Practice Second CS106A Midterm Exam

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.

Question		**Points**	**Grader**
(1) Isograms	(10)	/ 10	
(2) Announcing Election Results	(10)	/ 10	
(3) Short Answer Questions	(10)	/ 10	
(4) Kerning	(10)	/ 10	
(5) Animal Hipsters	(10)	/ 10	
	(50)	**/ 50**	

Good Luck!

Problem One: Isograms (10 Points)

An *isogram* is a word that contains no repeated letters. For example, the word "computer" is an isogram because each letter in the word appears exactly once, but the word "banana" is not because 'a' and 'n' appear three times each. "Isogram" is itself an isogram, but "isograms" is not because there are two copies of 's'. There are many long isograms in English; for example, "uncopyrightable" and "computerizably."

Write a method

```
private boolean isIsogram(String word)
```

that accepts as input a string containing a single word, then returns whether that word is an isogram. The input word won't contain spaces or punctuation, but it might contain both upper-case and lower-case letters.

```
private boolean isIsogram(String word) {
```

Problem Two: Announcing Election Results (10 Points)

Suppose that you are in charge of writing software to tally up votes in an election. Your job is to write a method

```
private String electionWinner(String[] votes)
```

This method accepts as input a `String[]` representing all the votes that were cast in an election. You should then determine whether any candidate received strictly more than half the votes. If so, you should return the name of that candidate. If no candidate won (perhaps, for example, there are three candidates and each got a third of the votes), you should return **`null`** as a sentinel.

In writing this method, you can assume the following:

- Candidate names are case-insensitive, so "Karel the Robot" and "KAREL the ROBOT" should both count as votes for Karel the Robot. Because candidate names are case-insensitive, you can return the name of the winning candidate in any capitalization you'd like.
- There can be any number of candidates, not just two.
- There can be any number of votes in the input array, including zero.

```
private String electionWinner(String[] votes) {
```

Problem Three: Short Answer (10 Points)

(i) Comparing Data Structures, Part One (4 Points)

The `String` type is Java's standard way of storing text. You could also use a `char[]` to store text by simply representing the text as an array of all of its characters.

Give one advantage of representing text as a `String` over representing text as a `char[]` and vice-versa.

(ii) Comparing Data Structures, Part Two (3 Points)

We've seen `ArrayList<`*type*`>`s as one way of storing a growable sequence of elements of type *type*. Another alternative would be to use a `HashMap<Integer, `*type*`>`, where the keys represent the indices and the values represent the values stored at each position.

Give one advantage of representing a list as an `ArrayList<`*type*`>` over representing the list using a `HashMap<Integer, `*type*`>` and vice-versa.

(iii) An iOS Vulnerability **(3 Points)**

Last year, Apple announced a serious security error in its iOS operating system that made it possible for hackers to bypass security measures used when communicating over the internet. This was due to a small programming error that's understandable given just what you've seen in CS106A so far.

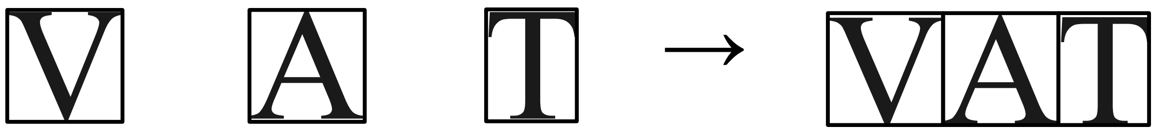
Below is some code that contains an error similar to the one that caused the security bug in iOS. For convenience, we've add line numbers to this code.

```
/**
 * Given a number, returns whether that number is divisible by
 * two, three, or five.
 *
 * @param number The number in question.
 * @return true if that number is divisible by two, three, or five
 *         and false otherwise.
 */
 1: private boolean hasSmallDivisor(int input) {
 2:    boolean result = false;
 3:
 4:    if (input % 2 == 0)
 5:        result = true;
 6:
 7:    if (input % 3 == 0)
 8:        result = true;
 9:        result = true;
10:
11:    if (input % 5 == 0)
12:        result = true;
13:
14:    return result;
15: }
```

This method is incorrect because it always returns **true**. Explain why. Be specific.

