Solutions to Quiz 1 (March 10)

Problem 1 (Testing Strategies) (24 points).
Given the following spec, write a black box testing strategy for it. Please include a short explanation of your strategy, and write down the test cases (example inputs) that you choose.

```java
/**
 * Given an int array a and an int x, searches for x in a,
 * and returns the location in the array of x.
 * @param a; an integer array
 * @param x; an integer
 * @throws NotFoundException if x is not found in a
 * @throws NullPointerException if a is null
 * @returns i; the location in the array where x was found
 */
```

Solution. Our testing strategy will partition the space of inputs (a and x) so that it covers all the behavior of the specification and also likely bugs, like off-by-one errors. So we’ll choose test cases that cover these conditions:

- a is null, and a is non-null (which triggers the NullPointerException, or doesn’t)
- x appears in a, or x doesn’t appear in a (which triggers the NoSuchElementException, or doesn’t);
- a.length is 0, 1, or greater than 1;
- x appears at the start, middle, or end of a;
- optional: x appears once in a, or multiple times in a.

The spec leaves unclear what happens in the last case, where x occurs more than once. It would be reasonable to omit this from the test suite, or it would be reasonable to include it but make no assumptions about which of the several locations of x is returned. It would not be correct to assume that the method will return a particular location, like the first occurrence. This would assume a stronger spec than was written.

We can avoid partitioning the space of values for x, because the spec doesn’t specify any operations on the value of x. For example, overflow can’t happen when you’re just comparing x for equality.

Example test cases that implement this testing strategy:

- a = null, x = 10; should throw NullPointerException
- a = [ ], x = 10; should throw NoSuchElementException
- a = [5, 10, 15], x = 10; should return 1
- a = [3], x = 3; should return 0
- optional: a = [5, 10, 10], x = 10; should return either 1 or 2
Problem 2 (Static and Dynamic Checking) (20 points).
For each of the methods below, indicate the result of attempting to include it in an otherwise well-formed Java class:

OK  Okay: the Java file containing the class with the method will compile, and the method will run without throwing any exceptions.

RTE  Runtime Error: the Java file containing the class with the method will compile, but when the method is run, it throws an exception.

CTE  Compile-time Error: the Java file containing the class with the method will fail to compile.

Write “OK”, “RTE”, or “CTE” corresponding to the choices above in the margin to the left of each method.

↓↓↓ Write your choices here.

```java
static void test1() {
    List<String> x = new ArrayList<String>();
    x.add("Test");
}

static void test2() {
    ArrayList<String> x = new List<String>();
    x.add("Test");
}

static void test3() {
    ArrayList<String> x = new ArrayList<String>();
    x.add("Test");
}

// Assume x is an ArrayList object.
static void test4(List<String> x) {
    x.add(5);
}

// Assume x is an ArrayList object.
static void test5(List<String> x) throws Exception {
    x.add(5);
}

static void test6() {
    System.out.println(5.2 / 0);
}

static void test7() {
    System.out.println(((int) 5.2) / 0);
}

static String test8() {
    String f = null;
    return f;
}
```
static void test9() {
    String f = null;
    System.out.println(f.equals(null));
}

static void test10() {
    int[] foo = new int[50];
    foo[-1] = 3;
}

Solution. Correct choices:

1 OK
2 CTE
3 OK
4 CTE
5 CTE
6 OK. As we discussed in recitation 1, when floating-point numbers are divided by zero (in Java, Javascript, C, C#, Objective C, and most other languages), you don’t get a runtime error. Instead you get a floating point value representing infinity.
7 RTE
8 OK
9 RTE
10 RTE

