CSE 331 Midterm Exam 11/9/15 Sample Solution

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. (10 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)

```
{ x < -3 && y = x }
x = x - 4;
{ x < -7 && y < -3 }
y = x + abs(x);
{ x < -7 && y = 0 }
z = (y+5) * (x+2);
{ x < -7 && y = 0 && z < -25 }
```

(b)

```
{ |x| < 5 }
if (x > 0)
   { |x| < 5 && x > 0 => 0 < x < 5 }
   y = x + 2;
   { 0 < x < 5 && 2 < y < 7 }
else
   { |x| < 5 && x <= 0 => -5 < x <= 0 }
   y = x - 1;
   { -5 < x <= 0 && -6 < y < 0 }

{ (0<x<5 && 2<y<7) || (-5<x<=0 && -6<y<0) }
=> { |x|<5 && (-6<y<0 || 2<y<7) }
```

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*(-7)+y > 0 => (-14)+y > 0 => y > 14 }
x = -7;
{ 2*x+y > 0 }
z = 2 * x + y;
{z > 0}
```

(b) (7 points)

```
{ (x>0 && -12<=x<=0) || (x<=0 && -12<=x<=12) }
 => { false || (-12 <= x <= 0) } => { -12 <= x <= 0 }
if (x > 0) {
   { -12 <= x <= 0 }
   x = x + 6;
   { -6 <= x <= 6 }
} else {
   { -12 <= x <= 12 }
   x = x / 2;
   { -6 <= x <= 6 }
}
{ |x| < 7 }
```

Question 3. (16 points) Loops. The following method takes two integer arrays and is supposed to store in each element `a[i]` the maximum of `a[i]` and `b[i]`. Your job is to complete the code and provide a proof that the code works as specified. You need to supply an appropriate loop invariant, write the rest of the code, and provide assertions as needed to prove it is correct.

Notation: we suggest using `a[k]` for the current contents of an array element and `A[k]` for the original value of that element, and similarly for `b` (although you can write `a[k]_old` if you really want). To reduce writing, you do not need to write `a!=null`, `b!=null`, and `a.length==b.length` anywhere. Assume that those are always true.

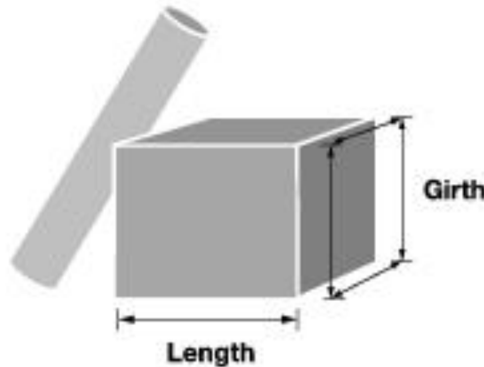
```
// set a[i] to max(a[i], b[i]) for all 0<=i<a.length.
// pre: a != null && b != null && a.length == b.length.
static void pairwisemax(int a[], int b[]) {
  // add initialization code and assertions as needed
   i = 0;
   { inv: 0<=k<i: a[k] = max(A[k],B[k]) }
   while ( i != a.length ) {
     { inv && i != a.length }
     if (a[i] < b[i]) {
        { inv && a[i] < b[i] }
        a[i] = b[i];
        { 0<=k<=i: a[k] = max(A[k],B[k]) }
     else
        { 0<=k<=i: a[k] = max(A[k],B[k]) }
     i = i + 1;
     { inv }
   }  // end of while loop
   { inv && i == a.length } =>
   { 0<=k<a.length && a[k]=max(A[k],B[k]) }
   { post: for 0 <= k < a.length, a[k] = max(A[k], B[k]) }
}
```

Note: The solution shows the essential parts of the proof that were needed to get full credit. Among other things, it doesn't repeat that unexamined elements of a and all of b retain their original values. It's fine if you included additional details, and those would be part of a complete, formal proof, but this is enough for an exam question.

The holidays are almost here and that means that many packages will be shipped during the next two months. The next several questions involve an ADT for describing packages shipped by the USPS (US Postal Service). First some background.

The USPS has standards and rules about the dimensions of parcels that can be shipped. Two particular rules that are relevant here are:

1. A parcel has to be at least 3 inches high, 6 inches long, and ¼ inch thick.
2. A parcel "cannot measure more than 108 inches in length and girth combined". Length is the measurement of the longest dimension and girth is the distance around the thickest part of the parcel perpendicular to the length.



The following questions concern a Java class `USPSParcel` that represents one of these packages. Each instance of this class holds information about the dimensions of the parcel and also has a tracking history showing where and when the parcel was processed from initial shipment to final delivery.

More specifically, the internal representation of a parcel must include the following:

- **length:** the longest dimension of the parcel
- **thickness:** the shortest dimension of the parcel
- **height:** the other dimension of the parcel that is neither the longest nor shortest.

These dimensions should be stored as `doubles` giving dimensions in inches.

The tracking history is simply an ordered list of non-null `String` values including date, time, and location of specific events like "12/22/15 16:30 North pole, received by carrier", "12/24/15 17:00 Seattle, arrived at carrier facility", "12/24/15 23:30 Seattle, delivered, left in front of fireplace". In the problems below we will not use the contents of these strings, so their format is not specified further.

(You can remove this page for reference while working on the following questions.)

Question 4. (12 points) Here is part of the Java code for the USPSParcel class specification, giving declarations for the instance variables implied above.

