Second Practice Second CS106A Midterm

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the second midterm final exam.

Second Midterm Exam is Open Book, Open Notes, Closed Computer
The examination is open-book (specifically the course textbook *The Art and Science of Java*) and you may make use of any handouts, course notes/slides, printouts of your programs or other notes you've taken in the class. You may not, however, use a computer of any kind (i.e., you cannot use laptops on the exam).

Coverage
The second midterm exam covers the material presented throughout the class (with the exception of the Karel material). You are responsible for all topics covered in lectures up through and including Wednesday's lecture and for topics from the assignments.

General instructions
Answer each of the questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem. In all questions, you may include methods or definitions that have been developed in the course, either by writing the `import` line for the appropriate package or by giving the name of the method and the handout or textbook chapter number 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.

In an effort to save trees, the blank pages that would be provided in a regular exam for writing your solutions have been omitted from this practice exam.

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.” Your job is to write a method that, given a list of all the words in the English language, finds out what the longest isogram actually is. Write a method

```
private String longestIsogram(ArrayList<String> allWords)
```

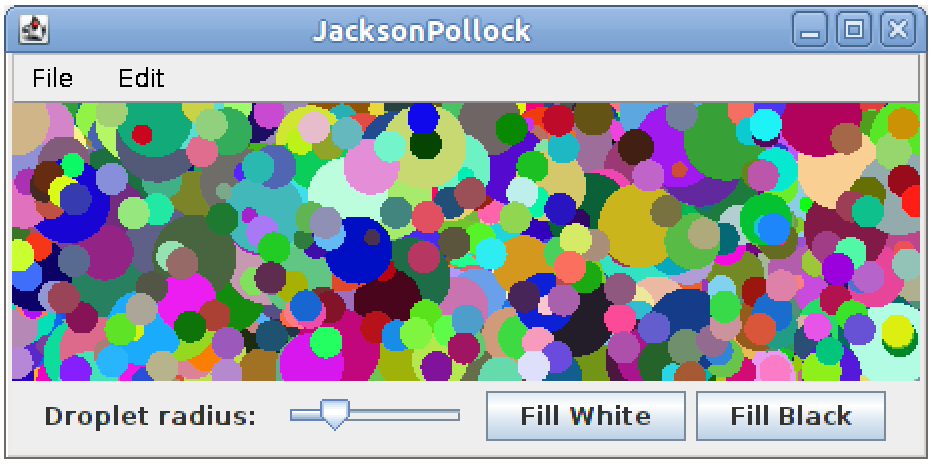
that accepts as input a **`ArrayList<String>`** containing all words in English (stored in lower-case) and returns the longest isogram in the list. If multiple words are tied as the longest isogram, feel free to return any one of them.

```
private String longestIsogram(ArrayList<String> allWords) {
```

Problem Two: Jackson Pollock (10 Points)

In this problem, you'll build a program that draws artwork in the style of the abstract expressionist painter Jackson Pollock. Pollock created paintings by laying the canvas down on the floor of his studio, then throwing paint of different colors onto it. The resulting paintings contain a mishmash of colors that are artistically and aesthetically interesting.

Your task is to write a program that simulates randomly-thrown droplets of colored paint landing on a canvas. Below is a screenshot of this program:



As soon as the program starts up, it begins drawing randomly-positioned circles on the canvas, each of which represents a drop of paint. The center of each circle is chosen as a random point inside the canvas, so the entire circle won't necessarily fit inside the window. In order to watch the art evolve over time, you should pause for **`PAUSE_TIME`** milliseconds after each drop of paint. Each circle's color should be chosen at random.

The radius of each circle should be determined by the value of a **`JSlider`** at the bottom of the window. The slider should range between the values **`MIN_RADIUS`** and **`MAX_RADIUS`**, and its default value should be **`DEFAULT_RADIUS`**. This slider should have a label to its left that reads "**Droplet radius:**" so that users understand what it controls.

