Second Practice CS106A Midterm Solutions

Problem One: Isograms (10 Points)

There are many solutions to this problem. Generally, the outline of the solution is to start off with a method like this:

```
private String longestIsogram(ArrayList<String> allWords) {
      String longest = "";
      for (String word: allWords) {
            if (isIsogram(word) && word.length() > longest.length()) {
                  longest = word;
            }
      }
      return longest;
}
```

Then to implement the `isIsogram` method to check whether or not a string is an isogram. There are many solutions to this problem; here are a few of them:

```
private boolean isIsogram(String word) {
  boolean[] used = new boolean[26];

  for (int i = 0; i < word.length(); i++) {
     char ch = word.charAt(i);

     if (used[ch - 'a']) return false;
     used[ch - 'a'] = true;
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);

    for (int j = i + 1; j < word.length();
         j++) {
      if (word.charAt(j) == ch) return false;
    }
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  String used = "";

  for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);

    if (used.indexOf(ch) != -1)
      return false;
    used += ch;
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);

    if (word.indexOf(ch, i + 1) != -1)
      return false;
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  for (char ch = 'a'; ch <= 'z'; ch++) {
    boolean found = false;
    for (int i = 0; i < word.length(); i++)
{
      if (word.charAt(i) == ch) {
        if (found) return false;
        found = true;
      }
    }
  }
  return true;
}
```

In case you're curious, the longest isogram in English is the word “subdermatoglyphic,” which means “pertaining to features of the skin determined by the layer of skin just below the surface.” Considering how long that description is, it's no wonder there's a word for it.

You can extend the idea of isograms to sentences that don't repeat any letters. Interestingly, there are several English sentences that use each letter exactly once. They're all pretty weird and either borrow from other languages or use acronyms that were later accepted as English words. For example:

Veldt jynx grimps waqf zho buck

As I'm typing this, my word processor is marking each of the above words (save for “buck”) as spelled incorrectly, though I promise they're real words. Honest.

Problem Two: Jackson Pollock (10 Points)

Here is one possible solution:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import acm.program.*;
import acm.graphics.*;
import acm.util.*;

public class JacksonPollock extends GraphicsProgram {
	/** Amount of time to pause between drops. */
	private static final double PAUSE_TIME = 0.5;

	/** The minimum, maximum, and default droplet radius. */
	private static final int MIN_RADIUS = 3;
	private static final int MAX_RADIUS = 20;
	private static final int DEFAULT_RADIUS = 7;

	/* The slider control that adjusts drop size. */
	private JSlider radiusSlider;

	public void init() {
		/* Label the slider. */
		add(new JLabel("Droplet radius: "), SOUTH);

		/* Construct and add the slider. */
		radiusSlider = new JSlider(MIN_RADIUS, MAX_RADIUS, DEFAULT_RADIUS);
		add(radiusSlider, SOUTH);

		/* Add the clear buttons and register them as listeners. */
		add(new JButton("Fill White"), SOUTH);
		add(new JButton("Fill Black"), SOUTH);
		addActionListeners();
	}

	public void actionPerformed(ActionEvent e) {
		/* Fill the canvas black or white as appropriate. */
		if (e.getActionCommand().equals("Fill White")) {
			removeAll();
			setBackground(Color.WHITE);
		} else if (e.getActionCommand().equals("Fill Black")) {
			removeAll();
			setBackground(Color.BLACK);
		}
	}
```

```
	public void run() {
		RandomGenerator rgen = RandomGenerator.getInstance();

		while (true) {
			/* Read the current radius from the slider. */
			double r = radiusSlider.getValue();

			/* Choose a random location for the center of the circle such
			 * that it fits into the window.
			 */
			double x = rgen.nextDouble(-r, getWidth() - r);
			double y = rgen.nextDouble(-r, getHeight() - r);

			/* Create the droplet. */
			GOval droplet = new GOval(x, y, r * 2, r * 2);
			droplet.setFilled(true);
			droplet.setColor(rgen.nextColor());
			add(droplet);

			pause(PAUSE_TIME);
		}
	}
}
```

Jackson Pollock's artwork is surprisingly subtle. Although his paintings appear totally random, other artists who tried to imitate his work often found that their paintings were nowhere near as elegant as Pollock's. There's some debate right now in the scientific community about why this is. Some researchers have alleged that Pollock's works display interesting fractal patterns, while his imitators' do not, though the findings are disputed.

Problem Three: Kerning (10 Points)

This was the algorithmically most difficult question on the exam. Probably the easiest way to solve this problem is to verbatim copy one image into the resulting image, then to copy just the black pixels from the second image into the result. A more difficult approach is to break the image into left, middle, and right pieces, copy the left and right verbatim, then blend the middle appropriately. The math here is a bit tricky either way. Below is code for the two solutions:

```
private boolean[][] kern(boolean[][] first, boolean[][] second, int kern) {
      /* Determine the size of the new image. */
      int numRows = first.length;
      int numCols = first[0].length + second[0].length - kern;

      /* Allocate the result. */
      boolean[][] result = new boolean[numRows][numCols];

      /* Copy the first image in verbatim. */
      for (int i = 0; i < first.length; i++) {
            for (int j = 0; j < first[i].length; j++) {
                  result[i][j] = first[i][j];
            }
      }

      /* Copy just the black pixels of the second image offset by some amount. */
      for (int i = 0; i < second.length; i++) {
            for (int j = 0; j < second[i].length; j++) {
                  /* If the pixel is black, copy it. */
                  if (second[i][j])
                        result[i][first[0].length - kern + j] = true;
            }
      }

      return result;
}
```

```
private boolean[][] kern(boolean[][] first, boolean[][] second, int kern) {
	/* Determine the size of the new image. */
	int numRows = first.length;
	int numCols = first[0].length + second[0].length - kern;

