CSE 131 Introduction to Computer Science **Fall 2015**

Exam II

Given: 5 November 2015 Due: End of Exam Session

This exam is closed-book, closed-notes, no electronic devices allowed. The exception is the "sage page" on which you may have notes to consult during the exam. Answer questions on the pages of the exam. Do not unstaple the pages of this exam, nor should you attach any other pages to the exam. You are welcome to use the blank space of the exam for any scratch work.

Your work must be legible. Work that is difficult to read will receive no credit. Do not dwell over punctuation or exact syntax in code; however, be sure to indent your code to show its structure.

You must sign the pledge below for your exam to count. Any cheating will cause the students involved to receive an F for this course. Other action may be taken. If you need to leave the room for any reason prior to turning in your exam, you must give your exam and any electronic devices with a proctor.

You must fill in your identifying information correctly. Failure to do so is grounds for a zero on this exam. When you reach this point in the instructions, please give the instructor or one of the proctors a meaningful glance.

Print clearly the following information:		
Name (print clearly):		
Student 6-digit ID (print *really* clearly):		
Your answers below tell us where to return your graded exam.		
What time do you actually attend studio/lab?		
What room (222, 218, 216, or 214)? your best guess		
Problem Number	Possible Points	Received Points
1	20	
2	10	
3	15	
4	15	
5	20	
6	20	
Total	100	

Pledge: On my honor, I have neither given nor received any unauthorized aid on this exam.

Signed: _____

(Be sure you filled in your information in the box above!)

1. (20 points) The bulleted items below have blanks where some information is missing. For each blank, fill in the letter[1] that fits best conceptually where the blank occurs. You may have to use a given letter more than once; some letters may not belong in any of the blanks.

   - When describing the nature of an object via a story, each _____ of the story usually corresponds to an instance variable for the resulting class.
   - We avoid writing similar code multiple times by capturing the basic idea of the code in a _____. If that code requires some variation in its execution, then _____ are provided so that different values can be supplied at different times for its execution.
   - The method that is responsible for giving birth to a class is called a _____. That method is also responsible for initializing the class's _____.
   - If a class `C`'s instance variables are declared `final`, this has the effect of making all instances of `C` _____.
   - We can sometimes formulate a method by having it call a "smaller" instance of the problem it sets out to solve. This technique is know as _____. Without a _____, such methods would in theory never terminate.
   - If a class `C` has instance variables that are private, then `C` must include _____ for those instance variables' if their values are needed outside of `C`.
   - A _____ for a class is that class's only method that has no declared return type (*i.e.*, `int`, `void`, *etc.*).

   a Method
   b Instance variables
   c Parameters
   d Recursion
   e Base case
   f Immutable
   g Setters
   h `boolean`
   i `toString()`
   j Constructor
   k Has-a
   l Getters

[1] Do not write the term or phrase in the blank space! Use a letter from the provided list.

2. (10 points) Consider the following unusual definition of `isOdd(int n)`:

$$isOdd(n) = \begin{cases} isOdd(n-2), \text{if } n \geq 2 \\ \\ \end{cases}$$

(a) (5 points) The above definition is missing something. What is it missing?

(b) (5 points) The code corresponding to the definition above is given below. Without changing the code given, and without making any other recursive calls, complete the code so that it works properly for any $n \geq 0$.[2]

```
public static boolean isOdd(int n) {
   if (n >= 2) {
      return isOdd(n-2);
   }

}
```

[2]For $n < 0$, you can do whatever you like. We won't grade on that behavior.

3. (15 points) Consider the following recursive formula:

$$f(n) = \begin{cases} 3 \times f(n-1) + 4, \text{if } n > 0 \\ 1, \text{otherwise} \end{cases}$$

(a) (5 points) Below, write the code for f:

```
public static int f(int n) {




}
```

(b) (10 points) Below, using substitution, show the evaluation of $f(2)$, making sure you transcribe lines faithfully down the page, so that each line you write is truly equal to $f(2)$:[3]

```
f(2) =

     =

     =

     =

     =

     =
```

[3]You may or may not need all of the lines provided; also, use more if you need them.