If the user clicks the **Fill White** button, then the display should be filled with a solid white color, representing what would happen if you covered the canvas in a complete coat of white paint. The **Fill Black** button is similar, except that it will fill the canvas with black paint.

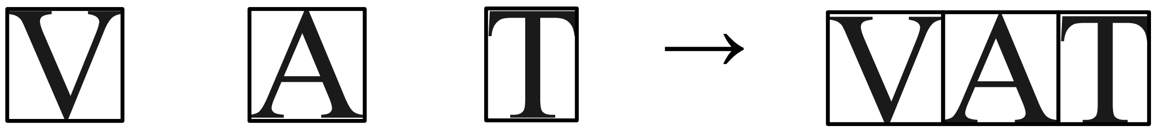
```
import /* … lots of imports … */;

public class JacksonPollock extends GraphicsProgram {
    /** Amount of time to pause between droplets, in milliseconds. */
    private static final double PAUSE_TIME = 1.0;

    /** Minimum, maximum, and default radius of each drop of paint. */
    private static final int MIN_RADIUS = 3;
    private static final int MAX_RADIUS = 20;
    private static final int DEFAULT_RADIUS = 7;
```

Problem Three: Kerning (10 Points)

Although we've used **`GLabel`** extensively in this class, we 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:

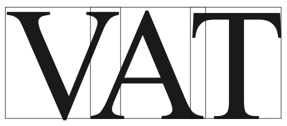


Unfortunately, this approach to laying out text will distort certain strings. For example, consider the following rendition of the string “THE VATICAN:”

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:

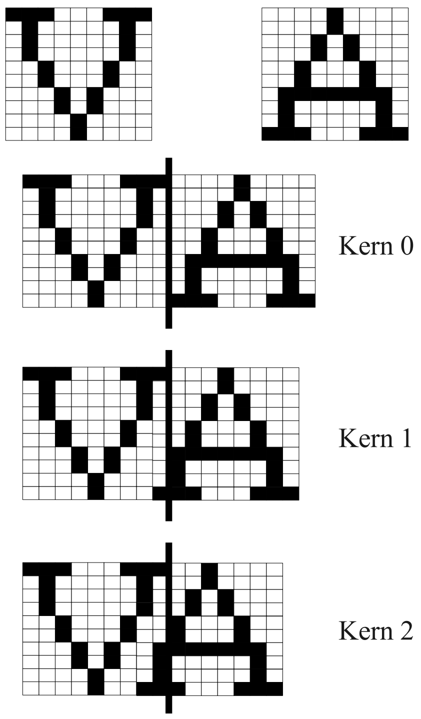


The amount that the images of two letters overlap is called the *kern*, and the process of overlapping letters this way is called *kerning*. Kerning can make text much more aesthetically pleasing. Compare the above rendition of “THE VATICAN,” which had no kerning, to this one, which has been kerned:

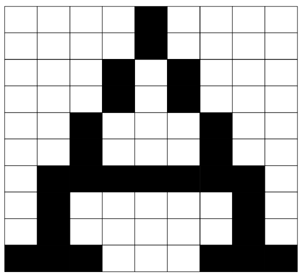
THE VATICAN

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:



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.

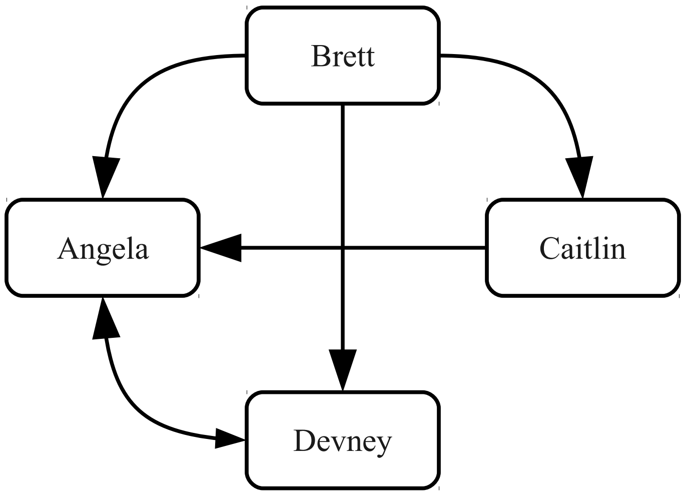
As shown in the sample outputs at the top of this page, the resulting image should be no wider than necessary. If the kern is zero, the width of the resulting image should be the width of the two individual images put together. As the kern increases, the width of the result image should decrease.

