CS106A Practice Midterm

This exam is closed-book and closed-computer but open-note. You may have a double-sided, 8.5” × 11” sheet of notes with you when you take this exam. Please hand-write all of your solutions on this physical copy of the exam.

In all questions, you may include methods, classes, or other definitions that have been developed in the course by giving the name of the method or class and the handout, chapter number, lecture, or file in which that definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do.

Unless stated otherwise, you do not need to worry about efficiency.

On the actual exam, there'd be space here for you to write your name and sign a statement saying you abide by the Honor Code. We're not collecting or grading this exam (though you're welcome to step outside and chat with us about it when you're done!) and this exam doesn't provide any extra credit, so we've opted to skip that boilerplate.

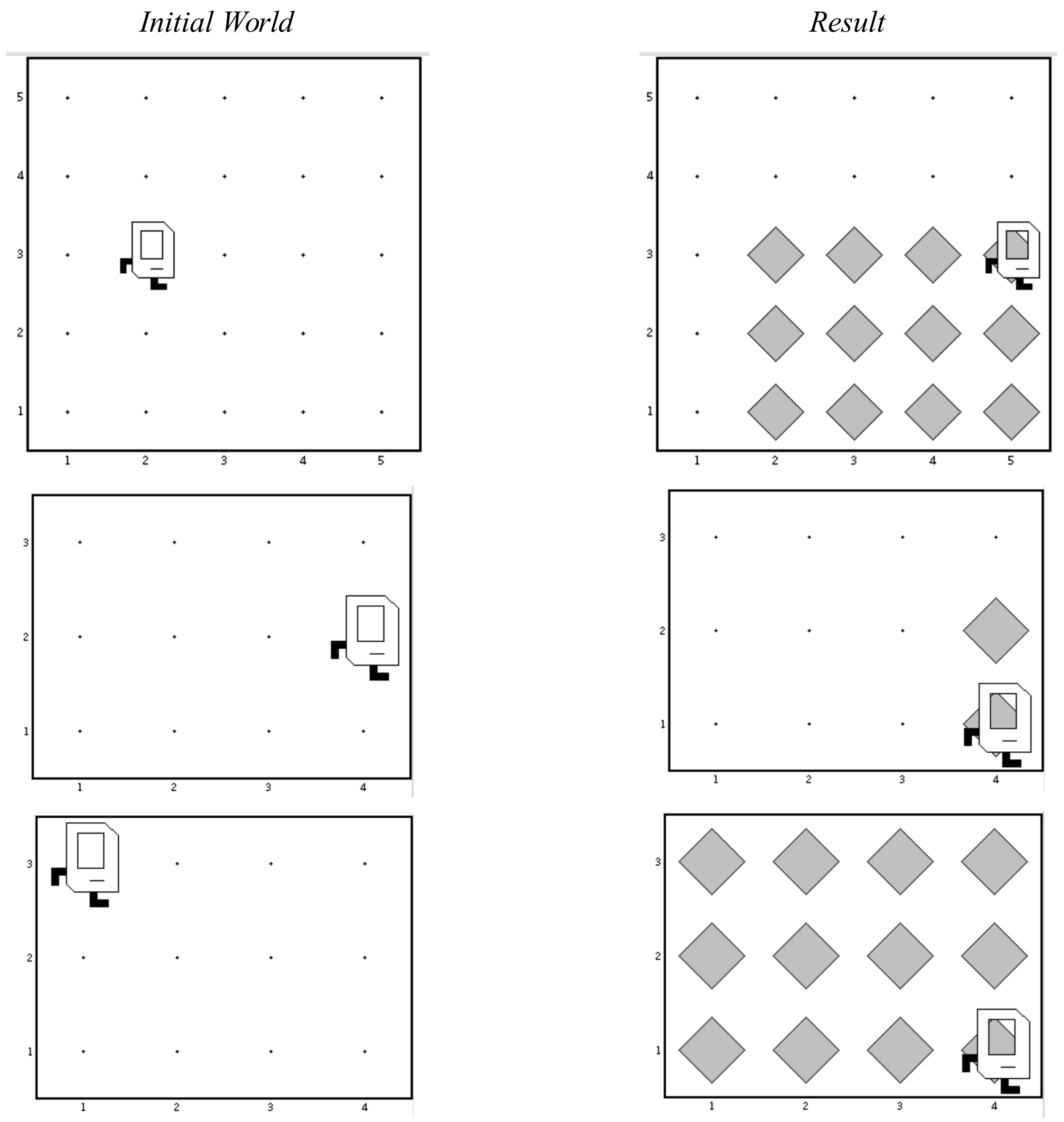
You have three hours to complete this exam. There are 50 total points, and this exam is worth 20% of your grade in this course.

