Solutions to Quiz 1 (February 21, 2014)

Problem 1 (Short Answer) (23 points).
Circle all correct answers for the following questions.

(a) When we implement a function recursively:

A. the postcondition of the function’s spec only restricts the base case of the recursion
B. we cannot use println debugging because recursive calls happen in arbitrary order
C. we cannot include tests that cause OutOfMemoryErrors if they violate the spec
D. changing the implementation to an iterative one will require us to change the spec
E. in Java, recursive calls will increase the depth of the call stack

Solution. C and E.

The postcondition restricts the return value of the function, regardless of whether the base case was reached (A). Recursive calls are not nondeterministic (B). And a good implementation will be hidden behind the spec (D).

(b) Which of the following are part of a function’s specification:

A. return type
B. restrictions on return value
C. number of arguments
D. argument types
E. restrictions on argument values

Solution. A, B, C, D, and E.

A, B, C, and D are statically checked by Java; E is usually not.

(c) Which of the following can be true about a pair of specifications A and B:

A. A can be stronger than B and have a weaker precondition
B. A can be stronger than B and have the same precondition
C. A can be stronger than B and have a stronger precondition
D. A can be stronger than B and have an incomparable precondition
E. A can be incomparable to B

Solution. A, B, and E.

If A has a stronger precondition, either it is weaker than B, or there is not a containment relationship (C). If the preconditions are not comparable, then the specs will not be comparable either (D).

(d) Which of the following are signs of an excellent specification:
A. the specification is declarative
B. the specification is operational
C. the implementation is allowed to ignore invalid arguments
D. the implementation is allowed to use different algorithms depending on the arguments
E. the specification utilizes the reader’s knowledge of the implementation

**Solution. A, D.**

We prefer declarative specs to operational ones (B). We would rather have clear specs and implementations that fail fast than allow the implementation to quietly fail (C). And we do not want the client to have to read the implementation at all (E).

(e) In 6.005, your software should be:

A. easy to understand
B. ready for change
C. devoid of test cases
D. safe from bugs
E. documented in Klingon

**Solution. A, B, and D.**

ngoq waH? puj programmer.

**Problem 2 (Code Review) (20 points).**

Consider the following method:

```java
public static final String f(int a) {
    int MAJOR = 6;
    if (a == MAJOR) {
        return "OK";
    } else {
        return "." + f(a-1);
    }
}
```

This code was code-reviewed, producing the comments below. **Circle AGREE or DISAGREE** depending on whether the comment is correct or incorrect, and **add your own one-sentence comment** explaining your answer. The right explanation is worth more than the right circle.

(a) “Since the MAJOR variable is used only once in this method, you should get rid of it and just say a == 6 instead.”

AGREE   DISAGREE

**Solution. DISAGREE.** The name MAJOR helps make the code easier to understand, since 6 would otherwise be a magic number.

(b) “This method needs to state a precondition, because it doesn’t work for all inputs.”

AGREE   DISAGREE
Solution. AGREE. If \( a < 6 \), then the recursion doesn’t stop. (In practice, either \( a \) will overflow and wrap around from negative MAX_INT to positive MAX_INT, or the call stack will overflow with too many recursive calls.)

(c) “But adding a precondition would mean that if you call \( f(66) \), the function would check whether to throw an exception about 60 times, not just once.”

AGREE DISAGREE

Solution. DISAGREE. Preconditions don’t have to be checked by the method. For defensive programming, it’s good to check preconditions though, and the statement is correct about how many times a recursive function would check them.

(d) “This method is not safe from bugs because it uses global variables.”

AGREE DISAGREE

Solution. DISAGREE. None of the variables here (MAJOR, a, b) are global.

Problem 3 (Recursion) (19 points).

Consider the printSubsequences method below, implemented using a private helper.

Fill in the missing code to complete the recursive function. In doing so, the following String methods may be useful:

- int length() – returns the length of the String
- char charAt(int x) – returns the character at position \( x \) in the String
- String substring(int x) – returns a substring by omitting the first \( x \) characters

```java
/**
 * Print all the subsequences of String s of length n, for n >= 0.
 * For example: printSubsequences("wxyz", 3) prints "wxy", "wxz", "wyz", "xyz"
 */
public static void printSubsequences(String s, int n) {
    findAndPrintSubseqs("", s, n);
}

private static void findAndPrintSubseqs(String subseq, String rest, int n) {
    // shift characters from rest into subseq while decreasing n
    if (n == 0) {
        // when n == 0, subseq has accumulated a subsequence of length n
        System.out.println(subseq);
        return;
    }

    if ( ______________________ ) return;

    findAndPrintSubseqs( ______________________, ______________________, __________ );
```

```java
```
findAndPrintSubseqs( ______________________, ______________________, __________ );
}