```
/** abstract description of USPSParcel here (see part(a)) */
public class USPSParcel {
  private final double length;     // rep: package dimensions
  private final double thickness;
  private final double height;
  private List<String> history;    // rep: tracking history
```

(a) (3 points) Give a suitable abstract description of this class as would be written in the JavaDoc comment above the `USPSParcel` class heading. A `USPSParcel` should represent a parcel that meets the USPS requirements for shipping (has suitable sizes).

A USPSParcel represents a package that meets the USPS standards for one that can be shipped. A USPSParcel has a length, height, and thickness where length >= height >= thickness and length >= 6, height >= 3, and thickness >= 0.25. In addition the girth (twice the height+thickness) plus length must be <= 108.

In addition, a USPSParcel contains a tracking history, which is an ordered list of non-null string values e1, e2, …, en giving events as the package was shipped, with e1 being the earliest event and en the latest so far.

(b) (5 points) Give a suitable Representation Invariant (RI) for this class. Besides constraints on the parcel dimensions, you also need to provide any necessary constraints on the `history` variable storing the tracking history.

RI: length >= 6.0 && height >= 3.0 && thickness >= 0.25 &&
length >= height >= thickness && (length+2*(height+thickness))<=108 &&
history != null && for all strings s in history, s!=null.

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

Variables length, height, and thickness contain those dimensions of the parcel, and for 0<=k<history.size(), history.get(k) is tracking event number k+1.

Question 5. (12 points) Complete the following partial implementation of the `USPSParcel` class constructor. The input parameter is an array of three doubles containing the dimensions in no particular order (clerks are lazy and enter the dimensions in the order they measure them). Hints: `Arrays.sort(a)` sorts array `a` in ascending order, and `Arrays.copyOf(a,n)` yields a copy of array `a` with `n` elements.

```
/** Initialize this USPSParcel
 * @param sizes package dimensions in no particular order
 * @requires sizes.length == 3
 * @modifies this
 * @effects this initialized with given package size and
 *          empty tracking history
 * @throws IllegalArgumentException if parcel dimensions
 *         do not meet USPS requirements for shipping
 */
public USPSParcel(double[] sizes) {

  assert sizes.length ==3 :
               "incorrect number of dimensions";

  double dim[] = Arrays.copyOf(sizes, sizes.length);
  Arrays.sort(dim);
  length = dim[2];
  height = dim[1];
  thickness = dim[0];

  if (length < 6.0 || height < 3.0 || thickness < 0.25)
    throw new IllegalArgumentException(
                        "parcel dimension too small");
  if (length + 2.0 * (height + thickness) > 108.0)
    throw new IllegalArgumentException(
                        "parcel length+girth too large");

  history = new ArrayList<String>();