Question	**Points**
(1) `RectangleKarel`	/ 10
(2) Jumbled Java hiJinks	/ 10
(3) Slicing a Cake	/ 10
(4) Pebbling a Checkerboard	/ 10
(5) Damaged DNA Diagnoses	/ 10
	/ 50

Good Luck!

Problem 1: `RectangleKarel` (10 Points)

In this problem, you will write a program to get Karel to fill a rectangular region of Karel's world with beepers. The rectangle's upper-left corner is given by Karel's initial position, and the rectangle's lower-right corner should be the lower-right corner of the world. For example:



In implementing your solution, you can assume the following:

- Karel always begins facing East, but otherwise Karel's initial position is unspecified.
- Karel's world has arbitrary size, no beepers, and no internal walls.
- Karel's beeper bag has infinitely many beepers in it.
- Karel's final position and orientation are irrelevant.

You are limited to the instructions in the Karel booklet. For example, the only variables allowed are loop control variables used within the control section of a `for` loop (where the loop upper bound is a fixed number), you must not use the `break` or `return` statements, etc. You may, however, use the `&&`, `||`, and `!` operators in the conditions of `if` statements and `while` loops. Write your solution on the next page, and feel free to tear out this page as a reference.

```
import stanford.karel.*;

public class RectangleKarel extends SuperKarel {
    public void run() {
```

(Extra space for Problem One, if you need it)

Problem Two: Jumbled Java hiJinks (10 Points Total)

(i) Solve for *x* (6 Points)

For each of these programs, determine whether there are any values of x that can be entered so that the program will print **success** without causing any errors and without printing anything else. If there are multiples values of x that will work, just give one of them. If there are no values of x that will work, write "no solution." No justification is necessary.

```
/* Program A */
int x = readInt();
if (x != 0 && x / 2 == 0) {
    println("success");
}
```

Answer for Program A: ______________________

```
/* Program B */
int x = readInt();
if (x >= 10000) {
   while (x != 0) {
      if (x % 10 != 9) {
         println("failure");
      }
      x /= 10;
   }
   println("success");
}
```

Answer for Program B: ______________________

```
/* Program C */
int x = readInt();
if (x > 7 && x / 0 == x) {
    println("success");
}
```

Answer for Program C: ______________________

```
/* Program D */
int x = readInt();
if (x != 0 || x != 1) {
    println("failure");
}
println("success");
```

Answer for Program D: ______________________

```
/* Program E */
int x = readInt();
if (1 - 3 - 5 - x == -10) {
    println("success");
}
```

Answer for Program E: ______________________

```
/* Program F */
int x = readInt();
if (x / 2 * 3 == 6) {
    println("success");
}
```

Answer for Program F: ______________________

(ii) Program Tracing **(4 Points)**

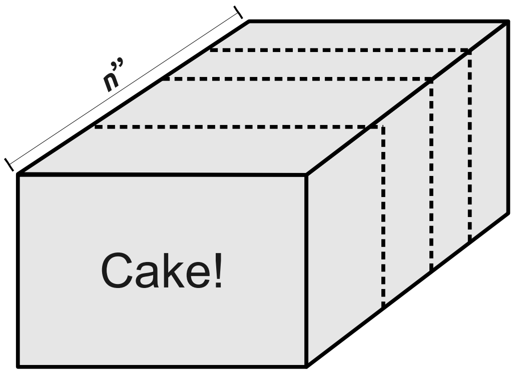
Determine the output of the following program and write it in the indicated box.

```
import acm.program.*;

public class FrozenJava extends ConsoleProgram {
      public void run() {
            int anna = 16;
            int elsa = 18;

            anna = arendelle(anna, elsa);
            println("anna = " + anna);
            println("elsa = " + elsa);
      }

      private int arendelle(int elsa, int anna) {
            String kristoff = "hans";

            weselton(kristoff);
            println("kristoff = " + kristoff);

            elsa = kristoff.length();
            return anna;
      }

      private void weselton(String olaf) {
            olaf += "el";
            println("olaf = " + olaf);
      }
}
```

Write the output of the program in this box:

