CSE 331 Winter 2019 Final Exam

Name ____

The exam is closed book and closed electronics. One page of notes is allowed.

Please **wait to turn the page** until everyone is told to begin.

Score: _________________ / 136

1. ______________ / 10

2. ______________ / 10

3. ______________ / 10

4. ______________ / 12

5. ______________ / 10

6. ______________ / 10

7. ______________ / 10

8. ______________ / 10

9. ______________ / 10

10. ______________ / 12

11. ______________ / 12

12. ______________ / 10

13. ______________ / 10

Bonus. _________________ / 20

Problem 1 (Specifications)

Fill in the specification of the given method making it as **strong as possible**.

```
/**
 * Returns the absolute value of numbers between -5 and 5.
 *
 * @param x A number
 * @throws IllegalArgumentException if |x| > 5
 * @return |x|
 */
public int whoKnows(int x) {
  if (x >= 0) {
    if (x > 5) {
      throw new IllegalArgumentException("too big");  // ***
    } else {
      return x;
    }
  } else {
    if (x < -5) {
      throw new IllegalArgumentException("too small");
    } else {
      return -x;
    }
  }
}
```

Suppose that Alice wants to change the line marked with `// ***` above, which currently throws an exception, to instead return `Integer.MAX_VALUE`. How could she change the specification so that it would allow either this new implementation or the original?

Write a new specification that is as **strong as possible** while allowing **both** implementations from above.

```
/**
  * Returns the absolute value of numbers between -5 and 5.
  *
  * @param x A number
  * @requires x <= 5
  * @throws IllegalArgumentException if x < -5
  * @returns |x|
  */
```

Problem 2 (Reasoning)

Fill in the implementation of the following method:

```
/**
 * Returns the value in A that is smallest out of all values in A
 * that are larger than x.
 *
 * @param x A number to compare to the values in A.
 * @param A A list of numbers
 * @requires A != null
 * @throws IllegalArgumentException if no value in A is
 *     <b>strictly</b> larger than x
 * @return the smallest of all values in A larger than x
 *
 */
public int nextLargest(int x, int[] A) {

  boolean hasLarger = false;
  int minLarger = Integer.MAX_VALUE;
  int i = 0;

  {{ Inv: let B = the set of values in A[0], ..., A[i-1] that are larger than x
       hasLarger is true iff B is not empty
       if B is not empty, then minLarger = min in B }}
  while (i < A.length) {
    if (x < A[i]) {
      hasLarger = true;
      minLarger = Math.min(A[i], minLarger);
    }

    i = i + 1;
  }

  if (!hasLarger)
    throw new IllegalArgumentException("nothing smaller");

  return minLarger;
}
```

Problem 3 (Testing)

Fill in the templates below to describe test cases for `nextLargest` on the previous page. For each one, specify the two inputs and describe what behavior you should see from the method.

Each test case must test a **distinct subdomain**. If you are concerned that it may not be entirely clear why the test case covers a distinct subdomain, you can write an explanation below the template.

a. Input: x = ___1______ A = [__2, 4______________________________________]

Result: ________returns 2___________________

b. Input: x = ___3______ A = [__2, 4______________________________________]

Result: ________returns 4____________________

c. Input: x = ___4______ A = [__2, 4______________________________________]

Result: ___throws IllegalArgumentException___

d. Input: x = ___2______ A = [__2, 4______________________________________]

Result: ________returns 4____________________

e. The above tests do not include the case A = null. How should that be handled in these tests?

it should be skipped

Problem 4 (Reasoning)

Consider the following code and assertions:

{{ A: w > 0 }}

```
x = 3 * w;
```

{{ B: ______________________________ }}

```
y = x + 10;
```

{{ C: y = 16 }}

a. Which of the following would be written for B using **forward reasoning**? Circle one.

[✓ w > 0 and x = 3 * w]   x > 0

y = 16 and x = 6   x = 6

b. Which of the following would be written for B using **backward reasoning**? Pick one.

w > 0 and x = 3 * w   x > 0

y = 16 and x = 6   [✓ x = 6]

c. Which of the following, when written for B, would make the triples valid? Circle all that apply.