Problem Three: Kerning (10 Points)

Although we've used `GLabel`, we've never discussed how the computer actually displays text. Internally, the computer maintains a set of images representing what each character looks like. To display text on the screen, the computer lays out these images side-by-side. For example, to display the string "VAT," the computer begins with a set of images for the letters V, A, and T, then places them side-by-side to form the string. This is shown here:

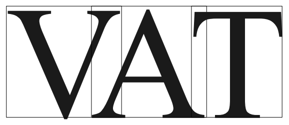


Unfortunately, this approach to laying out text will distort certain strings. For example, consider the following rendition of the string "THE VATICAN:"

THE VATICAN

Notice how the V, A, and T in "VATICAN" appear to be spaced out more than the T, I, and C. The reason for this is that the images for the letters V, A, and T have a lot of whitespace in them. When the images for the letters are placed next to one another, this whitespace adds up and spaces the letters farther apart than they should be.

To correct for this, the computer typically overlaps the images for certain pairs of letters to reduce whitespace. For example, if we slightly overlap the images for V and A and the images for A and T, we get this rendering of the word VAT:

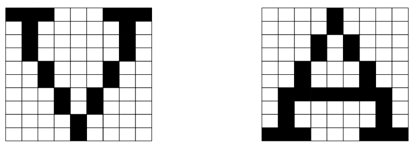


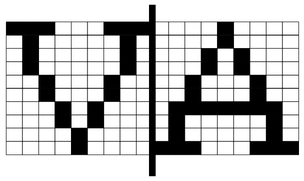
The amount that the images of two letters overlap is called the *kern*, and the process of overlapping letters this way is called *kerning*. Kerning can make text much more aesthetically pleasing. Compare the above rendition of "THE VATICAN," which had no kerning, to this one, which has been kerned:

THE VATICAN

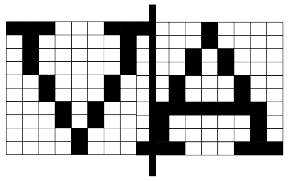
Notice how there is less blank space between the V, A, and T in VATICAN.

Your task in this problem is to write a method that will accept as input images of two letters, then will kern the images by some specified amount. For example, here is the sample output of this method on the letters V and A with several different kerns; the vertical bar in the outputs marks the end of the V image:

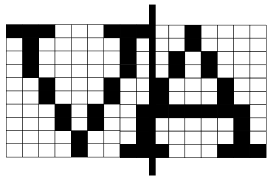




Kern 0

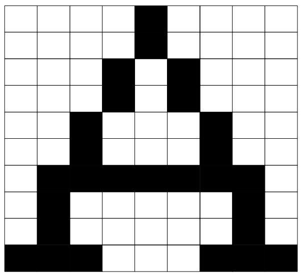


Kern 1



Kern 2

For simplicity, and to avoid some of the complexities of `GImage`, we will represent the images of letters as two-dimensional arrays of `boolean`s indicating for each pixel in the image whether the pixel is white (`false`) or black (`true`). As an example, the letter A might be represented as follows:



```
{
 { false, false, false, false, true,  false, false, false, false },
 { false, false, false, false, true,  false, false, false, false },
 { false, false, false, true,  false, true,  false, false, false },
 { false, false, false, true,  false, true,  false, false, false },
 { false, false, true,  false, false, false, true,  false, false },
 { false, false, true,  false, false, false, true,  false, false },
 { false, true,  true,  true,  true,  true,  true,  true,  false },
 { false, true,  false, false, false, false, false, true,  false },
 { false, true,  false, false, false, false, false, true,  false },
 { true,  true,  true,  false, false, false, true,  true,  true  }
}
```

Write a method

```
  private boolean[][] kernLetters(boolean[][] first, boolean[][] second, int kern)
```

that accepts as input two **`boolean`** arrays representing images of letters, along with an amount to overlap the two images, then returns a new **`boolean`** array representing the image formed by kerning the two letters by the given amount. You can assume that the two images have the same height, though they might not have the same width. You can also assume that the amount to kern the letters is nonnegative and is smaller than the widths of either image.