Problem 3 (Recursive Descent Parsers) (24 points).
For the following questions, refer to this parser:

```java
public class Parser {
    private char[] input;
    private int i = 0;

    /** Creates a new parser over the given input. */
    public Parser(char[] input) {
        this.input = input;
    }

    public void parse() {
        foo();
        if (i != input.length) {
```
throw new RuntimeException("Char " + i + ": expected end of input");
}
}

private void foo() {
    if (input[i] == '<') {
        i++;
        bar();
        eat('>');
    } else {
        bar();
    }
}

private void bar() {
    if (input[i] == '#') {
        i++;
        baz();
        eat('#');
    } else {
        baz();
    }
}

private void baz() {
    if (input[i] == '<' || input[i] == '#') {
        foo();
    } else if ('a' <= input[i] && input[i] <= 'z') {
        i++;
    } else {
        throw new RuntimeException(
            "Char " + i + ": Got '" + input[i] + ", expected letter."");
    }
}

/** Ensures the next input character is c. */
public void eat(char c) {
    if (input[i] != c)
        throw new RuntimeException(
            "Char " + i + ": Got '" + input[i] + ", expected '/' + c + '/'");
    i++;
}

(a) Write a grammar describing the language that this parser parses. Use Foo, Bar, and Baz as nonterminals in your grammar. You can use any of the usual regular expression operators as well.

Solution. The code can be described by the following grammar:
(b) When the parser finds an invalid string, it throws an exception. Write “E” (for exception) or “OK” next to each of the following strings depending on what that parser will do with them.

______  <#a#>
______  #z#
______  <r>
______  #<f#>
______  #<v>#
______  <>
______  <<c>>
______  <xx>

Solution. Filled-in:
OK  <#a#>
OK  #z#
OK  <r>
E   #<f#>
OK  #<v>#
E   <>
OK  <<c>>
E   <xx>

Problem 4 (Abstract Data Types) (12 points).
Recall that the abstraction function of an abstract data type (ADT) implementation is a function from the concrete state of the class to the abstract state used to specify the ADT. This question considers abstraction functions for a few classes.
In Problem Set 1, you modified an implementation of a `Pair` class. We will reuse `Pair` in this question, for whose purposes you may treat `Pair` as if it were defined like:

```java
public class Pair<X, Y> {
    public final X first;
    public final Y second;

    public Pair(X first, Y second) {
        this.first = first;
        this.second = second;
    }
}
```

To avoid clutter we will treat a `Pair<Integer, Integer>` as interchangeable with the mathematical domain \( \mathbb{Z} \times \mathbb{Z} \). Another simplification is that we pretend Java `int` s are true mathematical integers!

(a) Any correct implementation of the `Set` ADT must be specified in terms of an abstraction function \( f \). Consider an arbitrary \( x \) representing the field values of a `Set<Pair<Integer, Integer>>` implementation associated with \( f \). Which of the following properties could be true, where we apply the usual mathematical notation for sets that lists their elements between curly braces, and where we write pairs as \((n, m)\)? (Mark T or F for each option, indicating whether it is true or false that the property could hold for some valid \( f \) and \( x \).)

- \( f(x) = 0 \)
- \( f(x) = (1, 2) \)
- \( f(x) = \{\} \)
- \( f(x) = \{0\} \)
- \( f(x) = \{(1, 2)\} \)
- \( f(x) = \{0, (1, 2)\} \)

Solution. Filled in:

- \( f(x) = 0 \)  F
- \( f(x) = (1, 2) \)  F
- \( f(x) = \{\} \)  T
- \( f(x) = \{0\} \)  F
- \( f(x) = \{(1, 2)\} \)  T
- \( f(x) = \{0, (1, 2)\} \)  F

(b) Here is an implementation of sets of integers.
// Mutable sets of integers, represented internally as unions of
// contiguous intervals
public class RangeSet {
    private final HashSet<Pair<Integer, Integer>> pairs;
    // Rep invariant: pairs != null

    /* Creates a new set representing the empty set. */
    public RangeSet() {
        pairs = new HashSet<Pair<Integer, Integer>>() {
        }
    }

    /* Creates a new set representing the contiguous interval.
    * @param lower lower bound of interval
    * @param upper upper bound of interval
    * Resulting set contains all n such that lower <= n <= upper.
    * Note that if lower > upper, the result is an empty set.
    */
    public RangeSet(int lower, int upper) {
        pairs = new HashSet<Pair<Integer, Integer>>() {
            pairs.add(new Pair<Integer, Integer>(lower, upper));
        }
    }