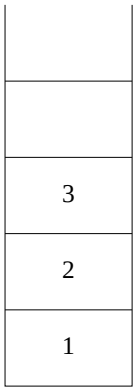
4. (15 points) Consider the method:

```
public static int foo(int x, int y, int z) {
   return x + y + z;
}
```

and a call to that method:

```
a = foo(b, c, d);
```

(a) (10 points) In the call to `foo`, just prior to `foo`'s execution, if the stack is as follows:



then what must have been the values of

- `b`? _____
- `c`? _____
- `d`? _____

(b) (5 points) Below, show the stack just prior to `foo` returning the answer that will be assigned to `a`:

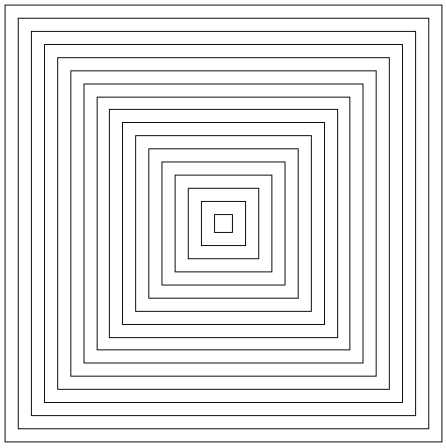
5. (20 points) Recall the API for drawing as square is as follows:

```
StdDraw.square(x,y,r)
```

draws a square *centered* at coordinate (`x`,`y`) whose half width and half height are both `r`. For example, given the default coordinates of a Sedgewick canvas, the call

```
StdDraw.square(0.5, 0,5, 0.5)
```

draws a single square centered on the screen at $(0.5, 0.5)$. Its lower left corner is at $(0, 0)$ and its upper right corner is at $(1, 1)$. It is in fact the outermost square in the following image:



In the above image, each contained square's half-width is 0.03 Sedgewick units less than its containing square.

In this problem you develop a recursive solution to drawing the image shown above.

(a) (5 points) Describe the substructure of the image you see above. More specifically, if the entire image contains squares whose half-widths are no more than 0.5, then how would you describe the next smaller instance of that image?

(b) (5 points) Describe the base case of the recursion present in the above image.

Continued on next page...

(c) (10 points) Complete the `squares` method below so that it draws (something resembling) the image above. **Be sure to circle the base case in your code.**

```
/**
 *
 * @param x x-coordinate of the center of this square
 * @param y y-coordinate of the center of this square
 * @param hw the half-width (and thus half-height) of this square
 */
public static void squares(double x, double y, double hw) {
     //
     // Complete this method    (5 points)
     //   I left you more room below than you probably need
     //
     //   Circle the base case! (5 points)
     //









}

public static void main(String[] args) {
     //
     // Start recursion using the unit square, centered at (0.5,0.5)
     //   and with a half-width of 0.5
     // Do not change this method!
     //

     squares(0.5, 0.5, 0.5);
}
```

6. (20 points) Consider the following user story:

   A (3-dimensional) `Box` has-a

   - width
   - length
   - height
   - indication of whether the `Box` is currently open or closed

   In terms of behavior, our `Box` object should offer the following functionality:

   - There should be getters, but no setters, for the `Box`'s width, length, and height.
   - There should be a getter and setter for the `Box`'s "openness" (whether the box is currently open or closed).
   - It should be possible to find the volume of the `Box`, computed as its width×length×height.
   - It should be possible to find the area of the bottom of the `Box`, computed as its width×length.
   - It should be possible to determine whether another `Box b` fits inside `this Box`, determined as follows:
     - The area of `b`'s bottom must be less than `this Box`'s bottom, **and**
     - The volume of `b` must be less than `this Box`'s volume.

   Note that the above requirements do not print out anything. Your methods therefore should not print anything. Instead they should affect the object by changing its instance variable(s) or by returning appropriately typed values to provide the desired functionality.

   (a) (3 points) Although you will not generate `hashCode` or `equals`, on which of the above "has-a"s would you base those methods, and why?

   (b) (17 points) Below, and on the facing page, write the code for your `Box` class, placing elements of your class between the comments as indicated.

```
public class Box {

   //
   // (4 points) Instance variable(s) below here.   For each instance
   //     variable, chose a type that best fits its purpose and usage.
```

```
//
// (3 points) Constructor below here.
//
```

```
//
// (2 points) toString() below here.
//    (something very simple: no need to include all instance variables)
```

```
//
//  (8 points) Other methods below here. DO NOT provide equals or hashCode.
//
```

```
}
```