}
```

Note: the assert statement at the beginning of the constructor is good practice, but not required for this problem. Similarly, there should be a `checkRep()` call at the end of any CSE 331 constructor, but if it was omitted on the exam we did not penalize that. When constructing exceptions it was fine to include an error message, as above, or to omit it. Either version received credit.

Question 6. (testing) (12 points) Describe four separate "black box" tests for your `USPSParcel` constructor (above). At least one test should verify that the constructor properly initializes an object representing a parcel that can be legally shipped (has appropriate dimensions). At least two tests should check that the constructor throws an appropriate exception if an attempt is made to construct a package that does not meet the requirements. For full credit, the tests should cover different input subdomains.

For each test give the input values and expected result(s). You do not need to write JUnit tests or other Java code, although you can if you wish. If you need them, you should assume that the class includes methods `getLength()`, `getHeight()`, `getThickness()`, and `getHistory()` that return values in the instance variables.

There are obviously a huge number of possible answers to this question; a few sample ones are given here.

(a) **Create a parcel with legal dimensions (6, 12, 1). Verify that the constructor creates the object successfully, getLength() == 12, getHeight() = 6, getThickness() = 1, and getHistory yields an empty list.**

(b) **Create a parcel with dimensions (1, 2, 3). Verify that an IllegalArgumentExcpetion is thrown because both length and height are too small.**

(c) **Create a parcel with dimensions (1, 200, 3). Verify that an IllegalArgumentExcpetion is thrown because the girth + length is too large.**

(d) **(Boundary condition) Create a parcel with legal dimensions (0.25, 6, 3). Verify that the constructor creates the object successfully, getLength() == 6, getHeight() = 3, getThickness() = 0.25, and getHistory yields an empty list.**

Question 7. (10 points) Specifications. We want to include a method to append a new string to the tracking history in a `USPSParcel`. The method implementation is trivial and is given below, but it still needs to be specified properly.

Complete the JavaDoc comments for the `addEvent` method below to provide the most suitable specification. Leave any unneeded parts blank. There is space at the beginning (before @param) to write the summary description of the method that should begin every JavaDoc specification. You may have to use your best judgment based on the implementation to decide how to specify some details. Hint: the answer probably won't need nearly this much space.

```
/**
 *
 * Add a new event to the end of the tracking history for this USPSParcel
 *
 *
 *
 * @param event String with description of new event
 *
 *
 *
 * @requires event != null
 *
 *
 *
 * @modifies this
 *
 *
 *
 * @effects appends event to the end of the tracking history for this USPSParcel
 *
 *
 *
 * @throws
 *
 *
 *
 * @returns
 *
 *
 */
public void addEvent(String event) {
  history.add(event);
}
```

Question 8. (12 points) Equals/Hashcode. There are several possible ways to decide if two parcels are equal. Here are three of them (with funny line spacing to save space):

```
// this.equals(other) if dimensions are same
public boolean equals1(Object o) {
  if (!(o instanceof USPSParcel)) return false;
  USPSParcel other = (USPSParcel) o;
  return length == other.length && height == other.height &&
    thickness == other.thickness;
}

// this.equals(other) if length+girth are same
public boolean equals2(Object o) {
  if (!(o instanceof USPSParcel)) return false;
  USPSParcel other = (USPSParcel) o;
  double thisg = 2 * (height + thickness);
  double otherg = 2 * (other.height + other.thickness);
  return (length + thisg) == (other.length + otherg);
}

// this.equals(other) if sum of dimensions are same
public boolean equals3(Object o) {
  if (!(o instanceof USPSParcel)) return false;
  USPSParcel other = (USPSParcel) o;
  return length + height + thickness ==
    other.length + other.height + other.thickness;
}
```

Here are some possible `hashCode` methods for this class.

```
public int hashCode1() {
  return (int)length;
}
public int hashCode2() {
  return (int)(length + height + thickness);
}
public int hashCode3() {
  return (int)(31*(length + 31*height) + thickness);
}
public int hashCode4() {
  return (int)Math.max(length,
                       Math.max(height, thickness));
}
public int hashCode5() {
  return 331;
}
```

(answer the question on the next page – you can remove this page for reference.)

Question 8. (cont.) (a) (9 points) complete the following table by writing "OK" in each entry where the `hashCode` implementation given in the left column is a correct implementation of `hashCode` for the `equals` method given in the top row. Leave the entry blank if it is not correct for that particular combination of `equals` and `hashCode`. (Do not be alarmed if most entries are blank or, for that matter, if most of them are OK. Just pick the right answers. ☺)

	equals1	equals2	equals3
hashCode1	**OK**		
hashCode2	**OK**		**OK**
hashCode3	**OK**		
hashCode4	**OK**		
hashCode5	**OK**	**OK**	**OK**

(b) (3 points) Ignoring issues about compatibility with the various `equals` methods, if we only needed to pick a good `hashCode` method, which one of the five given `hashCode` methods would likely be best and why? Be brief.

hashCode3() would be best. It uses all three size fields of the object, and computes a hashCode that will have different values even if the dimensions are permutations of each other.

Question 9. (4 points) One of the programmers has pointed out that we haven't yet written the code for the method to return the tracking history to clients. The suggestion is to include this implementation, and since `Strings` are immutable, nothing can go wrong. Is that right? Why or why not? (briefly)

```
public List<String> getHistory() {
  return history;
}
```

No. The client would be given a reference to the history list. Although the client could not change the actual String values, it would be possible to alter the list itself.