

Extra-Credit Problem Set

Due date: Wednesday, February 24, 5:00 P.M. (No late days on problem sets.)

Name (*please print*) _____

Section Leader _____

The purpose of this handout is to give those of you who had troubles with the midterm exam an opportunity—along with an incentive—to get more practice with algorithmic problems. There are five problems on this problem set. Each answer that is substantially correct will add two points to your percentage score on the midterm, up to a maximum of 100% on the exam. Answers that are close enough to receive at least half credit on an exam will give you one extra point. If most of you turn in this problem set and do reasonably well, the median on the midterm will rise from 70% to something in the high 70s, which is what we're shooting for. The scaling of the exam will remain exactly the same as shown on Handout #43, so that turning in this extra-credit assignment can help your grade but not hurt it. Moreover, deciding not to turn in this problem set won't hurt your course grade at all; you'll simply keep your original grade on the midterm exam.

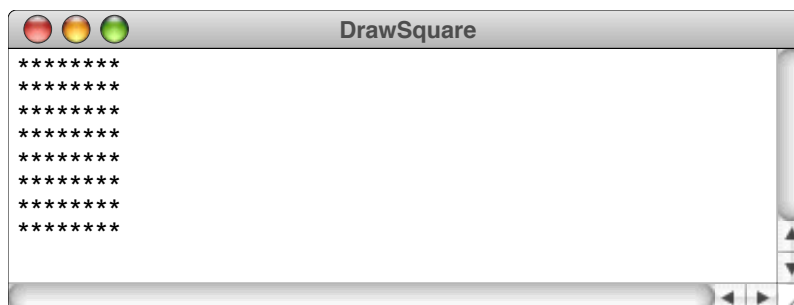
These problems are designed to be completed by hand, just as if they were exam problems, given that exams are, after all, where the problems seem to come up. Feel free, however, to run them on the computer if you want to check your algorithms. (If you do implement them, you can go ahead and staple in listings of the code, since I suspect your section leaders would rather look at a printed copy than try to decipher handwritten code.) We will grade these problems as if they were handwritten on an exam and not be sticklish about semicolons and other minor syntactic problems.

1. Console graphics

Back in the days before computer graphics, one of the typical assignments that we would give early in CS106A was to draw some figure on the console using characters. As a simple example, if you execute the nested loop

```
for (int i = 0; i < 8; i++) {  
    for (int j = 0; j < 8; j++) {  
        print("*");  
    }  
    println();  
}
```

draws an 8×8 square on the console composed of asterisks, as follows:



These exercises were not as flashy as the **acm.graphics** applications you write today, but they did have the advantage of forcing students to learn how to use nested **for** loops and how to think algorithmically about how to create the figure.

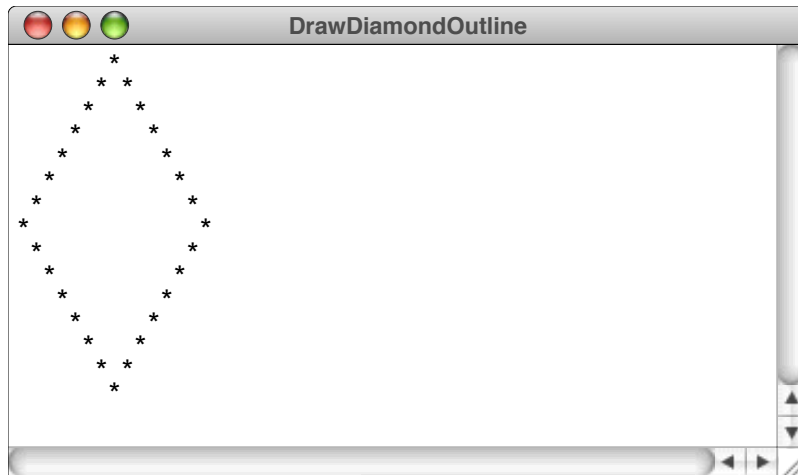
In this problem, your mission is to write a function

```
private void drawDiamondOutline(int size)
```

that draws the outline of a diamond whose sides are composed of the number of stars indicated by **size**. For example, if your **run** method is

```
public void run() {  
    drawDiamondOutline(8);  
}
```

your program should print asterisks and spaces in just the right places to produce the following sample run:



Answer to problem 1:

```
private void drawDiamondOutline(int size) {
```

```
}
```

2. The consecutive heads problem (Chapter 6, exercise 2, page 214)

Heads...