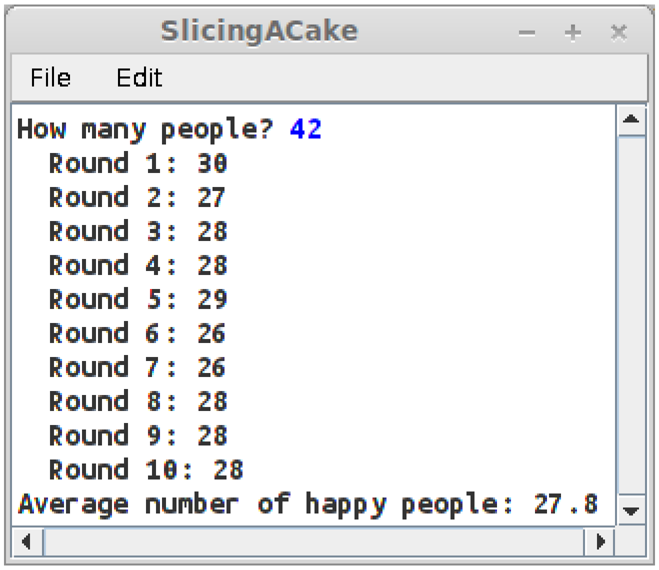
Problem Three: Slicing a Cake (10 Points)

Suppose that *n* people want to share a rectangular cake that's *n* inches long (that is, the cake's length is equal to the number of people, in inches). Each person takes turns going up to the cake and cutting a slice for herself. In a perfectly fair world, each person would cut a one-inch piece of cake so that everyone would end up with the same amount of cake. We'll say that a person is *happy* if she gets at least a one-inch slice of cake. Unfortunately, people are greedy and won't always cut a piece of the right size. Suppose that the cake gets sliced in the following way. Each person, one after the other, goes up to the cake to slice a piece. If there is at least 2" of cake left, the current person randomly chooses an amount of cake between 1" and 2" and cuts a piece of that size, ending up happy. If there is between 1" and 2" of cake left, then the current person takes whatever's left and is happy. Otherwise, there's less than 1" left, and the current person takes what's left but isn't happy.

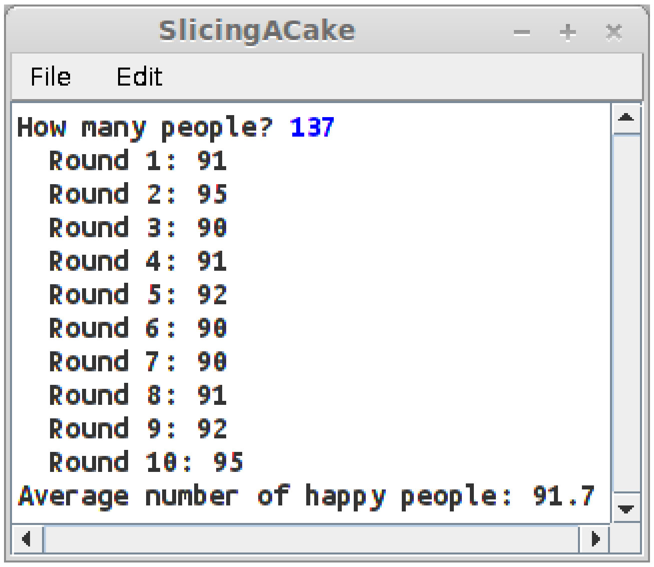


Your job in this problem is to write a program that estimates the average number of people who will end up happy this way. On startup, your program should prompt the user for the number of people who will split the cake. You should then run `NUM_TRIALS` simulations of slicing the cake according to the above rules. (In the sample runs below, `NUM_TRIALS` is 10). After each simulation, your program should print out how many total people end up happy (that is, with at least a 1" slice of cake). Once all the simulations finish, you should print out the average number of happy people across all the rounds.

Two sample runs of this program are shown below:



```
How many people? 42
  Round 1: 30
  Round 2: 27
  Round 3: 28
  Round 4: 28
  Round 5: 29
  Round 6: 26
  Round 7: 26
  Round 8: 28
  Round 9: 28
  Round 10: 28
Average number of happy people: 27.8
```



```
How many people? 137
  Round 1: 91
  Round 2: 95
  Round 3: 90
  Round 4: 91
  Round 5: 92
  Round 6: 90
  Round 7: 90
  Round 8: 91
  Round 9: 92
  Round 10: 95
Average number of happy people: 91.7
```

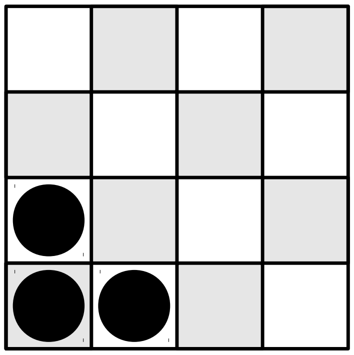
Write your answer on the next page, and feel free to tear out this page as a reference.

```
import acm.program.*;
import acm.util.*;

public class SlicingACake extends ConsoleProgram {
      private static final int NUM_TRIALS = 10;

      public void run() {
```

