1. [8 points] Answer "true" or "false" for each of the following statements about Java, and give a short explanation (≤ 20 words) for each answer. *IMPORTANT:* you *must* give an explanation for each to get credit!

a. If `x` is an `int` variable, the legal Java statement

```
x = (x - 1) & x;
```

turns off the rightmost 1-bit of `x`, if it has one, and has no other effect.

True. If `x` *is 0, then '*`&x`*' will return 0. Otherwise,* `x` *has the form* $2^n(k+1)$ *for* $k, n \geq 0$. *Thus* $x - 1 = 2^n k + (2^n - 1)$. *Since* $2^n - 1$ *is a string of* n *1-bits and* x *ends in* n *0's, we're left with just* $2^n k$, *which is* x *with the last 1 turned off.*

b. I decided to borrow Java's pseudo-random number generator to build my own coin-flipping class (it's a pretty good one of its kind).

```
public class RandomCoin {
    public RandomCoin (long seed) { value = seed; }
    /** True iff the next random coin flip turns up heads. */
    public boolean heads () {
        return next () % 2 == 1;
    }

    private long next () {
        value = value * 0x5DEECE66DL + 11L;  /* "L" means "long" */
        return value;
    }
    private long value;
}
```

This is good way to compute a random sequence of coin tosses (assuming I choose the seed well).

False. `value` *strictly alternates between being even and odd, so* `.next` *will return true and false in alteration.*

c. The following (legal) code

```
double[] zeroRow = new double[N];  /* Initialized to 0s */
double[][] I = new double[N][];
for (int i = 0; i < N; i += 1)
    I[i] = zeroRow;
for (int i = 0; i < N; i += 1)
    I[i][i] = 1.0;
```

sets the array `I` to the identity matrix (all 1's on the diagonal, 0's off the diagonal).

False. All rows of `I` *point to the same object, and so are all the same. Since each position gets set to 1's, this is therefore an array of all 1's.*

d. The following subprogram can work as specified in its comment.

```
/** Returns an array, [n0, n1, n2, ..., nk], where n0>0 is the number of
 *  times the smallest value (call it m) in B occurs in B,
 *  n1 is the number of times m+1 occurs, etc., and nk>0 is the number of
 *  times the largest value in B, m+k, occurs. Assumes B is not empty. */
int[] frequencies(int[] B) {
    int min, max;
    min = max = 0;
    findMinMax(B, min, max);
    int[] counts = new int[max-min+1];
    for (int n : B)
        counts[n - min] += 1;
    return counts;
}
```

False. No matter how it is defined, `findMinMax` *cannot set* `min` *and* `max` *(Java always passes by value).*

2. [6 points] Consider the following somewhat unorthodox implementation of a FIFO queue:

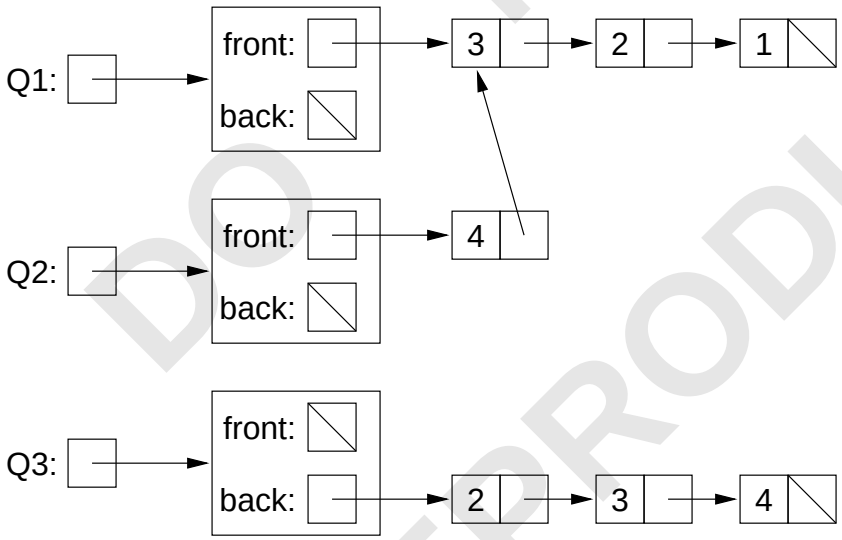
```
/** A FIFO queue with non-destructive operations. */
public class NDQueue<E> {
  private static class Node<T> {
    public T value;
    public Node<T> next;
    public Node(T value, Node<T> next) { this.value = value; this.next = next; }
  }

  private Node<E> front;
  private Node<E> back;

  private NDQueue(Node<E> front, Node<E> back) {
    this.front = front;
    this.back = back;
  }

  /** An empty queue. */
  public NDQueue() { front = null; back = null; }

  /** A new queue identical to THIS with the addition of ITEM at the end. */
  public NDQueue<E> enqueue(E item) {
    return new NDQueue<E>(new Node<E>(item, front), back);
  }

  /** A new queue identical to THIS but with the first item removed and
   *  stored in RESULT[0]. */
  public NDQueue<E> dequeue(E[] result) {
    Node<E> b = back;
    if (b == null) {
      if (front == null) {
        result[0] = null;
        return new NDQueue<E>(null, null);
      }
      for (Node<E> f = front; f != null; f = f.next)
        b = new Node<E>(f.value, b);
    }
    result[0] = b.value;
    return new NDQueue<E>(null, b.next);
  }
}
```

