CSE 331 Spring 2018 Midterm (SOLUTION)

Name ____

There are 8 questions worth a total of 93 points. **Please budget your time so that you get as many points as possible.** We have done our best to make a test that folks can complete in 50 minutes, but everyone works at a different pace, and that is just fine!

The exam is closed book, closed electronics, closed classmates, open mind. Many of the questions have short answers, even if the prompt is a little long. Don't worry!

For all questions involving proofs, assertions, invariants, etc., please assume that all integer quantities are unbounded (e.g., overflow cannot happen) and that ***integer division and square root (sqrt) are truncating as in Java***, i.e., 5/3 evaluates to 1 and sqrt(17) evaluates to 4.

If you do not remember the syntax of some command or the format of a command's output, make the best attempt you can. We will not be grading syntactic details.

Relax and have fun! We're all here to learn.

Please wait to turn the page until everyone is told to begin.

1.	__________________	/ 12	5.	__________________	/ 12
2.	__________________	/ 12	6.	__________________	/ 10
3.	__________________	/ 10	7.	__________________	/ 7
4.	__________________	/ 20	8.	__________________	/ 10

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: Forward Reasoning (12 points)

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)


{ x < -1 }

y = x * x;


{ y > 1 && x < -1 }


z = x * y;


{ z < -1 && y > 1 && x < -1 }


w = z < x;


{ w = true && z < -1 && y > 1 && x < -1 }
```

[Question 1 continued]

(b) (7 points)

```
{ |x| < 5 }

if (x % 2 == 0) // if x is even...

     { x % 2 = 0 && |x| < 5 }

     y = x * x;

     { x % 2 = 0 && |x| < 5 && y < x * x }

else

     { x % 2 = 1 && |x| < 5 }

     y = x + x;

     {  x % 2 = 1 && |x| < 5 && y = 2 * x }



{     |x| < 5 &&
     x % 2 = 0 -->  0 <= y <= 16) &&
     x % 2 = 1 --> -6 <= y <= 6)
}
```

QUESTION 2: Backward Reasoning (12 points)

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)


{ (a * a - 5) * 10 - a < 0 && a * a - 5 >= 0 }
==> { 10 * a * a - 50 - a < 0 && a * a >= 5 }
==> { 10 * a * a - a < 50 && |a| >= 3 }
==> { False }

b = a * a - 5;


{ b * 10 - a < 0 && b >= 0 }


c = b * 10 - a;


{ c < 0 && b >= 0 }
```

[Question 2 continued]

(b) (7 points)

We accepted solutions that do not simplify the precondition, solutions that simplify the the precondition using real number arithmetic, and solutions that simplify the precondition using integer arithmetic.

We gave a 1 point bonus for solutions that simplified the precondition to just x=7. (Note that it is stated at the beginning that variables are integers)

```
{ (x < 0 && 3 * y = 10) || (x >= 0 && 3 * x / 2 = 10) }
==> { (x < 0 && false) || (x >= 0 && 3 * x / 2 = 10) }
==> { x >= 0 && 3 * x / 2 = 10 }
==> { x = 7 }

if (x < 0)

     { 3 * y = 10 }

     x = 2 * y;

     {  x + y = 10 }

else

     { 3 * x / 2 = 10 }

     y = x / 2;

     { x + y = 10 }


{ x + y = 10 }
```

QUESTION 3: Loop Invariants and Proofs (10 points)

In this question, we want to verify that the difference between any two elements of an array is less than or equal to the result returned by the `range` method below. Fill in invariants to complete the proof.

```
public static int range(int[] a) {

    { a.length > 0 }

    int min = a[0];  int max = a[0];  int i = 1;


    { forall p, 0 <= p < i --> min <= a[p] <= max }

    while(i < a.length) {

        { forall p, 0 <= p < i --> min <= a[p] <= max }

        if(a[i] < min) min = a[i];

        { (forall p, 0 <= p < i + 1 --> min <= a[p]) &&
          (forall p, 0 <= p <   i   --> a[p] <= max)
        }

        if(a[i] > max) max = a[i];

        { forall p, 0 <= p < i + 1 --> min <= a[p] <= max }

        i++;

        { forall p, 0 <= p < i --> min <= a[p] <= max  }
    }

    { forall p, 0 <= p < i --> min <= a[p] <= max && i = a.length }
    ==> { forall p, 0 <= p < a.length --> min <= a[p] <= max }

    // invariant immediately above should imply
    // { forall p, q. a[p] - a[q] <= max - min }

    return max - min;
}
```

(If you need more space for an invariant, you can use the final blank page of the exam.)