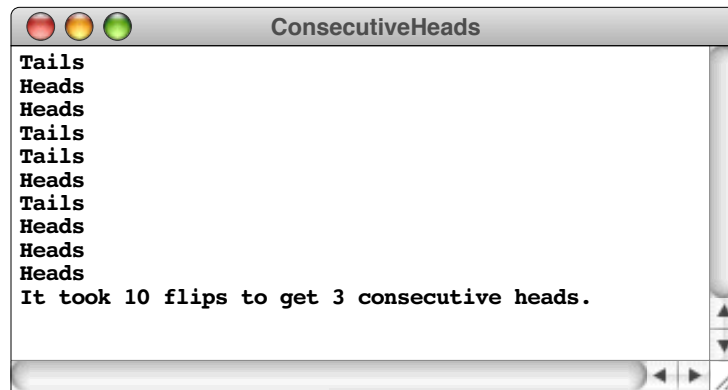
Heads...

Heads...

A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.

—Tom Stoppard, *Rosencrantz and Guildenstern are Dead*, 1967

Write a program that simulates flipping a coin repeatedly and continues until three *consecutive* heads are tossed. At that point, your program should display the total number of coin flips that were made. The following is one possible sample run of the program:



Answer to problem 2:

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

```
public class ConsecutiveHeads extends ConsoleProgram {
```

```
    private RandomGenerator rgen = RandomGenerator.getInstance();  
}
```

Problem 3: A series approximation for pi

In my lecture on random numbers, I showed how one might approximate π by simulating the process of throwing darts at a circular dart board. At the time, I noted that there are other, more reliable ways to compute the value of π . The German mathematician Leibniz (1646–1716), for example, discovered the rather remarkable fact that π can be computed using the following mathematical relationship:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

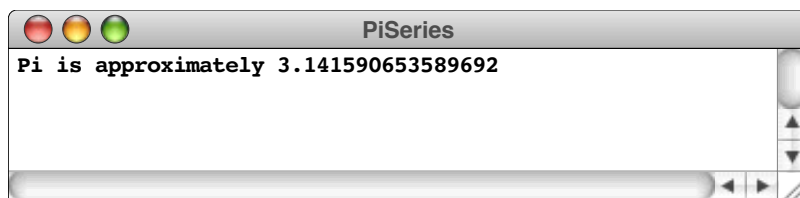
The formula to the right of the equal sign represents an *infinite series*; each fraction making up the series is called a *term*. If you start with 1, subtract one-third, add one-fifth, and so on, for each of the odd integers, you get a number that gets closer and closer to the value of $\pi/4$.

Write a program that uses this formula to calculate the approximation to π that results from carrying out this computation until the value of a term becomes less than a threshold defined as a named constant as follows:

```
private static final double TERM_THRESHOLD = .000001;
```

To make the steps in this calculation a little clearer, your program must

1. Include a loop that keeps track of the total on the right-hand side of the formula at each step in the process.
2. Keep track of the current term, which involves computing the next odd number and then taking its reciprocal (i.e., what you get if you divide 1 by that number).
3. Continue to cycle through the loop until the current term is smaller than the value of **TERM_THRESHOLD**.
4. Keep track of whether to add or subtract the term from the running total. The first term is added, the second subtracted, the third added, and so on, alternating back and forth.
5. Multiply the total by 4 at the end of the loop.
6. Display the result in a line like this:



Note: Even though this problem involves fractions, you should not be misled into using the **Rational** class introduced in Chapter 6. Doing so will not help at all but will instead just get you into trouble. All you need for this problem are **ints** and **doubles**.

Answer to problem 3:

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

```
public class PiSeries extends ConsoleProgram {
```

```
}
```

4. Inverting an encryption key

In last Wednesday's class that introduced cryptography, one of the code examples was a program that encrypted messages using a letter-substitution cipher. What we did not have time to write was the program to reverse that encryption, making it possible for the receiver to read the encrypted text. Although it is easy enough to add a separate decryption phase to the program we wrote in class, another approach is to write a function that, given a 26-letter encryption key, figures out what key you would need to *invert* (a fancy mathematical term for *undo*) the original encryption.

