CSE 331 Winter 2019 Midterm Exam

Name _____

UW Email: _____ @uw.edu

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: _____ / 65

1. _____ / 10
2. _____ / 10
3. _____ / 10
4. _____ / 12
5. _____ / 11
6. _____ / 12

Bonus: _____ / 4

Problem 1 (Reasoning I)

Consider the following code, which takes an integer x as input and returns the smallest power of 10 that is larger than x. For example, if x = 12, it would return 100.

```
{{ x > 0 }}
int nextPowerOf10(int x) {
  int k = 1;

  {{ Inv: k is a power of 10 and satisfies k <= 10x }}
  while (x >= k)
    k = 10 * k;

  {{ k is a power of 10 and satisfies x < k <= 10x }}
  return k;
}
```

Answer the following questions to explain why the loop is correct. Be concise. *Please!* (All three parts have short answers that are fully correct.)

Why does the loop invariant hold initially?

Why does the body of the loop preserve the loop invariant?

Why is the postcondition true when the loop exits?

Problem 2 (Reasoning II)

Fill in an implementation of the method `removeDups`. It takes as input a sorted array `a` containing at least `n` integers and writes into `b` the set of *distinct* elements from `a`, also in sorted order.

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.

```
{{ P: a, b != null and 0 < n <= a.length, b.length }}
int removeDups(int[] a, int[] b, int n) {
  int i =
  int j =

  {{ Inv: P and b[0], ..., b[i] holds the distinct elements of a[0], ..., a[j] in sorted order
          and i, j >= 0 }}

  while (_____________________________________________________) {

    j = j + 1;
  }

  {{ b[0], ..., b[i] holds the distinct elements of a[0], ..., a[n-1] in sorted order }}
  return i+1;
}
```

Bonus: Why is the second line of the loop invariant necessary? Specifically, what could go wrong in the proof if we removed that?

Problem 3 (Specifications)

Complete in the JavaDoc documentation for the `removeDups` method from the previous page. You may skip the `@param` tags.

```
/**
 * Copies the unique elements from the sorted array stored in
 * a[0], a[1], ..., a[n-1] into the beginning of the array b.
 *
 * @param(s) ... omitted ...
 *
 *
 *
 *
 *
 *
 *
 *
 */
public int removeDups(int[] a, int[] b, int n)
```

Suppose that the author wants to change the specification so that `a` is no longer required to be sorted. Instead, she will sort the array herself in `removeDups`. However, suppose that she also wants to leave herself room to change the implementation in the future to no longer use sorting (perhaps she will use a hash table instead).

How would she change the specification above for this scenario? You only need to write the lines that should change.

How does this new specification relate to the first one above?[1]

stronger weaker incomparable

Suppose that author wanted to change the method to *ignore* the value of the input parameter `n` and instead use the length of the array `a` in its place.

How would the specification for that method relate to the first one above?

stronger weaker incomparable

[1] If you don't remember the right word, just say whether implementations of the new specification necessarily satisfy the first specification or vice versa or neither.

Problem 4 (ADTs)

Consider the following class:

```
/** Represents a rectangle with positive area. Each one can
  * be thought of as a pair (width, height), containing the width
  * and height of the rectangle, respectively. For example, the
  * pair (10, 5) is a rectangle with width 10, height 5, and
  * hence, an area of 50 (since area = width * height). */
public class Rectangle
```

The author expects the usage of the class to be dominated by calls to get the area, so she decides to directly store the area in her concrete representation.

Fill in the documentation of this representation below:

```
// RI:
//
// AF(this) =
//
private int width;
private int area;
```

Give an implementation of the `equals` method for this class that is not only correct but also follows the usual Java idiomatic form for `equals`:

Give an implementation of the `hashCode` method that satisfies the normal specification but **does not distinguish** all `Rectangle`s that are unequal.

Problem 5 (Testing)

Consider the following method:

```
/** @requires 0 <= x <= 59
  * @return "top" if x is < 15 or >= 45 and otherwise "bottom"
String halfOfHour(int x)
```

Fill in empty templates below to describe three tests for the method above that are from distinct subdomains according to the **specification** testing heuristic:

On input x = _____10_____

Expect output _____"top"_____

On input x = _____

Expect output _____

On input x = _____

Expect output _____

List **six** different inputs that should be tested for checking important **boundary cases** (just list the values for "x" not the expected outputs):

(problem continued on the next page...)

Suppose that the method is implemented as follows:

```
String halfOfHour(int x) {
  if (x / 10 == 0 || x / 10 == 5) {
    return "top";
  } else if (x / 10 == 1) {
    if (x - 10 < 5)
      return "top";
    else
      return "bottom";
  } else if (x / 10 == 4) {
    if (x - 40 < 5)
      return "bottom";
    else
      return "top";
  } else {
      return "bottom";
  }
}
```

Fill in the templates below to describe six tests for the method above that are from distinct subdomains according to the **implementation** testing heuristic:

On input x = _____

Expect output _____

On input x = _____

Expect output _____

On input x = _____

Expect output _____

On input x = _____

Expect output _____

On input x = _____

Expect output _____

On input x = _____

Expect output _____

Problem 6 (Miscellaneous)

Write a **one sentence** (or shorter) answer to each of the following questions.

1. Describe a bug that would be caught by adding the `@Override` annotation.

2. Describe a bug that would be caught by adding a call to `checkRep` at the beginning of your methods (even when it was already called at the end of those methods).

3. Describe a bug that cannot occur if you never define a method where adjacent arguments (in the arguments list) have the same type.

4. Describe a bug that cannot occur if, instead of returning a reference to an object held in a private field of your class, you return a reference to a fresh copy.

5. Describe a bug that cannot occur if you only use immutable objects as the keys in a `Map` collection.

6. The `.test` files of many student submissions for HW5 part 1 contained typos (e.g., writing "`ListNode`" instead of "`ListNodes`"). Which of these approaches discussed in class would be the best way to eliminate those mistakes? (Circle one.)

tools	inspection
testing	defensive programming