CSE 373 Practice Final Exam #2

1. Big-Oh Analysis

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*.

```
a)
Map<Integer, Integer> map =
        new TreeMap<Integer, Integer>();
for (int i = 1; i <= N; i *= 2) {
    map.put(i, i + 1);
}
System.out.println("done!");
```

```
b)
Set<Integer> set = new HashSet<Integer>();
for (int i = 0; i < 9999999; i++) {
    set.add(i * N);
}
System.out.println("done!");
```

```
c)
int sum = 0;
for (int j = 0; j < 10000 * N; j++) {
    for (int i = N; i > 0; i /= 2) {
        sum++;
    }
}
System.out.println(sum);
```

```
d)
List<Integer> list = new
ArrayList<Integer>();
for (int i = 0; i < N; i += 3) {
    list.add(i);
}
Queue<Integer> pq =
        new PriorityQueue<Integer>();
for (int i = 0; i < list.size(); i++) {
    pq.add(list.remove(0));
}
System.out.println("done!");
```

2. Java / Guava Collection Programming

Write a method called `commonFirstName` that accepts as parameters two `List`s of strings, one of which is a list of first names and the other of which is a list of their corresponding last names, and returns the first name that is associated with the greatest number of unique last names. The first name at index 0 in the first list goes with the last name at index 0 in the second list; the first name at index 1 goes with the last name at index 1; and so forth. For example, if passed these lists:

```
  index          0     1      2      3      4            6      7      8       9
first names: [  Bob, Mary, Steve, Derek,  Mary, Derek,  Joe, Derek, Nicole, Mary ]
last names:  [Jones, Ford, Akers, Smith, Giles, Smith, Fell, Jones, Jones,  Stepp]
```

Your method should return "Mary" because Mary is associated with three last names (Ford, Giles, and Stepp), but all the other first names are only associated with one or two last names. Although there are three entries for Derek, there are only two unique corresponding last names (Smith and Jones), and therefore its count is less than that of Mary. If there is a tie, your method may return either answer. You may use *one Guava collection* as auxiliary storage.

You may assume that neither list nor any of their elements are `null`. If passed two lists of different lengths or if either list is empty, your method should throw an `IllegalArgumentException`. Do not modify either list.

3. Heaps

Given the following integer elements:

- 138, 21, 10, 96, 209, 107, 64, 32, 75, 53, 19

a) Draw the tree representation of the **binary min-heap** that results when all of the above elements are **added** (in the given order) to an initially empty binary minimum heap. Circle the final tree that results from performing the additions. Also show the final array representation of the heap.

Please show your work. You do not have to draw an entirely new heap after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to show the heap at various important stages, along with an indication of any bubbling performed, to help earn partial credit in case of an error. Please ***circle your final answer*** *to make it clear what part of the page is supposed to be graded.*

b) After adding all the elements, perform **2 remove-min operations** on the heap. Circle the tree that results after the two elements are removed. Also show the final array representation of the heap.

4. Sort Tracing

a)
```
// index  0   1   2   3   4   5   6   7   8   9
       { 26,  7, 63, 42, 12, 34,  1, 10, 14, 30}
```
Trace the execution of the *merge sort* algorithm over the array above. Show each pass of the algorithm and the state of the array after the pass has been performed, until the array is sorted.

b)
```
// index  0   1   2   3   4   5   6   7   8   9
       { 11, 37, 99, 77, 60, 68, 53, 10, 70, 56}
```
Trace the execution of the *heap sort* algorithm over the array above. Show the max-heap being built and each max element being "removed" until the array is sorted.

c)
```
// index  0   1   2   3   4   5   6   7   8   9
       {  6,  0,  9,  3,  6,  5,  2,  3,  1,  1}
```
Trace the execution of the *bucket sort* algorithm over the array above. Show the buckets created and their contents, along with the final sorted array.

5. Sorting Algorithm Implementation

Write a method called `charBucketSort` that is a version of bucket sort that works on `char` values. The method is passed an array of `char`s, each of which consists of one lowercase letter from `a` through `z`. You may assume that these are the only characters in the array. Your code should count up the characters in buckets and use the buckets to restore the sorted contents of the array. For example, if passed the array:

```
{'y', 'c', 'r', 'b', 'c', 'i', 'm', 'c', 'i', 'i', 'r', 'i'}
```

You should compute the following bucket counts to sort it:

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*
char	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*	*q*	*r*	*s*	*t*	*u*	*v*	*w*	*x*	*y*	*z*
count	0	1	3	0	0	0	0	0	4	0	0	0	1	0	0	0	0	2	0	0	0	0	0	0	1	0

It may help you to remember that `char` values can be cast to and from type `int` to temporarily treat them as numeric values.