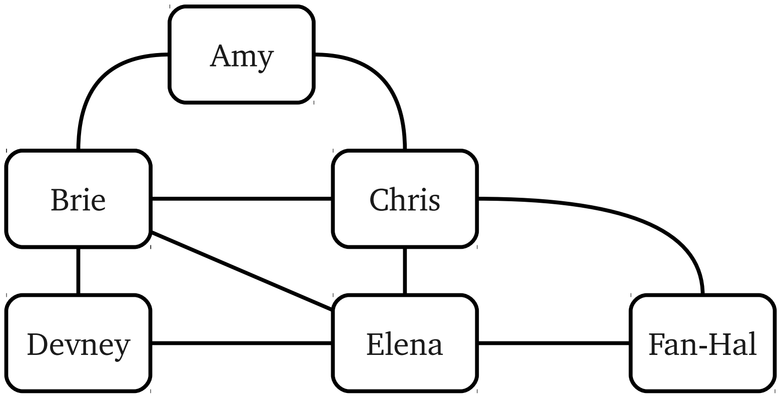
As shown in the sample outputs at the top of this page, the resulting image should be no wider than necessary. If the kern is zero, the width of the resulting image should be the width of the two individual images put together. As the kern increases, the width of the result image should decrease.

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

```
private boolean[][] kernLetters(boolean[][] first, boolean[][] second, int kern) {
```

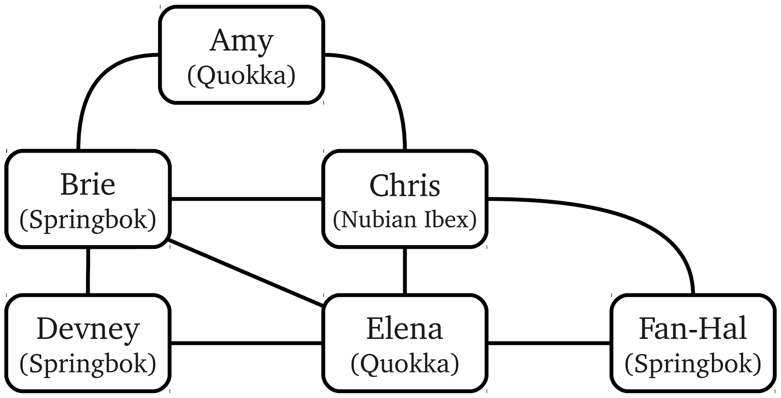
Problem Five: Animal Hipsters (10 Points)

Suppose that you have a social network represented as a graph, like this one here:



As in lecture, we will represent this graph as a `HashMap<String, ArrayList<String>>`, where each key in the `HashMap` is the name of a person and each value is an `ArrayList` of the names of the people they are friends with.

Let's suppose that every person in a social network has a favorite animal. We'll say that a person is an *animal hipster* if their favorite animal is different from all of their friends' favorite animals. For example, suppose that everyone's favorite animals are specified as follows:



Given the above social network, we would have that Amy, Chris, Elena, and Fan-Hal are animal hipsters, but Brie and Devney are not (because both of them like springboks and are they friends of one another). Although both Amy and Elena like quokkas, they are still animal hipsters because they are not friends of one another.

Write a method

```
private ArrayList<String> findAnimalHipsters(HashMap<String, ArrayList<String>> network,
                                             HashMap<String, String> favoriteAnimals)
```

that accepts as input a social network `network` and a `HashMap<String, String> favoriteAnimals` associating each person in the network with their favorite animal, then returns an `ArrayList<String>` containing all the people in the network who are animal hipsters.

(Continued on the next page)

In writing this method, you should assume the following:

- The `network` and `favoriteAnimals` `HashMaps` have the same set of keys, so every person in the graph has a favorite animal and everyone who has a favorite animal is in the graph.
- For simplicity, you can assume animal names are case-sensitive, so “Nubian Ibex” and “nubian ibex” should be treated as different animals.
- You are free to return the animal hipsters in any order that you'd like, though each animal hipster should appear in the list at most once.

Although we haven't seen this yet, you can iterate over the keys of a `HashMap` by using a range-based for loop by writing

```
for (String key: map.keySet()) {
     /* … */
}
```

Write your method in the space below.

```
private ArrayList<String> findAnimalHipsters(HashMap<String, ArrayList<String>> network,
                                             HashMap<String, String> favoriteAnimals) {
```