    /* Check if a value belongs to the set.
    * @param n value to check for membership
    */
    public boolean contains(int n) {
        for (Pair<Integer, Integer> p : pairs)
            if (p.first <= n && n <= p.second)
                return true;
        return false;
    }

    /* Extend this set with all the values in another set.
    * @param that other set to union in
    * Modifies this set.
    */
    public void unionWith(RangeSet that) {
        pairs.addAll(that.pairs);
    }
}

Let $f$ be the abstraction function associated with HashSet, an (assumed correct) implementation of Set from the Java standard library. We should associate an abstraction function $g$ with RangeSet. Which of the following are correct definitions of $g(p)$, where $p$ is a RangeSet's pairs field value? (Mark T for correct and F for incorrect.)
\[
\{ n \in \mathbb{Z} : p = (u, v) \text{ and } u \leq n \leq v \}.
\]
\[
\{ n \in \mathbb{Z} : p \in f(n) \}.
\]
\[
\{ n \in \mathbb{Z} : n \in f(p) \}.
\]
\[
\{ n \in \mathbb{Z} : \text{there exists } (u, v) \in f(p), \text{ such that } u \leq n \leq v \}.
\]
\[
\{ n \in \mathbb{Z} : \text{there exists } (u, v) \in p, \text{ such that } f(u) \leq n \leq f(v) \}.
\]
\[
\{ n \in \mathbb{Z} : \text{there exists } (u, v) \in p, \text{ such that } u \leq f(n) \leq v \}.
\]

**Solution.** Filled in:

\[
\{ n \in \mathbb{Z} : p = (u, v) \text{ and } u \leq n \leq v \}.
\]
\[
\{ n \in \mathbb{Z} : p \in f(n) \}.
\]
\[
\{ n \in \mathbb{Z} : n \in f(p) \}.
\]
\[
\{ n \in \mathbb{Z} : \text{there exists } (u, v) \in f(p), \text{ such that } u \leq n \leq v \}.
\]
\[
\{ n \in \mathbb{Z} : \text{there exists } (u, v) \in p, \text{ such that } f(u) \leq n \leq f(v) \}.
\]
\[
\{ n \in \mathbb{Z} : \text{there exists } (u, v) \in p, \text{ such that } u \leq f(n) \leq v \}.
\]

\[\blacksquare\]

**Problem 5 (Equality) (20 points).**

Louis Reasoner is designing an immutable datatype representing a Person, reproduced below:

```java
// Person is an immutable class representing a human being
public class Person {
    private String name;
    private Calendar birthday;
    public Person(String name, Calendar birthday) {
        this.name = name;
        this.birthday = birthday;
    }
    public String getName() { return name; }
    public Calendar getBirthday() { return birthday; }
    public boolean equals(Object _that) {
        if (!_that instanceof Person) return false;
        Person that = (Person) _that;
        return this.name.equals(that.name) && this.birthday.equals(that.birthday);    
    }
}
```

Recall that in Java, String is an immutable type and Calendar is a mutable type. Louis is considering the following set of triplets:

Calendar day = new GregorianCalendar(1994, Calendar.MARCH, 14);
Person larry = new Person("Larry", day);
Person darrellA = new Person("Darrell", day);
Person darrellB = new Person("Darrell", day);

Louis makes the following statements. For each one, state whether you AGREE or DISAGREE, and explain your answer in a sentence or two.

(a) “larry != darrellA, but darrellA == darrellB.”

Solution. DISAGREE. All three references point to different objects, so they are all !=.

(b) “larry.equals(darrellA) returns false, but darrellA.equals(darrellB) returns true.”

Solution. AGREE. larry.name.equals(darrellA.name) returns false. But all the parts of darrellA compare equals() to all the parts of darrellB.

(c) “My equals() is consistent over time. If two Person objects are ever equals(), then they will always be equals().”

Solution. DISAGREE. Even though Person is intended to be immutable, it suffers from rep exposure, storing the mutable Calendar object received from the constructor AND returning it as the result of getBirth-day(). So a Person object can actually be mutated.