An interval is a compact representation of a contiguous set of numbers. The next few questions concern a class called `Interval` that implements integer intervals. The representation of an interval is given by two fields `lo` and `hi`.

```
class Interval {
      private int lo;
      private int hi;

      public Interval(int lo, int hi) {
            this.lo = lo;
            this.hi = hi;
      }

      // accessors for lo and hi endpoints of this interval
      public int getLo() { return lo; }
      public int getHi() { return hi; }

      // return whether x is in this interval
      public boolean contains(int x) {
            return lo <= x && x <= hi;
      }

      // return union of this with another interval
      public Interval union(Interval other) {
            int uLo = Math.min(this.getLo(), other.getLo());
            int uHi = Math.max(this.getHi(), other.getHi());
            return new Interval(uLo, uHi);
      }

      // return integer in this interval closest to x
      public int clamp(int x) {
            if(x < lo) return lo;
            if(x > hi) return hi;
            return x;
      }
}
```

Please remove this page from your exam. This will help you to refer to the code in following questions and help us when scanning the exam to get consistent page number across students.

Please remove this page from your exam. This will help you to refer to the code in following questions and help us when scanning the exam to get consistent page number across students.

QUESTION 4: Specification and Design (20 points)

(a) `Interval` objects are immutable: (circle one) **TRUE** False

(b) Provide an example call to the `Interval` constructor that produces an empty interval (i.e., an `Interval` value `v` such that `v.contains(x)` always returns `false`):

new Interval(1, 0)3

(c) Give a suitable Representation Invariant (RI) for `Interval` (hint: it may be very simple):

Many valid choices. The simplest can just be "True." (Note that the correct answers to some of the other questions depends on the answer to this question.)

(d) The RI for `Interval` needs to be checked in every method: (circle one) **TRUE** **FALSE**

(The intent was for this to be FALSE, but since the fields were not marked "final" we accepted either answer.)

(e) Give a suitable Abstraction Function (AF) for `Interval`:

AF(this) = { x | this.lo <= x <= this.hi }

The key here is that the AF relates appropriately to the RI.

[Question 4 continued]

(f) Complete the JavaDoc comments below to provide the most suitable specification for `union`. Leave any unneeded parts blank. There may be multiple ways to get full points.

```
/** Return union this with another interval.
  *
  * @param Interval other to union with
  *
  * @requires other != null
  *
  * @modifies
  *
  * @effects
  *
  * @throws
  *
  * @returns a new Interval that contains this and other.
  *
  */
public Interval union(Interval other) { /* (see earlier code) */ }
```

(g) Complete the JavaDoc comments below to provide the most suitable specification for `clamp`. Leave any unneeded parts blank. There may be multiple ways to get full points.

```
/** Return integer in this interval closest to x.
  *
  * @param int x to find the closest interval element to
  *
  * @requires AF(this) != empty
  *
  * @modifies
  *
  * @effects
  *
  * @throws
  *
  * @returns closest integer to x in the interval
  *
  */
public int clamp(int x) { /* (see earlier code) */ }
```

[Question 4 continued]

(h) Complete the JavaDoc comments and the implementation below to provide a `size` method for `Interval`. Leave any unneeded parts blank. Your answer should respect the RI and AF. Hint: it may be helpful to consider the `contains` method.

```
/** Return number of integers in this interval.
  *
  * @param
  *
  * @requires
  *
  * @modifies
  *
  * @effects
  *
  * @throws
  *
  * @returns the number of elements in this interval
  *
  */
public int size() {

     if(lo > hi)
          return 0;


     return hi - lo + 1;










}
```

QUESTION 5: Testing (12 points)

For each part below, describe **two** separate, distinct "black box" tests for the `Interval` method in question. For each test give the input values and expected result(s). You do not need to write JUnit tests or other Java code. Reminder: there is a `contains()` observer method defined for this class that might be useful and you are also encouraged to use the `size()` method you defined earlier.