WARNING: Careful! This implementation has subtleties. For example, `.front` and `.back` might not be the first and last items in the queue.
Questions begin on the next page.

a. Show the state of the data structures (using boxes and arrows) after executing the following code:

```
NDQueue<Integer> Q1, Q2, Q3;
Q1 = (new NDQueue<Integer>()).enqueue(1).enqueue(2).enqueue(3);
Q2 = Q1.enqueue(4);
Integer[] a = new Integer[1];
Q3 = Q2.dequeue(a);
```

For brevity, feel free to represent `Integer`s as if they were plain `int`s.



Questions continue on the next page.

b. Give an asymptotic bound on the worst-case time for the following code snippet, assuming that the `.process` method and allocation of new objects are $\Theta(1)$. Give your reasoning (you must do this for full credit).

```
NDQueue<Integer> Q = new NDQueue<Integer>();
for (int i = 0; i < N; i += 1) {
   Q = Q.enqueue(i);
}
Integer[] r = new Integer[1];
for (int i = 0; i < N; i += 1) {
   Q = Q.dequeue(r);
   process(r[0]);
}
```

`Enqueue` *is clearly an* $O(1)$ *operation.* `Dequeue` *can be* $O(N)$ *in the worst case, but since each item that is enqueued is added to the* `back` *list only once, and removal from that list is* $O(1)$*, the total time for the entire sequence of operations is* $O(N)$*. In fact, the* `enqueue` *and* `dequeue` *operations have* $O(1)$ *amortized cost, since the total cost incurred by* `dequeue`*s is always no worse than proportional to the number of preceding enqueues.*

c. Give an asymptotic bound on the worst-case time for the following code snippet, and your reasoning for this bound, under the same assumptions as part (b).

```
NDQueue<Integer> Q = new NDQueue<Integer>();
Integer[] r = new Integer[1];
for (int i = 0; i < N; i += 1) {
   Q = Q.enqueue(2*i);
   Q = Q.enqueue(2*i + 1);
   Q = Q.dequeue(r);
   process((Integer) r[0]);
   Q = Q.dequeue(r);
   process((Integer) r[0]);
}
```

Also $O(N)$*. The reasoning is the same as for part (b).*

3. [5 points] A `ThingummyTree` is a binary search tree for `Thingummy` objects, which have a rather eccentric design. Instead of having a `.compareTo` function, like `Strings`, they define a `.choose` method:

```
public class Thingummy {
    ...
    /** If THIS is less than X, execute CHOICES.lt(); if equal to X,
     *  execute CHOICES.eq(), and otherwise CHOICES.gt().  X must not
     *  be null. */
    public void choose (Thingummy x, TriAction choices) {... }
}
```

where `TriAction` is defined:

```
public interface TriAction {
    void lt ();
    void eq ();
    void gt ();
}
```

Fill in the implementation of `ThingummyTree` on the next page *without* using Java library calls, nor any of the keywords **if**, **while**, **for**, **try**, or **switch**, nor the operators `?:`, `&&`, or `||` anywhere in your solution. Define all the additional instance variables, class variables, methods, constructors, and classes you want, but they also must abide by the restriction. (This is not a complete class: we are not asking for `.insert`, `.remove`, or `.size` methods.)