w > 0 and x = 3 * w   x > 0

y = 16 and x = 6   x = 6

[✓ w = 2 and x = 6]   w = 2 and x > 0

The next few problems consider an implementation of the following ADT:

```
/**
 * A queue of integers **without** the ability to see the value of the
 * items that are removed from the queue but **with** the new ability
 * to retrieve the largest item in the queue.
 *
 * For example, enqueuing values 5, 9, 3, 4, in that order, gives
 * the abstract state [?, 9, ?, 4], where the ?s indicate values
 * that are no longer visible to the client because they will never,
 * at any point, be the maximum. The value 9 is the current maximum,
 * until it is removed from the queue, when 4 becomes maximum. If we
 * were to add 10, the state becomes [?, ?, ?, ?, 10].
 *
 * More specifically, an element in the queue becomes a "?" when
 * a larger element is added after it. Note that a "?" can never
 * become visible again since the larger element will always
 * remain after it in the queue.
 */
public interface MaxQueue {
  /**
   * Adds the given value to the end of the queue.
   * @param x the value to add
   * @modifies this
   * @effects Changes the state from [y1 ... yN] to [y1 ... yN, x]
   *    with some of the y's potentially becoming ?s
   */
  public void enqueue(int x);

  /**
   * Removes the first element from the front of the queue.
   * @requires the queue is not empty
   * @modifies this
   * @effects Changes the state from [x, y1 ... yN] to [y1 ... yN]
   *    with no changes to the y's
   */
  public void dequeue();

  /**
   * Returns the largest value current in the queue
   * @requires the queue is not empty
   * @returns largest value currently in the queue
   */
  public int getMax();

  /**
   * Returns the number of elements in the queue.
   * @returns the number of elements in the queue
   */
  public int size();
}
```

Problem 5 (ADTs)

In this problem, you will provide part of the implementation of the ADT on the prior page in the class `MaxQueueImpl`. Your concrete representation will be a pair of linked lists:

```
private LinkedList<Integer> visibleValues;
private LinkedList<Integer> hiddenCounts;
```

The list `visibleValues` contains all the numbers are still visible (i.e., the ones that could potentially become maximums if enough prior elements are removed), while the list `hiddenCounts` contains the number of '?'s between the visible value at the same index and the one prior to it.

For example, after adding 5, 9, 3, 2, 4, in that order, `visibleValues` would contain [9, 4] and `hiddenCounts` would contain [1, 2] since 9 has 1 "?" in front of it and 4 has 2 "?"s in front of it but after the 9.

a. Write the abstraction function and representation invariant for this representation:

```
// RI: visibleValues != null and hiddenCounts != null and
//     visibleValues.size() == hiddenCounts.size() and
//     every value in hiddenCounts is nonnegative
//
// If visibleValues = [v1, ..., vN] and hiddenCounts = [c1, .., cM]
// then AF(this) = [c1 x ?, v1, c2 x ?, v2, ..., cN x ?, vM]
//
// where the notation c x ? means c "?" values in a row.
```

b. Implement the default constructor of this class:

```
public MaxQueueImpl() {
  this.visibleValues = new LinkedList<Integer>();
  this.hiddenCounts = new LinkedList<Integer>();
}
```

c. Implement the method `getMax` defined in the `MaxQueue` interface:

```
@Override
public int getMax() {
  assert visibleValues.size() > 0;
  return visibleValues.get(0);
}
```

Problem 6 (Reasoning)

Fill in the implementation of the method `enqueue`, defined in the `MaxQueue` interface, using the concrete representation defined in the previous problem.

The invariant for the loop is provided for you. ***Do not*** add any additional loops.

You do not need to *turn in* a complete proof of correctness, but you should complete one since your code will be graded on correctness.

Recall that `LinkedList` provides the methods `getFirst`, `addFirst`, `removeFirst`, where `getFirst` returns the element at the front of the list, `addFirst` puts a new element at the front, and `removeFirst` removes and returns the element at the front. It also provides `getLast`, `addLast`, `removeLast`, defined similarly.

```
@Override
public void enqueue(int x) {

  int hiddenCount = 0;

  {{ Inv: hiddenCount is the number of hidden & visible elements removed so far }}
  while (visibleValue.size() > 0 && visible.getLast() < x) {
    hiddenCount += hiddenCounts.removeLast() + 1;

    visibleValues.removeLast();
  }

  visibleValues.addLast(x);
  hiddenCounts.addLast(hiddenCount);
}
```

Fill in the implementation of the method `dequeue` from MaxQueue for this same class:

```
@Override
public void dequeue() {

  assert hiddenCounts.size() > 0;
  if (hiddenCounts.getFirst() == 0) {
    hiddenCounts.removeFirst();
    visibleValues.removeFirst();
  } else {
    hiddenCounts.addFirst(hiddenCounts.getFirst() - 1);
  }


}
```

Problem 7 (Testing)

Write two **methods** for a JUnit implementation test of the class `MaxQueueImpl` of the previous problems that test the `enqueue` method. Your two tests must cover *distinct conditions* but must, together, achieve *100% branch coverage* of `enqueue`. (Recall that a loop has an implicit branch in it.)

