CSE 373, Winter 2013, Final Exam
ANSWER KEY

1. Big-Oh Analysis

a) O($N \log N$)
b) O(N^3)
c) O(N^2)

2. Sort Tracing

a) shell sort

```
           0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
original [38, 36, 51, 45, 66, 58, 53,  9, 90, 91, 85, 48, 13, 40, 58, 45, 23]
gap 8    [23, 36, 51, 45, 13, 40, 53,  9, 38, 91, 85, 48, 66, 58, 58, 45, 90]
gap 4    [13, 36, 51,  9, 23, 40, 53, 45, 38, 58, 58, 45, 66, 91, 85, 48, 90]
gap 2    [13,  9, 23, 36, 38, 40, 51, 45, 53, 45, 58, 48, 66, 58, 85, 91, 90]
gap 1    [ 9, 13, 23, 36, 38, 40, 45, 45, 48, 51, 53, 58, 58, 66, 85, 90, 91]
```

b) quick sort

```
           0   1   2   3   4   5   6   7   8   9
original [31, 23, 56, 10, 60,  8, 17, 12, 70, 49]
          49                                  31    swap pivot to end
          12      17       8  60  56  49            partition
                              31              60    swap pivot back in

          12, 23, 17, 10,  8
           8              12    swap pivot to end
              10  17  23        partition
                  12      17    swap pivot back in

           8, 10

                      23, 17
                      17, 23

                                  56, 49, 70, 60
                                  60, 49, 70, 56   pivot to end
                                  49  60           partition
                                      56      60   swap pivot back in
                                  49, 56, 70, 60
                                  49
                                          70, 60
                                          60, 70

           8, 10, 12, 17, 23, 31, 49, 56, 60, 70
```

3. Sorting Algorithm Identification

a) selection sort, or merge sort

b) merge sort, or quick sort

c) insertion sort

d) insertion sort, or shell sort

4. Sort Implementation / Collection Programming

```
// common version using for-each loop
public static void guavaSort(String[] a) {
    Multiset<String> mset = TreeMultiset.create();   // array -> multiset
    for (String s : a) {
        mset.add(s);
    }

    int i = 0;                                       // multiset -> array
    for (String s : mset) {
        a[i] = s;
        i++;
    }
}

// version using elementSet and count
public static void guavaSort(String[] a) {
    Multiset<String> mset = TreeMultiset.create();   // array -> multiset
    for (String s : a) {
        mset.add(s);
    }

    int i = 0;                                       // multiset -> array
    for (String s : mset.elementSet()) {
        int count = mset.count(s);
        for (int j = 0; j < count; j++) {
            a[i] = s;
            i++;
        }
    }
}

// Iterator solution
public static void guavaSort(String[] a) {
    Multiset<String> mset = TreeMultiset.create();   // array -> multiset
    for (int i = 0; i < a.length; i++) {
        mset.add(a[i]);
    }

    int index = 0;                                   // multiset -> array
    Iterator<String> itr = mset.iterator();
    while (itr.hasNext()) {
        String s = itr.next();
        a[index] = s;
        index++;
    }
}

// short ninja version
public static void guavaSort(String[] a) {
    int i = 0;
    for (String s : TreeMultiset.create(Arrays.asList(a))) {
        a[i++] = s;
    }
}
```

5. Graph Properties

a)

directed, undirected
weighted, ***unweighted***
connected, ***unconnected***
cyclic, acyclic

b)

The vertex with the largest in-degree is **E**, which has an in-degree of **3**.
The vertex with the largest out-degree is **A**, which has an out-degree of **4**.

c) adjacency matrix

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A		1		1	1	1		
B			1					
C					1			1
D								
E		1						
F								
G						1		
H					1			

6. Graph Paths

a)

BFS marks in this order: B, C, E, H, I.
BFS(B, H) returns: `[B, E, I]`

b) Dijkstra's

Vertex	Cost	Previous
A	0	/
B	4	A
C	7	B
D	5	A
E	8	C
F	infinity	/
G	3	A
H	9	E
I	11	H

path from A to I: `[A, B, C, E, H, I]`, cost = `11`

c) topological sort ordering, any of the following:

[A, B, D, F, C, E, G, H, I]
[A, B, D, F, C, G, E, H, I]
[A, B, D, F, G, C, E, H, I]
[A, B, D, G, F, C, E, H, I]
[A, B, F, C, D, E, G, H, I]
[A, B, F, C, D, G, E, H, I]
[A, B, F, D, C, E, G, H, I]
[A, B, F, D, C, G, E, H, I]
[A, B, F, D, G, C, E, H, I]
[A, D, B, F, C, E, G, H, I]
[A, D, B, F, C, G, E, H, I]
[A, D, B, F, G, C, E, H, I]
[A, D, B, G, F, C, E, H, I]
[A, D, F, B, C, E, G, H, I]
[A, D, F, B, C, G, E, H, I]
[A, D, F, B, G, C, E, H, I]
[A, D, F, G, B, C, E, H, I]
[A, D, G, B, F, C, E, H, I]
[A, D, G, F, B, C, E, H, I]
[A, F, B, C, D, E, G, H, I]
[A, F, B, C, D, G, E, H, I]
[A, F, B, D, C, E, G, H, I]
[A, F, B, D, C, G, E, H, I]
[A, F, B, D, G, C, E, H, I]
[A, F, D, B, C, E, G, H, I]
[A, F, D, B, C, G, E, H, I]
[A, F, D, B, G, C, E, H, I]
[A, F, D, G, B, C, E, H, I]
[F, A, B, C, D, E, G, H, I]
[F, A, B, C, D, G, E, H, I]
[F, A, B, D, C, E, G, H, I]
[F, A, B, D, C, G, E, H, I]
[F, A, B, D, G, C, E, H, I]
[F, A, D, B, C, E, G, H, I]
[F, A, D, B, C, G, E, H, I]
[F, A, D, B, G, C, E, H, I]
[F, A, D, G, B, C, E, H, I]

7. Graph Implementation

```
// solution that closely follows the pseudocode from lecture slides
public List<V> topologicalSort() {
    // set up initial collections:
    Map<V, Integer> map = new HashMap<V, Integer>();   // Map of (vertex -> indegree)
    Queue<V> queue = new LinkedList<V>();
    List<V> out = new ArrayList<V>();                  // LinkedList also okay

    // initialization of data structures
    for (V v : vertices()) {
        int degree = inDegree(v);
        map.put(v, degree);         // initialize map of (vertex -> indegree)
        if (degree == 0) {
            queue.add(v);           // initialize queue with 0-indegree vertices
        }
    }

    // repeatedly "remove" vertices with in-degree 0 and their neighboring edges
    while (!queue.isEmpty()) {
        V v = queue.remove();
        out.add(v);
        for (V n : neighbors(v)) {
            int degree = map.get(n);
            if (degree > 1) {
                map.put(n, degree - 1);   // decrease degree by 1
            } else {
                queue.add(n);             // in-degree is 0; add to queue to process
            }
        }
    }

    if (out.size() == vertexCount()) {
        return out;
    } else {
        return null;   // not every vertex was reached, so no sort was found
    }
}


// solution that uses vertexInfo instead of map/queue  (slower but still full credit)
public List<V> topologicalSort() {
    clearVertexInfo();
    for (V v : vertices()) {
        vertexInfo(v).setCost(inDegree(v));
    }

    List<V> out = new ArrayList<V>();
    while (true) {
        boolean changed = false;
        for (V v : vertices()) {
            if (!vertexInfo(v).isVisited() && vertexInfo(v).cost() == 0) {
                vertexInfo(v).setVisited();
                changed = true;
                out.add(v);
                for (V n : neighbors(v)) {
                    int ncost = vertexInfo(n).cost();
                    if (ncost > 0) {
                        vertexInfo(n).setCost(ncost - 1);
                    }
                }
            }
        }
        if (!changed) {break;}
    }

    if (out.size() == vertexCount()) {
        return out;
    } else {
        return null;
    }
}
```

8. Parallelism / Concurrency

Here is an example order of execution for 2 threads that breaks the state of the stack:

```
Stack state:  bottom  [a, b, c]  top
Thread 1:  String s1 = stack.peek();
Thread 2:  String s2 = stack.peek();
```

Neither thread modifies the stack, so both should receive "c" from their calls to `peek`.

```
    // 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() {
 2      if (this.isEmpty()) {
 3          throw new NoSuchElementException();
 4      } else {
 5          E topElement = this.pop();
 6          this.push(topElement);
 7          return topElement;
 8      }
 9  }
```

Here is an execution order that causes incorrect behavior:

- Thread 1 runs lines 1-5. It grabs `"c"` as the `topElement`. Stack is now [a, b].
- Thread 2 runs lines 1-5. It grabs `"b"` as the `topElement`. Stack is now [a].
- Thread 1 runs lines 6-9. It pushes `"c"` back onto the stack and returns. Stack is now [a, c].
- Thread 2 runs lines 6-9. It pushes `"b"` back onto the stack and returns. Stack is now [a, c, b].

This example violates the first promise; the stack state is changed to [a, c, b]. It also violates the second promise; Thread 2's peek() call returns "b", though it should return "c".

Another incorrect behavior can result if the stack contains just a single element, such as `[a]`. The following execution order breaks the behavior:

- Thread 1 runs lines 1-4. It checks that the stack is not empty and therefore enters the `else` branch.
- Thread 2 runs lines 1-4. It checks that the stack is not empty and therefore enters the `else` branch.
- Thread 1 runs line 5. It grabs `"a"` as the `topElement`. Stack is now [], empty.
- Thread 2 runs line 5. It tries to grab the `topElement`, but the stack is empty, so it crashes.

This violates the first promise; Thread 2's peek call does not return the top element of the stack.