Reminder: no Java library calls, **if**s, **while**s, **for**s, **try**s, or **switch**es, nor the operators `?:`, `&&`, or `||`.

```
public class ThingummyTree {
    /** An empty tree, containing no Thingummys. */
    public static ThingummyTree EMPTY () { // FILL IN
        return EMPTY;
    }

    /** A new tree whose root contains LABEL, with left and right
     *  children LEFT and RIGHT.  The labels in LEFT must be less than
     *  LABEL and those in RIGHT must be greater than LABEL. */
    protected static ThingummyTree cons (Thingummy label,
                                         ThingummyTree left,
                                         ThingummyTree right) { // FILL IN
        return new NonEmpty(label, left, right);
    }

    /** True iff T contains X. */
    public boolean contains (Thingummy x) { // FILL IN
        return false;
    }

    private ThingummyTree() { }

    private final static ThingummyTree EMPTY = new ThingummyTree();

    protected Thingummy label;
    protected ThingummyTree left, right;

    private class Chooser implements TriAction {
        ThingummyTree tree;
        Thingummy target;
        boolean found;
        Chooser (ThingummyTree T, Thingummy x) { tree = T; target = x; }
        public void lt () {
            found = tree.left.contains (target);
        }
        public void eq () { found = true; }
        public void gt () {
            found = tree.right.contains (target);
        }
    }
```

```
    private static class NonEmpty extends ThingummyTree {
        NonEmpty (Thingummy label, ThingummyTree left, ThingummyTree right) {
            this.label = label; this.left=left; this.right=right;
        }
        public boolean contains (Thingummy x) {
            Chooser chooser = new Chooser (this, x);
            x.choose(label, chooser);
            return chooser.found;
        }
    }
}
```

4. [9 points] In this problem, you are to design and implement a simple route-planning class, such as might be employed by a GPS system. Basically, this system maintains a set of named *locations,* (which, depending on how one chooses to use it, could denote cities, intersections, houses, mile markers, or the like), with *route segments* (think roads, paths, etc.) linking them. The route segments contain information about their travel times. Clients using this class must be able to do the following:

- Create a new location, giving it a unique string that identifies it. Attempting to create a location a second time should have no effect.
- Add or modify a route segment linking two already existing locations, giving the time required to traverse this segment (represented by a `double`). Non-positive times should cause exceptions.
- Request the shortest or fastest route between two locations.

We should be able to use your class as follows:

```
String[] locations = { "Soda", "Cory", "VLSB", "Doe", "Wheeler", "Evans" };
Object[][] segments = {
   { "Soda", "Cory", 1.0 }, {"Cory", "Evans", 2.0 },
   { "Evans", "Doe", 3.0 }, { "Evans", "Wheeler", 2.5 },
   { "Wheeler", "VLSB", 2.0 }, { "Doe", "VLSB", 1.0 } };

RoutePlanner map = new RoutePlanner ();
for (String loc : locations)
    map.addLocation (loc);
for (Object[] seg : segments)
    map.addSegment ((String) seg[0], (String) seg[1],
                    (double) (Double) seg[2]);
for (String loc : map.fastestRoute ("Soda", "VLSB"))
    System.out.printf ("%s ", loc)
System.out.println ();

map.addSegment ("Soda", "Doe", 5.0);
for (String loc : map.fastestRoute ("Soda", "VLSB"))
    System.out.printf ("%s ", loc)
System.out.println ();
```

which will print

```
Soda Cory Evans Doe VLSB
Soda Doe VLSB
```

As you can see, it is possible to additional route segments after computing a path, thus changing the path. You should also be able to intermix the addition of segments and locations.

To make things easier, we have provided a utility routine that performs Dijkstra's algorithm. You don't have to use it. If you do, you'll see that it uses a data structure that is probably not exactly what's needed for this problem.

```
class Paths {

    /** An abstraction of a set of weighted edges in a graph whose
     *  vertices are identified by integers running from 0 .. size()-1. */
    static public interface Edges {
        /** The length of the edge running from vertex I to vertex J,
         *  or infinity if there is no such edge.  Assumes I and J are
         *  non-negative and less than size().  */
        double len(int i, int j);

        /** The number of vertices in the underlying graph. */
        int size();
    }


    /** Compute a shortest path in a graph whose edge weights are
     *  specified by E. START and DEST denote two vertices in the graph (so
     *  both must be in [0, E.size()-1]).  Returns an array (call it P)
     *  of size N such that a shortest path from START to DEST is
     *  the REVERSE of the path DEST, P[DEST], P[P[DEST]], ..., START
     *  (that is, if x is on the path, then P[x] is the vertex just before
     *  x on the path). */
    public static int[] getPath (Edges e, int start, int dest) {
       ...
    }
}
```

Fill in solution on the next pages.