Additional notes:

- You also assume that `MaxQueueImpl` has a `toString` method that returns a string description of the abstract state, e.g., a string like "[?, 9, ?, 4]".
- You can also assume that all the usual JUnit classes are imported.

Write your two test methods below:

```
@Test
public void testEnqueueSmaller() {

  MaxQueueImpl Q = new MaxQueueImpl();

  Q.enqueue(5);
  assertEquals("[5]", Q.toString());
  Q.enqueue(4);
  assertEquals("[5, 4]", Q.toString());

}


@Test
public void testEnqueueLarger() {

  MaxQueueImpl Q = new MaxQueueImpl();

  Q.enqueue(5);
  assertEquals("[5]", Q.toString());

  Q.enqueue(9);
  assertEquals("[?, 9]", Q.toString());

}
```

Do your methods also achieve path coverage?

Yes [✓ No]

Problem 8 (Generics I)

Write a new version of the `MaxQueue` interface that works not only with integers but with any type that supports comparison.

Inside the body of the new interface, include only the methods that would **change**. Likewise, include only those parts of the Javadoc that would change.

```
/**
 * (as above)
 *
 * @param <T> type of objects stored in queue
 */
public interface MaxQueue<T extends Comparable<T>> {

  /**
   * (as above)
   */
  public void enqueue(T x);

  /**
   * (as above)
   */
  public T getMax();

}
```

Write the **signature** for a new `enqueueAll` method that adds many elements to the queue in a single call. Try to make the method as general as possible.

```
public <S extends T> void enqueueAll(Collection<S> vals);
```

Problem 9 (Generics II)

Consider the following class definitions:

```
interface Animal {}
interface Mammal extends Animal {}
interface EggLayer extends Animal {}
interface Platypus extends Mammal, EggLayer {}

List<Animal> animals = new ArrayList<Animal>();
List<? extends Mammal> mammals = new ArrayList<? extends Mammal>();
List<? super EggLayer> eggLayers = new ArrayList<? super EggLayer>();
List<Platypus> platypuses = new ArrayList<Platypus >();

Animal animal = null;
Mammal mammal = null;
EggLayer eggLayer = null;
Platypus platypus = null;
```

Draw a line through those statements below that are **illegal** in Java.

```
~~mammals = animals;~~
mammals = platypuses;

eggers = animals;
~~eggers = platypuses;~~

~~mammals.add(animal);~~
~~mammals.add(mammal);~~
~~mammals.add(eggLayer);~~
~~mammals.add(platypus);~~

eggLayers.add(animal);
~~eggLayers.add(mammal);~~
eggLayers.add(eggLayer);
~~eggLayers.add(platypus);~~

animal = mammals.get(0);
mammal = mammals.get(0);
~~eggLayer = mammals.get(0)~~;
~~platypus = mammals.get(0)~~;

~~animal = eggLayers.get(0)~~;
~~mammal = eggLayers.get(0)~~;
~~eggLayer = eggLayers.get~~(0);
~~platypus = eggLayers.get~~(0);
```

Problem 10 (Subtypes)

Suppose we define the following interface:

```
/** Enumerates through a sequence of elements of type T. */
interface Enumerator<T> {

  /** @returns true iff there is another element to return */
  boolean hasNext();

  /** @require hasNext is true
    * @return the next element */
  T next();

}
```

(This is like `Iterator` in Java but with no remove method.)

a. Which of these are true? Circle all that apply.

[✓ `Enumerator<Mammal>` is a true subtype of `Enumerator<Animal>`]

`Enumerator<Mammal>` is a Java subtype of `Enumerator<Animal>`

b. Which of these are true? Circle all that apply.

`Mammal[]` is a true subtype of `Animal[]`

[✓ `Mammal[]` is a Java subtype of `Animal[]`]

c. When passed a `List<Mammal>`, a call to `Collections.unmodifiableList` returns an object of Java type `List<Mammal>` but with a different specification. Is the type of that object a true subtype of `List<Mammal>`? Briefly explain.

No. Its behavior differs when calling mutators.

Problem 11 (Exceptions)

Consider the following code:

```
double y = ... initialize somehow ...

try {
  System.out.println("Answer: " + sqrt(square(y) - 1));
} catch (AssertionError e) {
  System.out.println("Error: |y| < 1");                  // ***
}
```

which references the following methods:

```
/** @returns an approximation of the square root of x
  * @throws AssertionError if x < 0 */
public double sqrt(double x) {
  if (x < 0)
    throw new AssertionError();

  ... calculate square root somehow ...
}

/** @requires |x| < 1e511
  * @returns an approximation of the square of x
public double square(double x) {
  if (Math.abs(x) >= 1e511)  // can't represent square as double
    throw new AssertionError();
  return x*x;
}
```

a. What is the bug affecting the line marked with `***`?

