1. In each of the following situations, would it be better to use array-backed storage or a singly linked list?

   (a) Inserting a random number into a sorted list.

   (b) Creating a list that will hold all the players in a game. The game has a max limit on the number of players.

   (c) Deleting the last element from the list.

   (d) Deleting from the head end of the list.

2. Suppose we are talking about the depth-first search (DFS) algorithm. Nodes are added to the data structures in alphabetical order.

   (a) What underlying data structure does this algorithm use?

   (b) Given the following graph, state the DFS traversal order and show the data structure at each step. Node A is the start node, and F is the destination node.

   (c) What path from A to F does the DFS algorithm return?

3. Now consider a BFS algorithm, again populating data structures in alphabetical order.

   (a) What changes would need to be made to a DFS implementation to turn it into a breadth-first search (BFS)?

   (b) Using the graph as described in Question 1, what is the BFS traversal order? Show the data structure at each step.
(c) What path from A to F results from the BFS algorithm?

4. Write code to naturally sort the given list in as few lines as possible. You should not need to write a
loop or use any numbers (Integer, int, Double, double, BigInteger, etc...).
*Hint: use the Java Collections Framework.*

```
List<String> list = Arrays.asList("a", "o", "d", "f", "x");
```

5. Convert the following code to use generics.

```java
interface StringCondition {
    boolean checkString(String s);
}

interface IntegerCondition {
    boolean checkInteger(Integer i);
}

class StringContainer {
    ArrayList<String> values;

    String getFirstWhereHolds(StringCondition condition) {
        for (String s : values) {
            if (condition.checkString(s))
                return s;
        }
        return null;
    }
}

class IntegerContainer {
    ArrayList<Integer> values;

    Integer getFirstWhereHolds(IntegerCondition condition) {
        for (Integer i : values) {
            if (condition.checkInteger(i))
                return i;
        }
        return null;
    }
}
```

```java
interface Condition<E> {
    boolean check(E value);`
```

```java
public class Container <E> {
    ArrayList<E> values;

    E getFirstWhereHolds (Condition<E> condition) {
        for (E v : values) {
            if (condition.check(v))
                return v;
        }
        return null;
    }
}
```
6. Reimplement the following function using an `Iterator` instead of a `for`-each loop.

```java
public static Integer sum ( ArrayList<Integer> lst ) {
    Integer total = 0;
    for ( Integer elem : lst ) {
        total = total + elem;
    }
    return total;
}
```

7. Briefly describe the difference (for objects) between `a.equals(b)`, `a==b`, `a.compareTo(b)`, and `Comparator.compare(a,b)`.

- `a.equals(b)` compares objects for equality. Class `Object` provides a default implementation (to be precise, `==` by default) that can be overridden for behavior necessary for a certain class. Returns a `boolean`.
- `a == b` checks memory locations (if the two objects are the SAME object, as defined by whether or not `a` and `b` point to the same spot in memory). Can also be used to check whether `a` is `null`. Also returns a `boolean`.
- `a.compareTo(b)` returns an `int` indicating whether `a` is less than (-1), equal to (0), or greater than (+1) `b`, according to their natural ordering. Specified by the `Comparable` interface.
- `Comparator.compare(a,b)` returns a negative if `a < b`, 0 if `a = b`, a positive if `a > b`.

8. Briefly explain the differences between the three kinds of exceptions: checked exceptions, runtime exceptions, and errors.

- Checked exceptions - Exceptions that a method signature must specify it throws. If a method may throw a checked exception, all calls to that method must be within a `try`-`catch` block. Checked exceptions should be used exclusively for foreseeable runtime mistakes, and any reasonably robust system should be able to recover from one. Classic example is `IOException`.
- Runtime exception - Not declared in a method signature and not anticipated to be thrown. Usually arise due to software bugs and often cause the program to crash. Classic examples are `NullPointerException` and `ArrayIndexOutOfBoundsException`.
- Errors - Represent a serious issue outside of the control of the programmer (hard drive failure, not enough memory, device issue). Examples are `IOException`, `VirtualMachineError` and `ThreadDeath` (see Java’s `Error` class).

9. Is there anything wrong with the following exception handler as written? Will this code work as intended?

```java
try {
    this.epicFail();
}
```
10. **Searching a Graph**

(a) Write a recursive algorithm that (given a graph, start vertex, and goal vertex), determines whether or not there is a path to the goal vertex.

Assume you are provided with a `Graph` class with a `getNeighbors(int vertex)` method, which returns a `Set<Integer>` representing the numbers corresponding to neighboring vertices. Assume `visited` is a `Set` keeping track of all visited vertices.

(Note: Your algorithm should return a Boolean value, not an actual path!)

```java
boolean hasPathToRec(Graph g, int start, int goal, Set<Integer> visited) {
    if (start == goal) {
        return true;
    } else {
        for (int n : g.getNeighbors(start)) {
            if (!visited.contains(n)) {
                visited.add(n);
                if (hasPathToRec(g, n, goal, visited)) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

(b) Rewrite your algorithm to be iterative instead.

(Hint: What data structure do you need to use if you no longer have recursion?)

```java
boolean hasPathToIter(Graph g, int start, int goal, Set<Integer> visited) {
    Stack<Integer> theStack = new Stack<Integer>();
    theStack.push(start);
    visited.add(start);
    while (!theStack.empty()) {
        int curr = theStack.pop();
        if (curr == goal) {
            return true;
        }
        for (int n : g.getNeighbors(curr)) {
            if (!visited.contains(n)) {
                visited.add(n);
                theStack.push(n);
            }
        }
    }
    return false;
}
```

11. Show the stages of a merge sort and a quicksort on the following list: [3, 5, 1, 3, 2, 7, 9]. Be sure to identify your pivot.
12. Fill in the table for the asymptotic running time of each sorting algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge sort</td>
<td>O(n * log(n))</td>
<td>O(n * log(n))</td>
<td>O(n * log(n))</td>
</tr>
<tr>
<td>Quicksort</td>
<td>O(n * log(n))</td>
<td>O(n^2)</td>
<td>O(n * log(n))</td>
</tr>
<tr>
<td>Heap sort</td>
<td>O(n * log(n))</td>
<td>O(n * log(n))</td>
<td>O(n * log(n))</td>
</tr>
</tbody>
</table>

13. In what scenario does Quicksort experience its worst-case time complexity? You may assume that we always pick the first element as the pivot.

14. What causes Quicksort to run so slowly on the input you describe in the last question?
15. Suppose you have an encoded file called secret.txt that must be decoded using a stream called DecodeStream. Write code to read the file, decode it using the DecodeStream, and print the contents. Assume the containing method throws all IOExceptions and DecodeStream is implemented to correctly handle buffered input.

```java
String line;
DecodeStream in = new DecodeStream(new BufferedReader(new FileReader("secret.txt")));
while ((line = in.readLine()) != null) {
    System.out.println(line);
}
```