(a) Tests for the `union` method:

(b) Tests for the `clamp` method:

(c) Tests for the `size` method:

QUESTION 6: Equals and Hashcode (10 points)

(a) Implement `equals` for `Interval`. Two intervals should be considered equal if they contain the same set of integers.

```
@Override
public boolean equals(Object o) {
      if (!o instanceof Interval)
            return false;
      Interval i = (Interval)o;
      return (this.size() == 0 && i.size() == 0) ||
             (this.getLo() == i.getLo() && this.getHi() == i.getHi());
}
```

(b) Implement `hashcode` for `Interval`. To receive full points, your implementation should be of high quality (i.e., avoid unnecessarily having unequal objects hash to the same value).

```
@Override
public int hashcode() {
      if (this.size() == 0) {
            return 47;
      }
      return this.getLo() * 23 + getHi();
}
```

(c) What property would a "perfect hashcode" for Interval guarantee?

```
Only objects which are .equals() to one another hash to the same
value.
Same hashcode => equal (OK)
Not equal => different hashcode (OK)
Equal => same hashcode (NO)
Different hashcode => not equal (NO)
```

Is it possible to implement such a perfect hashcode method? (circle one) Yes [✓ NO]

QUESTION 7: Equals Equivalence Relation (7 points)

Classes overriding `equals` must implement an *equivalence relation*:
`a.equals(a) == true` (*reflexive*);
`a.equals(b) == b.equals(a)` (*symmetric*); and
`a.equals(b) && b.equals(c) == true` implies `a.equals(c) == true` (*transitive*).

Put a **check** next to the valid overriding implementations of `equals` for `ConstantInt` below.

```
public class ConstantInt {
  private int val;
  public ConstantInt(int v) {
    this.val = v;
  } }
```

	VALID?

`public boolean equals(Object x) {`
`  return false; }`
	NO

`public boolean equals(Object x) {`
`  return true; }`
	NO

`public boolean equals(ConstantInt x) {`
`  return this == x; }`
	NO

`public boolean equals(ConstantInt x) {`
`  return this.val == x.val; }`
	NO

`public boolean equals(Object x) {`
`  return this.val.equals(x); }`
	NO

`public boolean equals(Object x) {`
`  if(!(x instanceof ConstantInt))`
`    return false;`
`  ConstantInt ci = (ConstantInt)x;`
`  return this.val == ci.val; }`
	YES

`public boolean equals(Object x) {`
`  if(this == x)`
`    return true;`
`  if(!(x instanceof ConstantInt))`
`    return false;`
`  ConstantInt ci = (ConstantInt)x;`
`  return this.val == ci.val; }`
	YES

QUESTION 8: Comparing Specifications (10 points)

Here are four possible specifications for a method that checks whether one integer is a multiple of another.

(S1)

```
@param n
@param f
@returns true if there exists g such that n = f * g
This means that the return value is unspecified if such a g does not
exist.
```

(S2)

```
@param n
@param f
@requires f ≠ 0
@returns true if there exists g such that n = f * g, otherwise false
```

(S3)

```
@param n
@param f
@requires f > 0
@returns true if there exists g such that n = f * g, otherwise false
```

(S4)

```
@param n
@param f
@returns true if there exists g such that n = f * g and g > 0,
         otherwise false
```

(a) Circle the specifications as strong as S1: [✓ S1] S2 S3 S4

(b) Circle the specifications as strong as S2: S1 [✓ S2] S3 S4

(c) Circle the specifications as strong as S3: S1 [✓ S2] [✓ S3] S4

(d) Circle the specifications as strong as S4: S1 S2 S3 [✓ S4]

(e) Is it possible for a single method to satisfy both S1 and S4? (circle one) YES [✓ NO]

(f) Is it possible for a single method to satisfy both S2 and S3? (circle one) [✓ YES] NO