	/* Allocate the result. */
	boolean[][] result = new boolean[numRows][numCols];

	/* Copy the first piece of the first image in verbatim. */
	for (int i = 0; i < first.length; i++) {
		for (int j = 0; j < first[i].length - kern; j++) {
			result[i][j] = first[i][j];
		}
	}

	/* Copy the second piece of the second image in verbatim */
	for (int i = 0; i < second.length; i++) {
		for (int j = kern; j < second[i].length; j++) {
			result[i][first[0].length - kern + j] = second[i][j];
		}
	}

	/* Merge the two pieces together. */
	for (int i = 0; i < first.length; i++) {
		for (int j = 0; j < kern; j++) {
			if (first[i][first[0].length - kern + j] || second[i][j])
				result[i][first[0].length - kern + j] = true;
		}
	}

	return result;
}
```

IF YOU REALLY HATE SOMEONE, TEACH THEM TO RECOGNIZE BAD KERNING.

(xkcd)

Computerized typesetting has a fascinating history. Some major pioneers in the computer science, most notably Don Knuth, spent a lot of time and effort developing software to draw text clearly and elegantly. Steve Jobs is famous for investing effort to make the fonts on Apple computers look as elegant as possible. In fact, in his commencement address here at Stanford in 2005, Jobs specifically mentioned typography and making elegant fonts:

> Reed College at that time offered perhaps the best calligraphy instruction in the country. Throughout the campus every poster, every label on every drawer, was beautifully hand calligraphed. Because I had dropped out and didn't have to take the normal classes, I decided to take a calligraphy class to learn how to do this. I learned about serif and san serif typefaces, about varying the amount of space between different letter combinations, about what makes great typography great. It was beautiful, historical, artistically subtle in a way that science can't capture, and I found it fascinating.
>
> None of this had even a hope of any practical application in my life. But ten years later, when we were designing the first Macintosh computer, it all came back to me. And we designed it all into the Mac. It was the first computer with beautiful typography. If I had never dropped in on that single course in college, the Mac would have never had multiple typefaces or proportionally spaced fonts. And since Windows just copied the Mac, it's likely that no personal computer would have them. If I had never dropped out, I would have never dropped in on this calligraphy class, and personal computers might not have the wonderful typography that they do. Of course it was impossible to connect the dots looking forward when I was in college. But it was very, very clear looking backwards ten years later.
>
> Again, you can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future. You have to trust in something — your gut, destiny, life, karma, whatever. This approach has never let me down, and it has made all the difference in my life.

(Source: http://news.stanford.edu/news/2005/june15/jobs-061505.html)

On a less serious note: the word “keming” (with an 'm') is sometimes used to mean “excessive kerning.” Also, go and do a Google search for the word “kerning.” It's hilarious.

Problem Four: Finding Celebrities (10 Points)

There are many ways to attack this problem. One option, as shown below, is to loop over the people, then check for each person if at least half of the people in the network know them:

```
private ArrayList<String>
findCelebrities(HashMap<String, ArrayList<String>> graph) {
    ArrayList<String> result = new ArrayList<String>();
    for (String person: graph.keySet()) {
        /* Count how many people know this person. */
        int whoKnowsMe = 0;
        for (String otherPerson: graph.keySet()) {
            if (graph.get(otherPerson).contains(person))
                whoKnowsMe++;
        }
        /* See if more than half the people know this person. */
        if (whoKnowsMe >= (double)graph.size() / 2)
            result.add(person);
    }
    return result;
}
```

The second option is to make a table that says, for each person, how many other people know them. You can then check this table for very popular people:

```
private ArrayList<String>
findCelebrities(HashMap<String, ArrayList<String>> graph) {
    HashMap<String, Integer> numberKnowing = new HashMap<String, Integer>();

    for (String person: graph.keySet()) {
        for (String knownPerson: graph.get(person)) {
            if (numberKnowing.containsKey(knownPerson)) {
                numberKnowing.put(knownPerson, numberKnowing.get(knownPerson) + 1);
            } else {
                numberKnowing.put(knownPerson, 1);
            }
        }
    }

    ArrayList<String> result = new ArrayList<String>();
    for (String person: numberKnowing.keySet()) {
        if (numberKnowing.get(person) >= (double)graph.size() / 2)
            result.add(person);
        }
    }
    return result;
}
```

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

Here is one possible solution:

```
private String imFeelingLucky(String searchQuery,
                              HashMap<String, String> textOfPages,
                              ArrayList<String> blacklistedURLs,
                              HashMap<String, Double> pageRank) {
    String bestPage = null;
    for (String url: textOfPages.keySet()) {
        String pageText = textOfPages.get(url).toLowerCase();
        if (pageText.indexOf(searchQuery.toLowerCase()) != -1 &&
            !blacklistedURLs.contains(url) &&
            (bestPage == null || pageRank.get(url) > pageRank.get(bestPage)) {
            bestPage = url;
        }
    }

    return bestPage;
}
```

When Google actually does searches, it doesn't actually go through this whole procedure. After all, that would require searching the entire web for your keywords! Instead, Google typically does this computation for all the major searches in advance and caches the results (using something conceptually similar to the cache you wrote in Problem 4). That way, it doesn't have to recompute the search results every time you make a search; it can just look at what has actually been precomputed.

Another interesting question you might want to investigate is the following: for each page, is there a search query you could make that would cause Google to rank that page the highest?