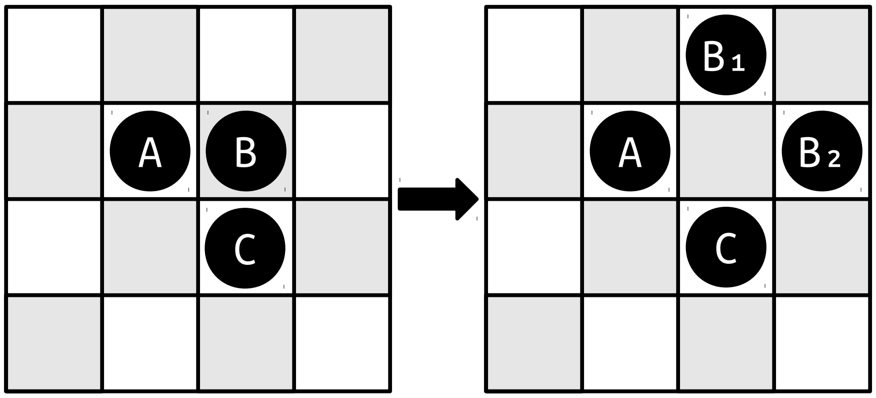
(extra space for Problem Three, if you need it)

Problem Four: Pebbling a Checkerboard (10 Points)

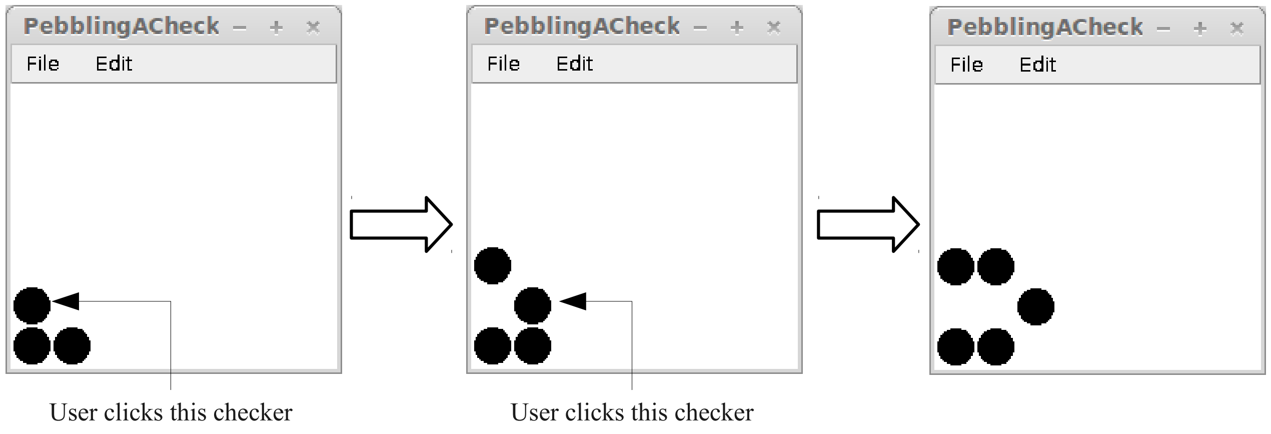
Suppose that you have a checkerboard of some size. You begin with three checkers in the lower-left corner in an L-shape, as shown here:



In this game, each move consists of removing a checker from a square and replacing that checker with two checkers – one directly above the original checker and one directly to the right of the original checker. It is only legal to move a checker if both of these adjacent squares are unoccupied. For example, in the diagram to the left, it's legal to move checker B by replacing it with the two checkers B₁ above it and B₂ to its right, yielding the given result. However, the other checkers in the initial setup can't move – checker A has a checker to its right (namely, B) and checker C has a checker above it (namely, B). After moving checker B, the player can either move checker A or move checker C.



