CSE 373 Practice Final Exam #1

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)
```
int sum = 0;
int X = 100000;
for (int i = 1; i <= 4 * N; i++) {
  for (int j = 1; j <= X + 2; j++) {
    sum++;
  }
  for (int j = 1; j <= X * 100; j += 2) {
    for (int k = 1; k <= X * X; k++) {
      sum++;
    }
  }

  sum++;
}
System.out.println(sum);
```

b)
```
Map<Integer, Integer> map =
    new TreeMap<Integer, Integer>();
for (int i = 0; i < N; i += 2) {
  map.put(i, N * N);
}
System.out.println("done!");
```

c)
```
int sum = 0;
for (int j = 0; j < 100 * N; j++) {
  for (int i = N; i > 0; i /= 2) {
    sum++;
  }
}
System.out.println(sum);
```

d)
```
Random rand = new Random();
Queue<Integer> pq =
    new PriorityQueue<Integer>();

for (int i = 0; i < 99999; i++) {
  for (int j = 0; j < 99999; j++) {
    pq.add(N * rand.nextInt());
  }
}
System.out.println("done!");
```

2. Java / Guava Collection Programming

Write a method named `mode` that accepts a `List` of strings as a parameter and returns an integer representing the number of occurrences of the most frequently occurring element of the list (also called the "mode" of the list). If the list is empty, return 0. If all elements in the list are unique, return 1. For example, if your method were passed the list `{A, ZZ, B, B, G, A, B, B, XY, K, F, B, C, A, D}`, you would return `5` because there are 5 occurrences of `"B"`, the most frequently occurring element. You may use one auxiliary collection from the Guava framework to help you.

Your code should run in O(*N*) time where *N* is the number of elements in the list. You may assume that neither the list nor any of its elements are `null`. Do not modify the list in any way.

---

3. AVL Trees

Given the following integer elements:

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

a) Draw the AVL tree that results when the above elements are **added** (in the given order) to an empty AVL tree.

Please show your work. You do not have to draw an entirely new tree 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 tree at various important stages, along with an indication of any rotations performed (which case it is, and what rotation(s) you are performing to fix it), to help earn partial credit in case of an error. Please ***circle your answer*** *to make it clear what part of the page is supposed to be graded.*

b) Draw the AVL tree from part (a) after all of the following elements are **removed**. Recall that our BST remove algorithm chooses the *leftmost element from the right subtree* if necessary on removal of a node.

- 19, 53, 96, 138, 21

4. Sort Tracing

a)
```
// index  0   1   2   3   4   5   6   7   8
       { 16, 21, 45,  8, 11, 53,  3, 26, 49}
```
Trace the execution of the *selection 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  10  11  12  13  14  15  16
       { 16, 21, 45,  8, 11, 53,  3, 26, 49, 31, 12, 55,  6, 18, 31, 15,  4}
```
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.

c)
```
// index  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
       { 16, 21, 45,  8, 11, 53,  3, 26, 49, 31, 12, 55,  6, 18, 31, 15,  4}
```
Trace the execution of the *merge sort* algorithm over the array above. Show each pass of the algorithm and the splitting/merging of the array, until the array is sorted.

d)
```
// 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 partitioning calls over a single element, because there is nothing to do.

5. Sorting Algorithm Selection

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 20-30 values for sorting purposes (e.g., the data is records of elementary-school students and the sort is by first letter of last name).

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.

6. Sorting Algorithm Implementation

Write a method called `bidiBubbleSort` that is a bi-directional version of bubble sort on an array of integers. That is, it makes alternating sweeps over the array, first going from the front to the end, then from the end to the front, and so on. When sweeping from front to end, it swaps forward an element that is larger than the element after it; when sweeping from end to front, it swaps backward an element that is smaller than the element before it. You may assume that the array is not `null`.

The following diagram shows a run of this algorithm over a given array:

```
{16, 21, 45,  3, 11, 53,  8, 26, 49}

                  -->                sweep right
 16  21   3  11  45   8  26  49  53

                  <--                sweep left
  3  16  21   8  11  45  26  49  53

                  -->                sweep right
  3  16   8  11  21  26  45  49  53

                  <--                sweep left
  3   8  16  11  21  26  45  49  53

                  -->                sweep right
  3   8  11  16  21  26  45  49  53

                  <--                sweep left
```

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

7. Graph Properties

For the graph shown below, answer the following questions:

a) Is the graph directed or undirected?

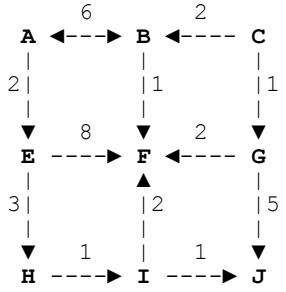
b) Is the graph weighted or unweighted?

c) Is the graph connected, strongly connected, or unconnected?
If it is not connected, give an example of why not.

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

e) What is the degree of each vertex?
If it is directed, what is the in-degree and out-degree of each vertex?

f) Write a complete *adjacency list* and *adjacency matrix* representation of the graph.



8. Graph Paths

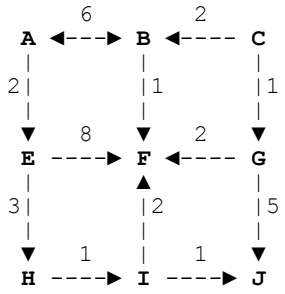
For the graph shown below, answer the following questions:

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

b) Write the path that a *breadth-first search* (BFS) would find from vertex A to vertex I. Assume that any for-each loop over neighbors returns them in ABC order.

c) 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 F and give the total cost of the path.

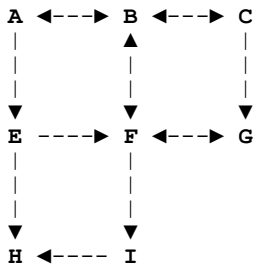
d) Perform a *topological sort* on the graph. Any valid topological sort ordering is correct. If it is not possible to produce a topological sort of the graph, write "There is no valid topological sort" and explain why using specific properties of the graph.



9. Graph Implementation

Write a method `isValidBidiPath` that accepts two parameters: a `Graph<String, String>`, and a `List` of vertices representing a path, and returns `true` if that path is an actual valid bi-directional path that can be traveled in the graph *in both directions*, or `false` if not. A valid bi-directional path is a path where every vertex in the path is found in the graph, and where there is an edge between each neighboring pair of vertices in the path along the way from start to end, and also back from the end to the start.

For example, if your method were passed given the graph below and the path [A, B, F, G], you would return `true` because all of those vertices are part of the graph and you can travel that path from A to G and back from G to A. If you were passed the same graph and the path [A, E, F, I], you would return `false` because that path does not go back from I to A. If you were passed the path [A, X, Z, B], you would return `false` because X and Z are not in the graph. You should not construct any auxiliary data structures while solving this problem, but you can construct as many simple variables as you like. You should not modify the state of the graph passed in. You may assume that the path and its elements are not `null`.



10. Parallel and/or Concurrent Programming

Consider the following code for making deposits and withdrawals on a `BankAccount` class. Is the code thread-safe? That is, are the invariant properties of `BankAccount` objects maintained even if a `BankAccount` object is accessed or modified by multiple threads concurrently? If so, explain why and justify your answer. If not, give an example of an order of execution for two threads that would cause a problem, and explain specifically how the code could be altered to become thread-safe.

```
// A BankAccount object represents one user's financial account data.
// Class invariant: balance >= 0.0 at all times.

public class BankAccount {
    private double balance;
    ...

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

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