CSE 373 Practice Final Exam #1
ANSWER KEY

1. Big-Oh Analysis

a) O(*N*)
b) O(*N* log *N*)
c) O(*N* log *N*)
d) O(1)

2. Java / Guava Collection Programming

```
public static int mode(List<String> list) {
    Multiset<String> mset = HashMultiset.create();
    for (String s : list) {
        mset.add(s);
    }

    int modeCount = 0;
    for (String s : mset.elementSet()) {
        int count = mset.count(s);
        if (count > modeCount) {
            modeCount = count;
        }
    }
    return modeCount;
}

public static int mode(List<String> list) {
    Multiset<String> mset = HashMultiset.create(list);
    int max = 0;
    for (String s : mset.elementSet()) {
        max = Math.max(max, mset.count(s));
    }
    return max;
}
```

3. AVL Trees

a) after adds

```
            96
         /      \
      32          138
    /    \       /   \
  19      64   107   209
 /  \    /  \
10  21  53  75
```

b) after removes

```
      64
    /    \
  32      107
 /       /   \
10      75   209
```

4. Sort Tracing

```
a) selection sort
{16, 21, 45, 8, 11, 53, 3, 26, 49}
{3, 21, 45, 8, 11, 53, 16, 26, 49}
{3, 8, 45, 21, 11, 53, 16, 26, 49}
{3, 8, 11, 21, 45, 53, 16, 26, 49}
{3, 8, 11, 16, 45, 53, 21, 26, 49}
{3, 8, 11, 16, 21, 53, 45, 26, 49}
{3, 8, 11, 16, 21, 26, 45, 53, 49}
{3, 8, 11, 16, 21, 26, 45, 53, 49}
{3, 8, 11, 16, 21, 26, 45, 49, 53}
```

```
b) shell sort
{16, 21, 45, 8, 11, 53, 3, 26, 49, 31, 12, 55, 6, 18, 31, 15, 4}
{4, 21, 12, 8, 6, 18, 3, 15, 16, 31, 45, 55, 11, 53, 31, 26, 49}
{4, 18, 3, 8, 6, 21, 12, 15, 11, 31, 31, 26, 16, 53, 45, 55, 49}
{3, 8, 4, 15, 6, 18, 11, 21, 12, 26, 16, 31, 31, 53, 45, 55, 49}
{3, 4, 6, 8, 11, 12, 15, 16, 18, 21, 26, 31, 31, 45, 49, 53, 55}
```

```
c) merge sort
{16, 21, 45, 8, 11, 53, 3, 26, 49, 31, 12, 55, 6, 18, 31, 15, 4}
{16, 21, 45, 8, 11, 53, 3, 26}{49, 31, 12, 55, 6, 18, 31, 15, 4}
{16, 21, 45, 8}{11, 53, 3, 26}
{16, 21}{45, 8}
{16}{21}
{16, 21}
        {45}{8}
        {8, 45}
{8, 16, 21, 45}
               {11, 53}{3, 26}
               {11}{53}
               {11, 53}
                       {3}{26}
                       {3, 26}
               {3, 11, 26, 53}
{3, 8, 11, 16, 21, 26, 45, 53}
                              {49, 31, 12, 55}{6, 18, 31, 15, 4}
                              {49, 31}{12, 55}
                              {49}{31}
                              {31, 49}
                                      {12}{55}
                                      {12, 55}
                              {12, 31, 49, 55}
                                              {6, 18}{31, 15, 4}
                                              {6}{18}
                                              {6, 18}
                                                     {31}{15, 4}
                                                         {15}{4}
                                                         {4, 15}
                                                     {4, 15, 31}
                                              {4, 6, 15, 18, 31}
                              {4, 6, 12, 15, 18, 31, 31, 49, 55}
{3, 4, 6, 8, 11, 12, 15, 16, 18, 21, 26, 31, 31, 45, 49, 53, 55}
```

```
d) quick sort
{16, 21, 45, 8, 11, 53, 3, 26, 49, 31, 12}
{12, 21, 45, 8, 11, 53, 3, 26, 49, 31, 16}
     3                  21
         11      45
{12, 3, 11, 8, 16, 53, 21, 26, 49, 31, 45}
{12, 3, 11, 8}    {53, 21, 26, 49, 31, 45}
{8, 3, 11, 12}
{8, 3, 11}
{11, 3, 8}
 3  11
{3, 8, 11}
{3}
      {11}
                  {45, 21, 26, 49, 31, 53}
                  {45, 21, 26, 49, 31}
                  {31, 21, 26, 49, 45}
                  {31, 21, 26, 45, 49}
                  {31, 21, 26}
                  {26, 21, 31}
                  {26, 21}
                  {26}
                      {21}
                                  {49}
```

