CSE 331 Midterm Exam 5/12/17   **Sample Solution DRAFT**

Remember: For all of the questions involving proofs, assertions, invariants, and so forth, you should assume that all numeric quantities are unbounded integers (i.e., overflow can not happen) and that integer division is truncating division as in Java, i.e., 5/3 => 1.

Question 1. (12 points) (Forward reasoning) Using forward reasoning, write an assertion in each blank space indicating what is known about the program state at that point, given the precondition and the previously executed statements. Your final answers should be simplified. Be as specific as possible, but be sure to retain all relevant information.

(a) (5 points)

```
{ a > 1 }
b = a + 3;
{ a > 1 && b > 4 }
a = a + 2;
{ a > 3 && b > 4 }
a = 2 * a + b;
{ a > 12 && b > 4 }
```

Note: In the final assertion, since a>3, we know a>=4, which means 2*a>=8 and 2*a+b>12. When grading we didn't deduct for answers that had a>10 or a>11, since those are basically correct, even if not quite as strong as possible.

(b) (7 points)

```
{ |y| > 2 }
if (x > 0)
   { x > 0 && |y| > 2 } => { x > 0 && (y>2 | y<-2) }
   y = y + x;
   { x > 0 && (y > 3 | y has any value) } => { x > 0 }
else
   { x <= 0 && (y > 2 | y < -2) }
   y = y - x;
   { x <= 0 && (y>2 | y has any value) } => { x <= 0 }
{ (x > 0 | x <= 0 } => { true }
```

Note: The final answer is unusual in that we really can't know anything about the final values of either `x` or `y`, but it does follow from the assertions.

Question 2. (12 points) (assertions) Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your final answers if possible.

(a) (5 points)

```
{ 2*p+1 > 0 && 2*p-q > 0 } => { 2*p > -1 && 2*p > q }
          => { p >= 0 && 2*p > q }
p = 2 * p;
{ p+1 > 0 && p-q > 0 }
q = p - q;
{ p+1 > 0 && q > 0 }
p = p + 1;
{ p > 0 && q > 0 }
```

(b) (7 points)

```
{ (a < b && a > 0) || (a >= b && a > b+1) }
    => { (a < b && a > 0) || (a > b+1) }
if (a < b) {
  { a+b > b } => { a > 0 }
  a = a + b;
  { a > b }
} else {
  { a > b +1 }
  b = b + 1;
  { a > b }
}
{ a > b }
```

Question 3. (14 points) Consider the following binary search implementation. Answer questions about this code on the next page, but *do not* remove this page from the exam since you need to add assertions to this code.

```
// return location of x in data if found, or -1 if not
// pre: data is sorted (data[0] <= data[1] <= ...)
public static int binSearch(int[] data, int x) {
  int low = 0;
  int high = data.length - 1;
  int mid = 0;
  boolean found = false;  { if found=true then we have
                             discovered that data[mid]=x }
  { inv }
  while(low <= high && !found) {
    { inv && low <= high && !found }
    mid = (low + high) / 2;
    if(data[mid] == x) {
      { inv && data[mid]=x }
      found = true;
      { inv && data[mid]=x && found }
    } else if(data[mid] < x) {
      low = mid + 1;
      { inv && !found }
    } else { // data [mid] > x
      high = mid - 1;
      { inv && !found }
    }
    { inv && ((found && data[mid]=x) || !found) }
  }
  { inv && (low > high || found) &&
                   ((found && data[mid]=x) || ! found) }
  if (found) {
    { found && data[mid]=x }
    return mid;
  } else {
    { !found }
    return -1;
  }
}
```

(continued on next page)

Question 3. (cont) We think this code "works" since it seems to produce reasonable answers when run on some test data, but we want to be sure by proving it is correct. Without modifying the code, answer the following questions:

(a) (6 points) Give a suitable invariant for the `while` loop. This is an assertion that would be true at least at the point labeled with `{inv}` in the above code, also at the very beginning of the loop body, at the end of the loop body, and possibly elsewhere. The invariant should include whatever is needed to analyze the code and prove it is correct. It will need to describe the relationships between the key variables, the contents of the array, and the value `x` that is being searched for in the array. You may assume the array is sorted (the precondition of the method). Write the invariant below:

The invariant needs to include

`{ data[0..low-1] < x && data[hi+1..data.length-1] > x }`

It's not clear how to handle `found`. In order to prove that we return the correct result, we need to know when the loop terminates that we can conclude `{found => data[mid]=x}`. So it seems that the loop invariant should capture information about `found` by including something like this:

`{ !found | (found && data[mid]=x) }`

But `found` is part of the loop condition (e.g., `while (`*B*`)`), which normally is not part of the invariant. So maybe this information belongs as an assertion in the definition of `found` and not as part of the invariant proper, and then we can use that assertion along with `{inv && !B}` after the loop terminates to show that the method returns the right result.

What should be clear from all this is that "binary search with exit in the middle if you find x" has a complicated proof. In grading the question we tried to give credit to solutions that captured the essential things needed to show the algorithm works and presented them clearly, without being excessively picky about the details.