- What is the runtime (Big-Oh) of this algorithm?

6. Graph Properties

For the graph shown below, answer the following questions:

a) Is the graph connected, strongly connected, or unconnected?
If it is not connected, give an example of why not.
If the graph were undirected, would that change your answer? Why or why not?

b) Is the graph cyclic or acyclic?
If it is cyclic, give an example of a cycle.

c) Which vertex has the greatest in-degree, and what is its in-degree?

d) Write a complete *edge list* and *adjacency list* representation of the graph.

```
          13
 +--- B ------► C ◄--+
1|    |         ▲    |
 |    |         |    |3
 ▼    |         |    |
 A  6 |       8 |    D
 |    |         |    ▲
 |    |         |    |
2|    ▼         |    |2
 +--► E ------► F ---+
           4
```

7. Graph Paths

For the graph shown below, answer the following questions:

a) Write the path that a *depth-first search* (DFS) would find from vertex B to vertex D. Assume that any for-each loop over neighbors returns them in ABC order.

b) Perform *Dijkstra's algorithm* on the following graph to find the minimum-weight paths from vertex A to all other vertices. Reconstruct the path from vertex A to vertex C and give the total cost of the path.

c) Perform a *topological sort* on 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.

$$\begin{tikzcd}
 & B \arrow[dl, "1"'] \arrow[dd, "6"] \arrow[r, "13"] & C & \\
A \arrow[dr, "2"'] & & & D \arrow[ul, "3"'] \\
 & E \arrow[r, "4"'] & F \arrow[u, "8"] \arrow[ur, "2"'] &
\end{tikzcd}$$

8. Graph Implementation

Write a method `popular` that accepts a `Graph<String, String>` as a parameter and returns a `Set` of vertices in that graph that are "popular". Assume that the graph represents users on a social network such as Facebook. A vertex represents a user, and a directed edge from A to B represents the fact that user A "likes" user B. The weight of the edge represents how much A "likes" B.

A user *v* is "popular" if all of the following conditions are met:

- At least 2 other users "like" *v*.
- More users "like" *v* than *v* "likes" other users. *(More arrows coming in than going out.)*
- The combined weight of all "likes" toward v is more than the combined outbound weight of all the edges to other users that *v* "likes". *(More total edge weight coming in than going out.)*

For example, in the graph below, vertex B is "popular" because vertices A, C, and E "like" him with a combined weight of 3+2+4=9, while he "likes" only vertex E with a weight of 4. For this particular example graph, your method would return the set `[B, F, G]`. The set's elements should appear in alphabetical order.

You may assume that the graph and its vertices are not `null`. Your method should run in at worst O(V^2) time where V is the number of vertices in the graph. You may not construct any auxiliary collections to solve this problem, but you may create as many simple variables as you like.

```
       3       2
   A ----► B ◄---- C
   |       ▲       |
  2|       |       |
   |       |4      |2
   ▼   5   ▼   1   ▼
   D ----► E ◄---► F
   |       |
  1|       |4
   |       |
   ▼   3   ▼
   G ◄---- H       I ---► J
```

9. Parallel and/or Concurrent Programming

The Major League Baseball Commissioner has asked your team to write software to manage baseball players. Your teammate has written some code for making a trade between two teams, where team #1 gives player #1 to team #2, who gives player #2 back to team #1 in exchange. Your teammate knows that concurrency is hard, so he has made some of the code `synchronized`. However, you can see that there is a case where if two threads run this code at the same time, and each thread passes certain parameters, and the right order of interleaved instructions are executed, then both threads would "deadlock" (become locked up and unable to proceed indefinitely).

What is such a case? Describe it by writing specifically what parameters would be passed by each thread, and in what order the lines would execute (give lists of ranges of numbers).

```
 1  // Moves player1 from team1 to team2, and moves player2 from team2 to team1.
 2  // If player1 is not on team1, or if player2 is not on team2,
 3  // throws an IllegalArgumentException.
 4
 5  public void trade(Player player1, Set<Player> team1,
 6                    Player player2, Set<Player> team2) {
 7      if (!team1.contains(player1) || !team2.contains(player2)) {
 8          throw new IllegalArgumentException();
 9      }
10      synchronized (team1) {
11          synchronized (team2) {
12              team1.remove(player1);
13              team2.remove(player2);
14              team1.add(player2);
15              team2.add(player1);
16          }
17      }
18  }
```