AssertionError is also thrown if y is too big

b. Would this bug be detected by reasoning? If so, how?

call to square(y) is not correct
may not satisfy the precondition

c. *As briefly as possible*, describe the best way to fix this bug?

sqrt should throw something other than AssertionError [3 points]

do a check that y is not too big [2 points]

d. Based on what you have seen in this example, why is it generally a bad idea to throw a `NullPointerException` to indicate an error (other than the error that an object was unexpectedly null)?

you cannot distinguish that condition
from any object on which a method was called being null

Problem 12 (Design Patterns)

a. Which of the following patterns is useful to prevent bugs where == is used in places of `equals`?

Builder Decorator

[✓ Intern] Composite

b. Which of the following patterns is useful to prevent bugs where arguments are passed in the wrong order (which the compiler will not catch)?

[✓ Builder] Decorator

Intern Composite

c. Which of the following is **not** an advantage of static factory methods, compared to calling constructors directly?

they need not create a new object in all cases

they allow you to give names to each method

they can return a subtype of the class

[✓ they can use any one of the superclass constructors to initialize the private superclass fields]

d. Which of the properties of high quality code does Iterator **most** improve?

correctness [✓ changeability]

readability modularity

e. Which of the properties of high quality code does Composite **most** improve?

correctness [✓ changeability]

[✓ readability] modularity

Problem 13 (Miscellaneous)

a. For which of the following return types do you **not** need to consider copying the output returned from your method? Circle one.

[✓ String] ArrayList<?>

String[] HashMap<?, ?>

b. In Effective Java, Josh Bloch suggests a conservative policy for preventing similar methods that might be easily confused. Which of the following did he suggest avoiding? Circle one.

two methods with the same name

[✓ two methods with the same name and number of parameters]

two methods with the same name and parameter types

c. CSE 331 focused on writing code of higher quality and increased complexity by using the following techniques. Which one best describes the use of `@Override`?

[✓ tools] inspection

testing modularity

d. Which of the following is **not** a step that must be performed when "designing for inheritance"? Circle one.

document parts of the RI that may not hold when each method is called

document the pattern of self-calls in the class

do not call non-private methods from the constructor

[✓ provide complete JavaDoc specifications for all private methods]

e. If you do not want to design for inheritance, then you should add which of the following modifiers to the declaration of your class?

private static

[✓ final] synchronized

Bonus Problem (Reasoning)

Call a list of values v_1, .., v_k "stable" if the largest element in the list is no more than 10% larger than the smallest element. Fill in the implementation of the method on the next page, which finds the length of the longest stable subarray of the given array.

The invariant for the loops are provided for you. ***Do not*** add any additional loops.

The provided code uses the `MaxQueue` interface defined before Problem 5, along with a `MinQueue` class that is analogous, with `getMaximum` replaced by `getMinimum`.

The notation "A[i..j]" in the invariants refers to the values A[i], A[i+1], ..., A[j].

Continued on the next page...

```
/**
 * Returns the length of the longest subarray of the given array
 * that is stable.
 *
 * @param A array of numbers to consider
 * @requires A != null and A.length > 0
 * @returns length of longest subarray of A that is stable
 */
public int longestStableSubarray(int[] A) {

  MaxQueue maxQ = new MaxQueue();
  MinQueue minQ = new MinQueue();

  int first = 0;
  int last = -1;
  int maxLen = 0;

  {{ Inv: (1) first <= last+1
          (2) minQ and maxQ contain A[first .. last]
          (3) A[first .. last] is stable
          (4) if first > 0, then maxLen = max length of any stable A[i .. j] with i < first }}
  while (first < A.length) {

    {{ Inv }}
    while (last+1 < A.length &&
           ((first == last) ||
            (vals[last+1] <= 1.1 * minQ.getMinimum() &&
             maxQ.getMaximum() <= 1.1 * vals[last+1])) {
      minQ.enqueue(vals[last+1]);
      maxQ.enqueue(vals[last+1]);


      last = last + 1;
    }

    {{ Inv and first <= last and A[first..last+1] is not stable or does not exist }}

    if (last+1 - first > maxLen)
      maxLen = last+1 - first;

    minQ.dequeue();
    maxQ.dequeue();

    first = first + 1;
  }

  {{ maxLen = max length of any stable subarray of A }}
  return maxLen;
}
```