5. Sorting Algorithm Selection

```
a) insertion sort.
It works well for ascending or mostly-ascending input (O(N)) because it does not need
to slide each element very far to put it into place.

b) heap sort.
Quick sort has O(N^2) worst-case runtime.
Merge sort uses extra O(N) memory.
And shell sort is O(N^1.25), not O(N log N).
So heap sort is the only choice that fits all the criteria.
If you build the heap in-place, it is O(N log N) worst-case and uses O(1) memory.

c) merge sort.
You can load each small piece into memory, sort it, and then write it back to disk.
Then later you can load pairs of pieces into memory and sort them.

d) insertion sort (possibly selection sort, shell sort...).
It is not worth the overhead of running one of the O(N log N) sorts like quick sort
since the data is so small.  So we optimize by calling insertion sort, the fastest
O(N^2) sort, like we did in our optimized quick sort method on small input sizes.

e) bucket sort.
Bucket sort is great for grouping similar data into "buckets" like this.  It can sort
the data in O(N) time if the number of buckets is reasonably small like this.

f) selection sort.
Just run the first k passes of the algorithm, then you know that elements 0--(k-1) are
now in sorted order.
```

6. Sorting Algorithm Implementation

```
public static void bidiBubbleSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        if (i % 2 == 0) {
            // sweep left to right
            for (int j = i / 2 + 1; j < a.length - i / 2; j++) {
                if (a[j - 1] > a[j]) {
                    swap(a, j - 1, j);
                }
            }
        } else {
            // sweep right to left
            for (int j = a.length - i / 2 - 2; j >= i / 2 + 1; j--) {
                if (a[j - 1] > a[j]) {
                    swap(a, j - 1, j);
                }
            }
        }
    }
}

// Big-Oh is N^2.
```

7. Graph Properties

a) directed

b) weighted

c) unconnected (example: can't get from C to any other vertex; can't get from F or J to any other vertex)

d) cyclic (example cycle: A -> B -> A)

e) A: in 1, out 2

B: in 2, out 2

C: in 0, out 2

E: in 1, out 2

F: in 4, out 0

G: in 1, out 2

H: in 1, out 1

I: in 1, out 2

J: in 2, out 0

f) adjacency list:

```
 +----+    +----+    +----+
A|    |-->|B:6|-->|E:2|
 +----+    +----+    +----+
B|    |-->|A:6|-->|F:1|
 +----+    +----+    +----+
C|    |-->|B:2|-->|G:1|
 +----+    +----+    +----+
E|    |-->|F:8|-->|H:3|
 +----+    +----+    +----+
F| /  |
 +----+    +----+    +----+
G|    |-->|F:2|-->|J:5|
 +----+    +----+    +----+
H|    |-->|I:1|
 +----+    +----+    +----+
I|    |-->|F:2|-->|J:1|
 +----+    +----+    +----+
J| /  |
 +----+
```

adjacency matrix:

	A	B	C	E	F	G	H	I	J
A	0	6	0	2	0	0	0	0	0
B	6	0	0	0	1	0	0	0	0
C	0	2	0	0	0	1	0	0	0
E	0	0	0	0	8	0	3	0	0
F	0	0	0	0	0	0	0	0	0
G	0	0	0	0	2	0	0	0	5
H	0	0	0	0	0	0	0	1	0
I	0	0	0	0	2	0	0	0	1
J	0	0	0	0	0	0	0	0	0