The idea of inverting a key is most easily illustrated by example. Suppose, as in Handout #34, that we are using the key "QWERTYUIOPASDFGHJKLZXCVBNM" (a sequence of letters unimaginately generated by typing the keys in order on the keyboard). That key represents the following translation table:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Q	W	E	R	T	Y	U	I	O	P	A	S	D	F	G	H	J	K	L	Z	X	C	V	B	N	M

Using this key, the letter 'A' gets translated into the letter 'Q', 'B' gets translated into 'W', and so on. If you want to turn this encryption process around, you have to read the translation table from bottom to top, looking to see where each letter in the ciphertext came from. Near the middle of the key, you discover that the letter 'A' in the ciphertext must have come from a 'K' in the plaintext. Similarly, the only way to get a 'B' in the ciphertext is to start with an 'X' in the original message. The first two entries in the inverted translation table therefore look like this:

A	B
↓	↓
K	X

If you continue this process by finding each letter of the alphabet on the bottom of the original translation table and then looking to see what letter appears on top, you will eventually complete the inverted table, as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
K	X	V	M	C	N	O	P	H	Q	R	S	Z	Y	I	J	A	D	L	E	G	W	B	U	F	T

The inverted key is simply the 26-letter string on the bottom row, which in this case is "KXVMCNOHPQRSZYIJADLEGWBUFT".

Write a method

```
private String invertKey(String key)
```

that uses this strategy to generate a decryption key for the encryption key passed as the argument. For example, if you were to call

```
invertKey("QWERTYUIOPASDFGHJKLZXCVBNM")
```

you should get back "KXVMCNOHPQRSZYIJADLEGWBUFT", as the earlier discussion shows.

Answer to problem 4:

```
private String invertKey(String key) {
```

```
}
```

5. The birthday paradox

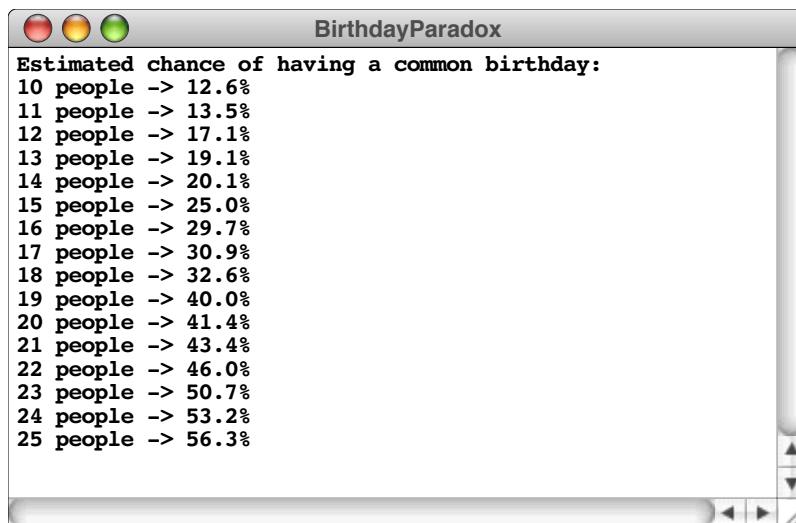
One of the reasons that our lives seem to be so full of coincidental happenings is that, given the large number of opportunities for such events to occur, “coincidences” happen more frequently than one might intuitively expect. For example, most people are surprised to discover that in a room with 23 people, the odds are slightly better than 50-50 that at least two of those people share the same birthday. This phenomenon is sometimes called the *birthday paradox*.

Although it is not hard to calculate the exact probability for duplicated birthdays in a group of N people, it is much easier to simulate this process. For example, you can generate N random birthdays—which you can assume is simply a random number between 1 and 365, ignoring leap years—and then look to see if any of those generated birthdays match. This outcome represents one trial in the simulation. If you repeat this trial **N_TRIALS** times and keep track of the number of times there was an overlap of birthdays, the ratio

$$\frac{\text{number of trials in which there was a common birthday}}{\mathbf{N_TRIALS}}$$

provides an estimate of the probability of an overlap. As **N_TRIALS** increases, this estimate will become more accurate.

Write a complete Java program that displays the result of this simulation, varying the number of people in a range specified by the constants **LOWER_LIMIT** and **UPPER_LIMIT**. For example, if these constants are 10 and 25, a sample run of the program might look like this:



```
Estimated chance of having a common birthday:
10 people -> 12.6%
11 people -> 13.5%
12 people -> 17.1%
13 people -> 19.1%
14 people -> 20.1%
15 people -> 25.0%
16 people -> 29.7%
17 people -> 30.9%
18 people -> 32.6%
19 people -> 40.0%
20 people -> 41.4%
21 people -> 43.4%
22 people -> 46.0%
23 people -> 50.7%
24 people -> 53.2%
25 people -> 56.3%
```


Answer to problem 5:

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

public class BirthdayParadox extends ConsoleProgram {

    /* Private constants */
    private static final int N_TRIALS = 1000;
    private static final int LOWER_LIMIT = 10;
    private static final int UPPER_LIMIT = 25;

    /* Instance variables */
    private RandomGenerator rgen = RandomGenerator.getInstance();
}
```