Feel free to add additional methods and classes, to add interfaces to `RoutePlanner`, or to use any classes in the Java library.

```
public class RoutePlanner implements Paths.Edges {

   private HashMap<String,Integer> locs = new HashMap<String,Integer>();
   private ArrayList<String> names = new ArrayList<String>();
   private ArrayList<ArrayList<Double>> edges =
       new ArrayList<HashMap<Integer,Double>> ();

   public void addLocation (String name) { // FILL IN
       if (!locs.containsKey (name)) {
           locs.add (name, locs.size ());
           edges.add (new HashMap<Integer, Double>());
           names.add (name);
       }
   }

   public void addSegment (String loc0, String loc1, double time) { // FILL IN
       // We didn't require the following check.
       if (! locs.containsKey (loc0) || ! locs.containsKey (loc1))
          throw new IllegalArgumentException ();
       int i = locs.get (loc0), j = locs.get (loc1);
       // We didn't care whether you interpreted these as directed
       // edges.  Here, we treat them as directed.
       edges.get (i).put (j, time);
   }

   public double len (int i, int j) {
       Double r = edges.get (i).get (j);
       if (r == null)
           return Double.POSITIVE_INFINITY;
       else
           return r;
   }

   public int size () { return locs.size (); }
```

```
    public List<String> fastestRoute (String fromLoc, String toLoc) { // FILL IN
        // We didn't require the following check.
        if (! locs.containsKey (fromLoc) || ! locs.containsKey (toLoc))
           throw new IllegalArgumentException ();
        int i = locs.get (fromLoc), j = locs.get (toLoc);
        int[] parents = Paths.getPath (this, i, j);
        ArrayList<String> result = new ArrayList<String> ();
        // We didn't require that you check that a path exists, so
        // we'll just assume it does.
        result.add (j);
        while (j != i) {
            j = parents[j];
            result.add (j);
        }
        Collections.reverse (result);
        return result;
    }
}
```

5. [1 point] What mathematically interesting thing can we now say with certainty about any simply connected, closed 3-manifold that we couldn't in 1904?

It is homeomorphic to a sphere (this is the Poincaré conjecture, proved in 2002–2003).

6. [10 points] The following questions involve sorting. Warning: do not assume that the algorithms illustrated always conform exactly to those presented in the reader and lecture notes. We are interested in whether you understand the major ideas behind the algorithms. Where the question asks for a reason, you *must* provide an explanation to get credit.

a. Last year, T. C. Pits[1] acquired a Sorting Adversary Machine (SAM), which, given a sorting program, will figure out an arbitrarily large sequence of test data sets of increasing size that make it look as bad or good as possible (depending on its dial settings) and determine how it scales with increasing input size for these test data sets. The keys used are strings, and he sets the dial to count machine operations. He uses it to find worst-case performance for an implementation of heapsort. "Confound it!" he cries, "I asked it to find a sequence of data sets where the program scales as $\Theta(N^2 \lg N)$ and it succeeded (N is the number of keys). My heapsort must be broken; it's supposed to operate in time $O(N \lg N)$." Is he necessarily right? Why or why not? If not, what kind of test data sets could SAM have come up with to get this scaling from a correct heapsort implementation?

He is wrong. SAM simply created a sequence of data sets of increasing size such that the data set with k records had keys that were k characters long and differed only in the last characters. Each comparison then takes $\Theta(k)$.

b. When he applied SAM to a second sorting algorithm and set the dials for "best cases" and had it count comparisons rather than operations. This time, he found out that it ran in time $O(N)$, where N is the number of records to be sorted. Could this algorithm have been quicksort? Why or why not?

It could not have been quicksort. The best case for quicksort occurs when the pivots divide the array evenly. In that case, the running time is given by the recurrence $C(n) = n + 2C(n/2)$, with $C(1) = 1$. This gives us $\Theta(N \lg N)$.

[1] The Celebrated Programmer In The Street.

c. We haven't considered using a trie for sorting. Suppose that we sort a set of N non-empty strings containing a total of $B > N$ characters by inserting them into a trie and then traversing the trie to extract the strings in sorted order. Assume that all strings end in a unique terminating character that only appears at the end of strings. What can you say about the worst-case performance of this algorithm, and why?