**Solution.** We shift characters from `rest` into `subseq`. In one recursive call, we pick a character; in the other, we skip it.

```java
private static void findAndPrintSubseqs(String subseq, String rest, int n) {
    if (n == 0) {
        System.out.println(subseq);
        return;
    }

    if (rest.length() == 0) return;

    findAndPrintSubseqs( subseq + rest.charAt(0) , rest.substring(1) , n - 1 );
    findAndPrintSubseqs( subseq , rest.substring(1) , n );
}
```

---

**Problem 4 (Static Checking) (12 points).**
For each of the following pieces of code, determine the result of writing the code in a Java program.

- Circle **OK** if the code will compile and run without any exceptions.
- Circle **Compile-time Error** if the code will fail to compile.
- Circle **Runtime Error** if the code will compile but will throw an exception when run.

(a) `int[] arr = new int[] { 1, 2 };`  
   `arr[2] = 3;`
   
   OK       Compile-time Error       Runtime Error

**Solution.** Runtime error `ArrayIndexOutOfBoundsException`. Java will check the array bounds at runtime.

(b) `int[] arr = new int[] { 1, 2 };`  
   `arr[0] = "2";`
   
   OK       Compile-time Error       Runtime Error

**Solution.** Compile-time error. The compiler rejects assignment of `String` to `int`.

(c) `int[] arr = new int[] { 1, 2 };`  
   `arr[0] = 2;`
   
   OK       Compile-time Error       Runtime Error

**Solution.** OK. A normal array assignment.

(d) `String s = null;`  
   `System.out.println(s == null);`
Solutions to Quiz 1 (February 21, 2014)

OK  Compile-time Error  Runtime Error

Solution. OK. We can check if a variable is null, although avoiding null would be better.

(e) String s = null;
    System.out.println(s.length());

OK  Compile-time Error  Runtime Error

Solution. Runtime error NullPointerException. null is a hole in the type system, passing the compiler’s static checks.

(f) String s = null;
    System.out.println(s);

OK  Compile-time Error  Runtime Error

Solution. OK. Printing out null is supported.

Problem 5 (Testing) (26 points).
Consider the following specification:

```java
/**
 * Reverses the end of a string.
 *
 * For example: reverseEnd("Hello, world", 5) returns "Hello dlrow ,"
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text non-null String that will have its end reversed
 * @param start the index at which the remainder of the input is reversed, requires 0 <= start <= text.length()
 * @return input with the substring from start to the end of the string reversed
 */
public static String reverseEnd(String text, int start)
```

(a) Which of the following inputs should we include in our test suite? Circle all the valid test cases.

1. null, 0
2. "abba", 1
3. "abracadabra", 12
4. "", 0
5. "a", 0

Solution. ANSWER: 2, 4, and 5 are all valid.

1 violates the precondition that text is non-null.

3 violates the precondition that start is <= text.length().
(b) We want to test `reverseEnd` systematically. We will partition the spaces of the inputs, then select test inputs from the Cartesian product of these partitions.

Fill in 3 partitions for `text` down the side of the table below.

Fill in 4 partitions for `start` across the top. You may define partitions for `start` in terms of `text.length()`. Then, choose test inputs, placing them at the appropriate intersection of partitions.

- If a box can be correctly filled with a valid test case from part (a) above, you must use that test.
- You may also devise additional test cases. **In those tests, use only the letters w, x, y, and z in `text`**.
- You may write the same test inputs in more than one box if appropriate.
- You may leave boxes blank if appropriate.

<table>
<thead>
<tr>
<th>partitions for <code>start</code></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions for <code>text</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;, 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;a&quot;, 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;wxy&quot;, 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution. We test boundary conditions for both `text` and `start`.

Boxes are blank when the precondition cannot be satisfied.

<table>
<thead>
<tr>
<th><code>text</code></th>
<th><code>start</code></th>
<th><code>start &gt; 0 and start &lt; len-1</code></th>
<th>length −1</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>0</td>
<td></td>
<td></td>
<td>&quot;&quot;, 0</td>
</tr>
<tr>
<td>length == 1</td>
<td>&quot;a&quot;, 0</td>
<td></td>
<td>&quot;a&quot;, 0</td>
<td>&quot;a&quot;, 1</td>
</tr>
<tr>
<td>length &gt; 1</td>
<td>&quot;wxy&quot;, 0</td>
<td>&quot;abba&quot;, 1</td>
<td>&quot;wxy&quot;, 2</td>
<td>&quot;wxy&quot;, 3</td>
</tr>
</tbody>
</table>

Other partitioning strategies were also accepted.