(b) (8 points) Prove that the binary search algorithm given on the previous page works as specified, i.e., it returns the location of `x` if `x` is present in the `data` array, and if it is not found returns -1. You need to add appropriate assertions in the code on the previous page. Because of the complexity of the algorithm and invariant you can be somewhat informal and don't need to write down every intermediate assertion, but do not skip any significant steps (and, of course, don't make misteaks).

See previous page.

Question 4. (10 points, 2 each) Testing. Describe five distinct black-box tests that could be used to verify that the binary search method from the previous problem works properly. Each test description should describe the test input and expected output. The input data array for your tests should be sorted properly (i.e., the appropriate precondition for the method must hold). For full credit each test should be different in some significant way from the other tests.

Here are several possible tests:

Check for found in middle of array (single copy of value):
Input: data = { -2, 0, 1, 4, 6, 6, 10 }, x = 4
Expected output: 3

Check for found in middle of array (multiple copies of value):
Input: data = { -2, 0, 1, 4, 6, 6, 6, 10 }, x = 6
Expected output: 5 or 6 or 7 (all are possible since the spec. does not guarantee which duplicate value would be located, and a black box test cannot examine the implementation to know exactly which answer to return)

Check for found at the beginning of array:
Input: data = { -2, 0, 1, 4, 6, 6, 10 }, x = -2
Expected output: 0

Check for found at end of array:
Input: data = { -2, 0, 1, 4, 6, 6, 10, 10 }, x = 10
Expected output: 6 or 7

Check for not found in middle of array:
Input: data = { -2, 0, 1, 4, 6, 6, 10 }, x = 5
Expected output: -1

Check for not found because smaller than all array elements:
Input: data = { -2, 0, 1, 4, 6, 6, 10 }, x = -10
Expected output: -1

Check for not found because larger than all array elements:
Input: data = { -2, 0, 1, 4, 6, 6, 10 }, x = 331
Expected output: -1

Obviously there are many other possible tests that differ from the above, including arrays with 0 or 1 elements, larger arrays with significantly different sets of values, and so forth.

The next few questions concern this simple stack ADT which we would like to clean up so we can add it to one of our projects. The code works ok, but the documentation is terrible. You can remove this page from the exam for reference while working on the following questions.

```
public class IntStack {
  private int[] vals;  // stack
  private int top;     // top
  private static int defaultCapacity = 100;

  // construct new stack with default capacity
  public IntStack() {
    vals = new int[defaultCapacity];
    top = 0;
  }

  // construct new stack with given capacity
  public IntStack(int capacity) {
    vals = new int[capacity];
    top = 0;
  }

  // operations
  public boolean push(int x) {
    if (top == vals.length)
      return false;
    vals[top] = x;
    top++;
    return true;
  }

  public int pop() {
    if (top == 0) { // throw runtime exception if empty
      throw new NoSuchElementException();
    }
    top--;
    return vals[top];
  }

  public int size() {
    return top;
  }
}
```

Question 5. (12 points) (a) (3 points) Give a suitable abstract description of the class as would be written in the JavaDoc comment above the `IntStack` class heading.

An `IntStack` is a finite stack of integers with a fixed capacity. A typical value would be s0, s1, s2, …, sn, where s0 is at the bottom of the stack and sn is at the top.

(b) (5 points) Give a suitable Representation Invariant (RI) for this class. (Remember that this RI should be sufficient to guarantee that the existing code executes successfully.)

`vals` is not `null`, `top` >= 0, and for 0 <= `k` < `top`, `vals[k]` has been initialized with stack elements.

Note: it would be incorrect to say that `vals[k]` is not `null`, since `int` values cannot be `null` – they are not references.

(c) (4 points) Give a suitable Abstraction Function (AF) for this class relating the representation to the abstract value of a `IntStack`.

`vals[0..top-1]` represents the stack of values, where `vals[0]` is the bottom element of the stack (s0), `vals[top-1]` is the top element on the stack (sn), and `top` is the number of items on the stack. If `top=0`, the stack is empty.

Question 6. (12 points) Specification. None of the methods in the `IntStack` are specified properly. Below, supply proper JavaDoc comments for the push and pop methods for the code on previous pages, including a summary description at the beginning of each JavaDoc comment. Leave any unneeded parts blank

```
 /**
  * Push a value onto this IntStack if there is space for it
  *
  * @param x value to be pushed
  *
  * @requires
  *
  * @modifies this
  *
  * @effects x is pushed onto the top of the stack if the
  *          stack is not yet full
  *
  * @throws
  *
  *
  * @returns true if x was successfully pushed, false if not
  *
  */
public boolean push(int x) { implementation omitted }

 /**
  * Return the top value from this stack and remove (pop) it
  *
  * @param
  *
  * @requires
  *
  * @modifies this
  *
  * @effects top element popped (deleted) from stack
  *
  * @throws NoSuchElement exception if this stack is empty
  *         when pop is called
  *
  * @returns element popped from the stack
  *
  */
public int pop() { implementation omitted }
```

Note: The most common errors in this problem were specifications that included information about `top` or `vals`. Those are private implementation details that should not appear in the public specification of the `IntStack` ADT.