The total time will be $O(B)$. Inserting a string S in a trie requires time $O(|S|)$ (where $|S|$ is the length of S). So inserting all strings will take time B, the sum of all the strings' lengths. Traversing the trie in symmetric order gives the sorted result. Such a traversal takes time proportional to the number of edges traversed, which, since a trie is a tree and its node count is $O(B)$, is again proportional to B.

d. What sorting algorithm does the following illustrate (and how do you know)? The first line is the input.

```
200 508 104 447 10 204 502 47 792 698 977 328 81 272 40 109
109 508 104 447 10 204 502 47 792 698 977 328 81 272 40 200
40 200 104 447 10 81 272 47 508 698 977 328 204 502 109 792
40 10 81 204 47 104 272 200 109 447 502 328 698 977 508 792
10 40 47 81 104 109 200 204 272 328 447 502 508 698 792 977
```

The first step sorted just 200 and 109—items number 0 and 15. The next step sorted 109, 47, and 40; 508, 792, and 200; 104 and 698; 447 and 977; 10 and 328; 204 and 81; and 502 and 272—all subsequences of items spaced 7 apart. The line does the same for items spaced 3 apart and the last, 1 apart. This is Shell's sort.

e. The following is supposed to illustrate major steps in a certain sorting algorithm. The first line is the input. Fill in the missing second-to-last line.

```
4759 4908 9801 28 8091 2451 7544 386 6468 4461 79 8317 2644 7484 717 9507
9801 8091 2451 4461 7544 2644 7484 386 8317 717 9507 4908 28 6468 4759 79
9801 9507 4908 8317 717 28 7544 2644 2451 4759 4461 6468 79 7484 386 8091

28 79 8091 8317 386 2451 4461 6468 7484 9507 7544 2644 4759 717 9801 4908
28 79 386 717 2451 2644 4461 4759 4908 6468 7484 7544 8091 8317 9507 9801
```

(This is LSD radix sort on numerals)

7. [12 points] Answer each of the following *briefly* and justify your answer. You *must* provide reasons to receive credit.

a. Since rooted trees are also graphs, one can topologically sort them using one of the algorithms we discussed (assume that the edges are taken to run from a node to its children). However, there is already a standard procedure on trees that would accomplish the same thing. What is it and how would one use it for this purpose?

A preorder tree traversal will visit all predecessors of a node before visiting the node.

b. Suppose that $f(x) \in O(x^2)$. Can we conclude that $1/f(x) \in O(1)$? Why or why not (if not, what is a counterexample)?

No. There is no lower bound on $|f(x)|$ implied by $O(\cdot)$, so its inverse can be unbounded.

c. While on a break from working on your project at home, your annoying little brother messes with your computer and randomly modifies your Subversion working directory so that you can no longer commit, update, or basically do anything useful. The Staff has decided to be unhelpful, having gotten tired of cleaning up your brother's messes. What do you do?

Simply check out a fresh copy of your working directory into a new directory, and then copy over any work you hadn't committed before the little brat got to work.

d. In a connected, directed graph that has a designated *entry node,* E, we say that node P *dominates* node C if all paths from E to C contain P. A node always dominates itself. Describe a procedure for determining if P dominates C.

If temporarily removing P and all edges incident to it makes C unreachable, then P must dominate C. In a recursive traversal, for example, you can simply mark P when you get to it, but then return immediately without traversing further. If you ever reach C by this procedure, then P does not dominate it.

e. Suppose that the loop

```
for (int i = 0; i < N; i += 1)
    P(i);
```

takes $O(N)$ time. Does this necessarily mean that `P` operates in $O(1)$ amortized time? Why or why not?

Not quite. In order for `P` *to operate in amortized time $O(1)$, the average time per execution of* `P` *so far must always be bounded by a constant. But in the loop above,* `P(0)` *could take time N and* `P(`i`)` *could take time 1 for $i > 0$.*

f. Why doesn't the following (legal) method work?

```
/** Returns true iff any two elements of L have the same contents. */
static public boolean hasDuplicates (ArrayList<int[]> L) {
    HashSet<int[]> seen = new HashSet<int[]> ();
    for (int[] item : L) {
        if (seen.contains (item))
            return true;
        seen.add (item);
    }
    return false;
}
```

Equality (`.equals`*) on arrays is the same as* `==`*: it tests for object identity without looking at contents. Hence, this procedure will detect duplicates only if they are in fact the same array objects.*