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

Problem Four: Finding Celebrities (10 Points)

In Facebook's social network, friendships are mutual: if person A is a friend of person B, then person B is a friend of person A. However, other personal relationships are not mutual. For example, the relationship "knows" might only go one way; if person A knows person B, person B might not know person A. We can represent who knows who as a graph: each node represents a person, and each edge from person A to person B represents that person A knows person B.

Let's call a person a *celebrity* if at least half of the people in a social network know that person. For example, consider this graph:



Here, Angela is a celebrity because 3 people know her (Brett, Caitlin, and Devney), and Devney is a celebrity because two people know her (namely, Angela and Brett). Caitlin is not a celebrity, since only one person knows her (Brett), and Brett is not a celebrity since no one knows him (though he somehow knows everyone else.)

Write a method

`private ArrayList<String> findCelebrities(HashMap<String, ArrayList<String>> graph)`

that accepts as input a **`HashMap<String, ArrayList<String>>`** representing the graph of who knows who, then returns an **`ArrayList<String>`** containing all the celebrities in the graph. You can assume that each person in the social network is a key in the map, even if they don't know anyone else (in which case their **`ArrayList<String>`** will be empty).

```
private ArrayList<String> findCelebrities(Map<String, List<String>> graph) {
```

Problem Five: I'm Feeling Lucky (10 Points)

If you'll recall from lecture, the PageRank algorithm assigns a score to each page on the web. The higher a site's PageRank, the more important the page. In our lecture example, we used PageRank to find the fifty most important pages on Wikipedia. Using PageRank to build a search engine requires a few more steps.

Here is a simplification of the algorithm that Google uses to perform searches:

1. First, Google compiles a list of all URLs for pages that contain the search query. These are URLs for pages likely to be relevant. All other URLs are ignored.
2. Next, Google filters this list of URLs by removing all URLs on a known blacklist (which usually contains malicious sites that steal personal information). This leaves a set of relevant URLs for reasonable sites.
3. Finally, Google sorts these URLs in descending order of their PageRank and displays the result.

In this problem, your job is to implement the following method, which returns the URL of the highest-rated page that Google would display for a search query:

```
private String imFeelingLucky(String searchQuery,
                              HashMap<String, String> textOfPages,
                              ArrayList<String> blacklistedURLs,
                              HashMap<String, Double> pageRank)
```

Here, the parameters are as follows:

- **`searchQuery`** is the user's search query.
- **`textOfPages`** is a **`HashMap<String, String>`** that stores all pages on the web. Each key in the map is a URL, and the value is the full text of the page with that URL.
- **`blacklistedURLs`** is a **`ArrayList<String>`** of URLs that have been blacklisted from appearing as a search result.
- **`pageRank`** is a **`HashMap<String, Double>`** from URLs to the PageRank score of the page with that URL. All PageRank scores are positive.

For this problem, you should check to see if the given search query appears on a page by checking, case-insensitively, if the search string appears verbatim anywhere in the page's text. For example, the phrase "cute cat" will match on a page with the phrase "CUTE CAT" on it. However, if you search for "cute cat" on a page whose text is "the cat was cute," that page shouldn't be added to the candidate set, since the exact string "cute cat" doesn't appear anywhere on the page.

If no pages can be returned – either because no pages contain the given search query, or because all the search results are blacklisted – your method should return **`null`** as a sentinel. If multiple pages are tied as the highest-rated page, then your method can return any one of them.

```
private String imFeelingLucky(String searchQuery,
                              HashMap<String, String> textOfPages,
                              ArrayList<String> blacklistedURLs,
                              HashMap<String, Double> pageRank) {
```