected]**.

- This graph is: **[cyclic ,** **acyclic]**.

b) The vertex with the **largest in-degree** is ___________, which has an in-degree of _________.

The vertex with the **largest out-degree** is ___________, which has an out-degree of _________.

c) Write a complete **adjacency matrix** representation of the graph below. (Be mindful of the orientation of the matrix; do not mix up the meaning of a row vs. a column if you want to receive full credit.)

6. Graph Paths

For the graph shown at right, answer the following questions:

```
           4          3
 +---- A ----► B ----► C ◄---+
 |     |       |       |     |
 |    5|       |7      |1    |2
 |     ▼       ▼       |     |
 |     D ----► E ◄-----+     F
3|     |   5   |\            |
 |     |       | \   8       |
 |     |       |  +----+     |6
 |    1|      1|       |     |
 |     |       |       |     |
 |     ▼       ▼       ▼     |
 +---► G ----► H ----► I ◄---+
           9       2
```

a) Write the **order** in which a *breadth-first search* (BFS) would mark the vertices as visited when searching for a path from vertex **B** to vertex **I**.

Then write the **path** that BFS would return from B to I. Assume that any for-each loop over neighbors returns them in ABC order.

Vertex Visit Order: ______________________________

Path Returned: ______________________________

b) In the table below, write the **cost and previous** values that *Dijkstra's algorithm* would compute when searching for the minimum-weight path from vertex A to all other vertices. Also write the **path** that the algorithm would reconstruct from vertex A to vertex I (the vertices of the path, not the edges) and give the total cost of that path. *(You may want to walk through Dijkstra's algorithm on your own paper to make sure you have the correct answers, but you are not required to show your work to get full credit.)*

Vertex	Cost	Previous
A		
B		
C		
D		
E		
F		
G		
H		
I		

Path from A to C: ______________________________, **Cost:** __________

c) Write a *topological sort* ordering for the vertices in the graph. Any valid topological sort ordering is considered correct. If it is not possible to produce a topological sort of the graph, write "There is no valid topological sort" and explain why using specific properties of the graph. *(You may want to walk through the topological sort algorithm on your own paper to make sure you have the correct answers, but you are not required to show your work to get full credit.)*

Topological Sort Ordering: ______________________________

7. Graph Implementation

Write a method named **`topologicalSort`** that could be added to the `SearchableGraph` class you wrote in Homework 8, that returns a `List` of vertices representing a topological sort of the graph. (Note that some of the graph programming problems on the practice exams involved using the `Graph` class externally, from the client's perspective; this problem is asking you to add a method to `SearchableGraph` that would be placed inside the graph itself, inside the `SearchableGraph` class, like you did on your homework.)

Recall that a topological sort ordering is an arrangement of the vertices such that for every directed edge (*v*, *w*), vertex *v* appears before vertex *w* in the list. So for the graph in the diagram below, one valid order to return would be the list `[B, A, C, D, E, F]`. Another valid order would be `[A, B, D, C, F, E]`. The exact order of the list you return does not matter as long as it satisfies the topological sort property described previously. If the graph does not have any valid topological sort orderings, your method should return `null`. If the graph is empty, return an empty list. If the graph contains only a single vertex, return a list containing only that vertex.

```
       B ---> C -----+
       |      |      |
       |      |      |
       |      ▼      ▼
       |      E      F
       |      ▲      ▲
       |      |      |
       ▼      |      |
A ---► D -----+      |
       |             |
       +-------------+
```

You should not modify the contents of the graph (such as by adding or removing vertices or edges from the graph). A solution that does so can receive partial credit but will receive a significant deduction. You may assume that the graph and its vertices are not `null`. You may construct auxiliary collections as needed to solve this problem.

Recall that the `SearchableGraph` class has the following methods that you may call as needed to solve the problem:

```
public void addEdge(V v1, V v2)
public void addEdge(V v1, V v2, E e)
public void addEdge(V v1, V v2, int weight)
public void addEdge(V v1, V v2, E e, int weight)
public void addVertex(V v)                  public boolean isEmpty()
public void clear()                         public boolean isReachable(V v1, V v2)
public void clearEdges()                    public boolean isWeighted()
public boolean containsEdge(E e)            public List<V> minimumWeightPath(V v1, V v2)
public boolean containsEdge(V v1, V v2)     public Set<V> neighbors(V v)
public boolean containsVertex(V v)          public int outDegree(V v)
public int cost(List<V> path)               public void removeEdge(E e)
public int degree(V v)                      public void removeEdge(V v1, V v2)
public E edge(V v1, V v2)                   public void removeVertex(V v)
public int edgeCount()                      public List<V> shortestPath(V v1, V v2)
public Collection<E> edges()                public String toString()
public int edgeWeight(V v1, V v2)           public String toStringDetailed()
public int inDegree(V v)                    public int vertexCount()
public boolean isDirected()                 public Set<V> vertices()
```

Write your answer on the next page.

7. Graph Implementation (writing space)

8. Parallelism / Concurrency

Suppose a `Stack<E>` class has been written with the following methods:

- `void push(E)` (add to top of stack)
- `E pop()` (remove and return element at top of stack)
- `E peek()` (return element at top of stack without removing it)
- `boolean isEmpty()` (return `true` if the stack does not contain any elements)

Suppose that the `peek` method has been implemented inside the stack class in the following way:

```
    // Returns the element on top of this stack without changing the stack's state.
    // If the stack is empty, throws an IllegalArgumentException.
 1  public E peek() {
 3      if (this.isEmpty()) {
 4          throw new NoSuchElementException();
 5      } else {
 6          E topElement = this.pop();
 7          this.push(topElement);
 8          return topElement;
 9      }
10  }
```

The method promises to always return the top element on the stack if it is non-empty, and promises not to modify the state of the stack (from the client's perspective). If a stack is used concurrently by two or more threads, is there an order of execution that will violate one or both of these promises? If so, state which of the two promises can be broken, and give an example execution order that breaks the promise(s). In your example, describe the stack's state (elements, if any) and the order in which the lines would execute by the two threads in order to cause the problem (give lists of ranges of numbers of lines that would execute by each thread).