Question 7. (10 points) After your successful experience in CSE 331, you've been hired as a TA and a student has come in for help with a bug in their hw5 code. They have done a good job of reducing the buggy code to the following test program:

```
import java.util.*;

public class Graph {

  private class Node {
    public final String label;
    public Node (String label) {
      this.label=label;
    }
  }

  private Set<Node> nodes = new HashSet<Node>();

  public void addNode(String label) {
    nodes.add(new Node(label));
  }

  public int getNumNodes() {
    return nodes.size();
  }

  public static void main (String[] args) {
    Graph g = new Graph();
    g.addNode("Waffles");
    g.addNode("Pancakes");
    g.addNode("Pancakes");
    g.addNode("Maple Syrup");
    System.out.println(g.getNumNodes());
  }
}
```

The trouble is that when they run this code it prints `4`, even though there should only be 3 distinct nodes in the set of nodes in the Graph.

Answer questions about this program on the next page. You should leave this page in the exam in case you need to mark corrections on the code above.

(continued on next page)

Question 7. (cont.) (a) (4 points) What's the bug? i.e., what is the most likely reason that the main program is printing 4 when there are only three different labels on nodes in the graph? Give a brief explanation (no more than a couple of sentences).

The `Node` class does not override `equals` and `hashCode`, so two different nodes with the same label are not treated as equivalent when they are inserted into the `HashSet`.

(b) (6 points) Describe how to fix the bug. If a small amount of changed or new code is needed, or if some code should be deleted from the original program, you can show the corrections on the previous page. If more extensive additions or corrections are needed, write them below but be sure it is clear from your description where any new or changed code should be placed in the existing code. You should not make fundamental changes to the data structures or operation of the existing code – just fix it.

Add these two methods to class `Node`. The implementation of `hashCode` can be different from the one here as long as two `Node`s that are equal have the same `hashCode`.

```
@Override
public boolean equals(Object other) {
  if (! (other instanceof Node))
    return false;
  Node n = (Node)other;
  return this.label.equals(n.label);
}

@Override
public int hashCode() {
  return this.label.hashCode();
}
```

Question 8. (10 points) Comparing specifications. Here are four possible specifications for a method that computes whether its integer parameter *n* is a perfect number. (It doesn't really matter for the question, but a perfect number is a positive integer whose value is equal to the sum of its proper positive divisors. For example, 6 is a perfect number because 6 = 1+2+3.)

Here are four possible specifications for a method that has a parameter `n`.

A.
```
@param n
@returns true if n is a perfect number, otherwise false
```

B.
```
@param n
@requires n > 0
@returns true if n is a perfect number, otherwise false
```

C.
```
@param n
@throws IllegalArgumentException if n <= 0
@returns true if n is a perfect number, otherwise false
```

D.
```
@param n
@requires n > 0
@returns true if n is a perfect number
@throws RuntimeException if n > 0 but n is not a
                                          perfect number
```

(a) List all of the specification that are stronger than A. **<u>none</u>**

(b) List all of the specification that are stronger than B. **<u>A, C</u>**

(c) List all of the specification that are stronger than C. **<u>none</u>**

(d) List all of the specification that are stronger than D. **<u>none</u>**

(e) Is it possible for a single method to satisfy A and C? (yes or no) **<u>no</u>**

(f) Is it possible for a single method to satisfy B and C? (yes or no) **<u>yes</u>**

Question 9. (6 points, 1 each) Mother's day is this Sunday and we've been working on an app to help people send last-minute gifts to Mom. First, we have a class hierarchy of possible things to send:

```
class Gift { ... }
class Flower extends Gift { ... }
class Rose extends Flower { ... }
class Violet extends Flower { ... }
```

Next, we have a basic class with a method that sends a gift:

```
class Delivery {
  Gift send(Flower x, Flower y) { ... }
}
```

We would now like to add a new class for Mother's day:

```
class MomDelivery extends Delivery { ... }
```

Listed below are several possible `send` methods we might want to include in this new class `MomDelivery`. For each method circle the right choice to indicate whether it *overrides* the send method inherited from `Delivery`, or *overloads* it, or causes the Java compiler to indicate a type *error*, or is *none of the previous* (i.e., there are no errors, but it doesn't overload or override the `Delivery send` method). Your answers should use the Java rules for subtypes and method overloading/overriding, even if true specification subtyping would give a different answer.

(a) `Gift send(Rose x, Violet y) { ... }`

[✓ overload] override error none of the previous

(b) `Flower send(Flower x, Flower y) { ... }`

overload [✓ override] error none of the previous

(c) `Object send(Flower x, Flower y) { ... }`

overload override [✓ error] none of the previous

(d) `Gift send(Flower x, Flower y) { ... }`

overload [✓ override] error none of the previous

(e) `Gift send(Gift x, Gift y) { ... }`

[✓ overload] override error none of the previous

(f) `Gift send(Flower x) { ... }`

[✓ overload] override error none of the previous

Question 10. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

(b) (1 point) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

No opinion / don't care

None of the above. My answer is ________________________.