8. Graph Paths

a) DFS:
A -> E -> H -> I

b) BFS:
A -> E -> H -> I

c) Dijkstra's:

	Visited?	Cost	Previous
A	X	0	/
B	X	6	A
C	X	inf	/
E	X	2	A
F	X	7	B
G	X	inf	/
H	X	5	E
I	X	6	H
J	X	7	I

d) There is no valid topological sort ordering because there is a cycle in this graph (A -> B -> A).

9. Graph Implementation

```
// single loop that checks both ways
public static boolean isValidBidiPath(Graph<String, String> g, List<String> path) {
    for (int i = 0; i < path.size() - 1; i++) {
        String first = path.get(i);
        String second = path.get(i + 1);
        if (!g.containsEdge(first, second) || !g.containsEdge(second, first)) {
            return false;
        }
    }
    return true;
}


// two loops, each checks one way
public static boolean isValidBidiPath(Graph<String, String> g, List<String> path) {
    // check forwards
    for (int i = 0; i < path.size() - 1; i++) {
        String first = path.get(i);
        String second = path.get(i + 1);
        if (!g.containsVertex(first) || !g.containsEdge(first, second)) {
            return false;
        }
    }

    // check backwards
    for (int i = path.size() - 1; i > 0; i--) {
        String first = path.get(i);
        String second = path.get(i - 1);
        if (!g.containsVertex(first) || !g.containsEdge(first, second)) {
            return false;
        }
    }

    return true;
}


// guaranteed O(N) for linked list or array list (above are O(N^2) for LL)
public static boolean isValidBidiPath(Graph<String, String> g, List<String> path) {
    Iterator<String> itr = path.iterator();
    String prev = null;
    while (itr.hasNext()) {
        String next = itr.next();
        if (!g.containsVertex(next)) {
            return false;
        }
        if (prev != null) {
            if (!g.containsEdge(prev, next) || !g.containsEdge(next, prev)) {
                return false;
            }
        }
        prev = next;
    }
    return true;
}
```

10. Parallel and/or Concurrent Programming

No, the code is not thread-safe. Here is an example order of execution for 2 threads that breaks the state of the account. Suppose the initial balance is $100.00 and two threads run the `withdraw` method at the same time. Thread 1 tries to withdraw $60 and thread 2 tries to withdraw $70. The account should not allow both of these to go through, because it would result in a balance of $-30. But if it executes in the order shown below, the account does receive the negative balance:

```
balance = 100.00
```

```
// Thread 1                  // 60     | // Thread 2
public void withdraw(double amount) {  | ...
    if (amount > 0.0 &&                | ...
            amount <= this.balance) { // true | ...
        double b = this.balance;  // 100 | ...                           // 70
---------------------------------------+-----------------------------------------------
        ...                            | public void withdraw(double amount) {
        ...                            |     if (amount > 0.0 &&
        ...                            |         amount <= this.balance) {  // true
---------------------------------------+-----------------------------------------------
        b = b - amount;           // 40 | ...
        this.balance = b;         // 40 | ...
    }                                  | ...
}                                      | ...
---------------------------------------+-----------------------------------------------
                                       |         double b = this.balance;  // 40
                                       |         b = b - amount;           // -30
                                       |         this.balance = b;         // -30
                                       |     }
                                       | }
```

The fix is to make the methods synchronized, so that only one thread could execute either method at a time:

```
public class BankAccount {
    private double balance;
    ...

    public synchronized void deposit(double amount) {
        if (amount > 0.0) {
            double b = this.balance;
            b = b + amount;
            this.balance = b;
        }
    }

    public synchronized void withdraw(double amount) {
        if (amount > 0.0 && amount <= this.balance) {
            double b = this.balance;
            b = b - amount;
            this.balance = b;
        }
    }
}
```