Your job is to write a program that lets the user play this game. Specifically, when your program begins, you should draw three checkers in an L shape in the bottom-left corner of the screen. Whenever the user clicks on a checker, if the squares directly above the checker and directly to the right of the checker are free, then you should remove the checker that was clicked on and add two checkers, one directly above it and one directly to the right. A sample run of the program is shown below:



(Continued on next page)

When implementing this program, you should adhere to the following:

- The checkers should be filled in solid black and have width and height given by the constant `CHECKER_SIZE`.
- You do not need to (and, for the sake of time, shouldn't) actually draw the squares on the checkerboard. You just need to show where the checkers are.
- You do not need to leave any "margins" between the checkers. The checkers can be flush up against one another.
- The player never "wins" or "loses" this game; all you need to do is implement the logic to let the user make moves.
- If the user clicks on a checker that cannot move, your program shouldn't move it. However, you do not need to report an error or otherwise inform the user of why the checker can't move.
- If the user clicks on a checker that's at the very top or far right of the window, it's okay to let the user move that checker, even though the resulting checkers will be off the screen.

As a reminder, you can use the `getElementAt` method to find which checker, if any, is at a particular location on the screen. You may want to make multiple calls to `getElementAt` in order to determine whether the user has clicked on a checker and whether there are any checkers above or to the right of the selected checker.

```
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
import java.awt.event.*;

public class PebblingACheckerboard extends GraphicsProgram {
    /* The size of a checker. */
    private static final double CHECKER_SIZE = 50;
```

(extra space for Problem Four, if you need it)

Problem Five: Damaged DNA Diagnoses (10 Points)

DNA molecules consist of two paired strands of *nucleotides*, molecules which encode genetic information. Computational biologists typically represent each DNA strand as a string made from four different letters – A, C, T, and G – each of which represents one of the four possible nucleotides.

The two paired strands in a DNA molecule are not arbitrary. In normal DNA, the two strands always have the same length, and each nucleotide (letter) from one strand is paired with a corresponding nucleotide (latter) from the second strand. In normal DNA strands, the letters are paired as follows:

A is paired with **T** and vice-versa.

C is paired with **G** and vice-versa.

Below are two matching DNA strands. Note how the letters are paired up according to the above rules:

```
GCATGGATTAATATGAGACGACTAATAGGATAGTTACAACCCTTACGTCACCGCCTTGA
↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕
CGTACCTAATTATACTCTGCTGATTATCCTATCAATGTTGGGAATGCAGTGGCGGAACT
```

In some cases, errors occur within DNA molecules. This problem considers two types of DNA errors:

- **Point mutations**, in which a letter from one strand is matched against the wrong letter in the other strand. For example, **A** might accidentally pair with **C**, or **G** might pair with **G**.
- **Unmatched nucleotides**, in which a letter from one strand is not matched with a letter from the other. We represent this by pairing a letter from one strand with the – character in the other.

For example, consider these two DNA strands:

GGGA-GAAT**C**TCTGG**A**C**T**

CCCT**A**CTTA-AGACC**G**G**T**

Here, there are two unmatched nucleotides (represented by one DNA strand containing a dash character instead of a letter) and two point mutations (**A** paired with **G** and **T** paired with **T**).

Although both of these types of errors are problematic, we will consider point mutations to be a less "costly" error than an unmatched nucleotide. Let's assign a point mutation (a letter matched with the wrong letter) a cost of 1 and an unmatched nucleotide (a letter matched with a dash) a cost of 2.

Your job is to write a method

```
private int costOfDNAErrorsIn(String one, String two)
```

that accepts as input two strings representing DNA strands, then returns the total cost of all of the errors in those two strands. You may assume the following:

- The two strings have the same length.
- Each string consists purely of the characters **A**, **C**, **T**, **G**, and – (the dash character). All letters will be upper-case.
- A – character in one string is never paired with a – character in the other.

Here are some example inputs to the method, along with the expected result. For simplicity, all errors have been highlighted in bold:

Strands	Result
ACGT TGCA	**0** *(No errors)*
A-C-G-T-ACGT T**T**G**G**C**C**A**A**TGCA	**8** *(Four unmatched nucleotides)*
AAA**A**AAA TTT**A**TTT	**1** *(One point mutation)*
GAT**T**A**C**A CTA**T**T-T	**3** *(One point mutation, one unmatched nucleotide)*
CAT-TAG**-AC**T GTA**T**ATC**CAA**A	**6** *(Two point mutations, two unmatched nucleotides)*
-------- **ACGTACGT**	**16** *(Eight unmatched nucleotides)*
TAATAA ATTATT	**0** *(No errors)*
GGGA-GAAT**A**TCTGG**A**C**T** CCCT**A**CTTA-AGACC**G**G**T**	**6** *(Two point mutations, two unmatched nucleotides)*

Feel free to tear this and the preceding page out as a reference, and write your implementation on the next page.

```
private int costOfDNAErrorsIn(String one, String two) {
```