Practice Second CS106A Midterm Exam

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: Oncogenes (18 Points)

Recall from the first midterm exam that a DNA sequence can be represented as a string of the letters A, C, T, and G. Let's define a ***gene*** to be a string of letters that appears somewhere within a DNA sequence. For example, **`CAT`** and **`AGCCA`** are genes in **`TAGCATCAGCCAG`**, but neither **`TCAT`** nor **`GAT`** are genes in this sequence.

Certain genes (called ***oncogenes***) are known to increase the risk of cancer. For the purposes of this problem, we'll consider an oncogene to be a gene that appears in a higher fraction of cancer cells than of normal healthy cells. For example, if a gene appears in 5% of cancer cells but only 1% of healthy cells, we would consider that gene to be an oncogene. On the other hand, if a gene appears in 10% of cancer cells and 10% of normal cells, we would not consider it an oncogene.

Suppose that you have a gene that you suspect may be an oncogene. To determine whether it is, you gather DNA from healthy cells and cancer cells, then store the DNA in two **`ArrayList<String>`**s, one holding DNA from healthy cells and one holding DNA from cancer cells.

Write a method

```
private boolean isOncogene(ArrayList<String> healthySequences,
                           ArrayList<String> cancerSequences,
                           String candidate)
```

that accepts as input two **`ArrayList<String>`**s representing DNA samples from healthy and cancer cells, along with a **`String`** that you suspect is an oncogene. Your method should then return whether that gene is an oncogene (that is, whether it appears a higher fraction of the time in the cancer DNA sequences than in the healthy DNA sequences).

As an example, if you wanted to check whether **`TGC`** was an oncogene given the following samples:

Healthy Cell DNA	Cancer Cell DNA
`TGCATCC`	**`ATTTGCAGG`**
`AAATTTGGGCCC`	**`TGCAAATTA`**
`ATGCGCTA`	**`AAAGGGCCCTTT`**
`GGGTACGGAG`	**`TGCGATACGTAGGACCA`**
`TTAATTGGG`	**`ACTCATTAGTGC`**
	`AAACGCTAGACACACAAG`

Your method would return **`true`**, because the sequence **`TGC`** appears in only 40% of healthy DNA sequences (2 of 5) but in 66.7% of cancer cell DNA sequences (4 of 6).

However, given the gene **`GGG`**, your method would return **`false`**, because **`GGG`** appears 60% of the time in healthy cells (3 of 5) and only 16.7% of the time in cancer cells (1 of 6).

(continued on next page)

You should assume the following:

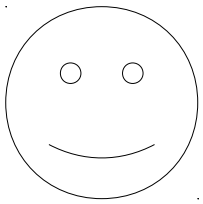
- All of the strings given as input are made up purely of the characters A, T, C, and G.
- The **`healthySequences`** and **`cancerSequences`** lists are guaranteed to be nonempty, though they do not have to be the same length.
- You should not modify any of the input lists.
- When determining whether a gene is present within a DNA sequence, you do not need to count the number of times that gene appears within the sequence. All that matters is whether or not that gene is present at all.

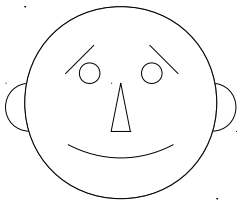
```
private boolean isOncogene(ArrayList<String> healthySequences,
                           ArrayList<String> cancerSequences,
                           String candidate) {
```

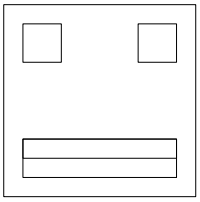
Problem Two: Stick Figure Factory (20 Points)

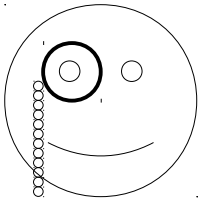
In this problem, you will create a program that will let users mix and match stick figure heads and bodies to create an assortment of different stick figures.

Suppose that you have two arrays of **`String`**s holding the names of image files – an array **`HEADS`** of names of files containing possible stick figure heads and an array **`BODIES`** containing names of files containing possible stick figure bodies. For example, here is one possible set of pictures of stick figure heads:


default.png

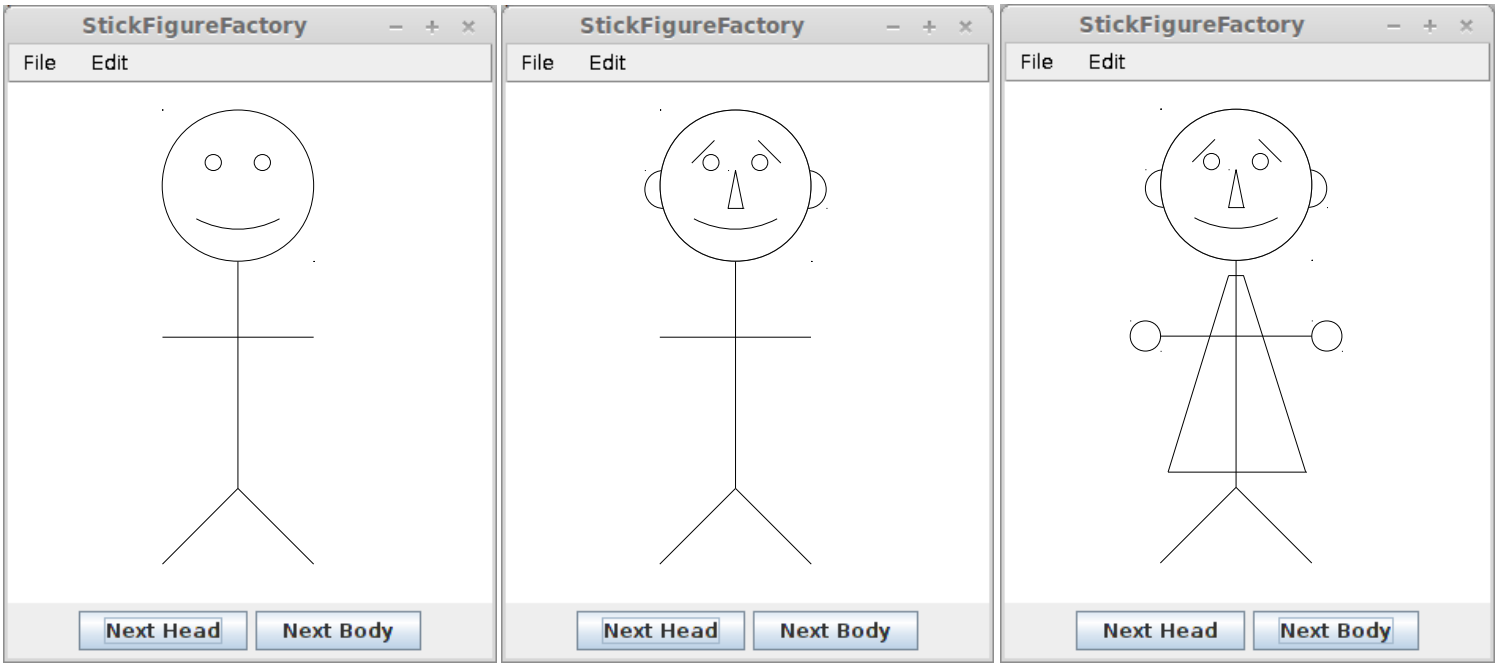

happy.png


robot.png


monocle.png

When the program starts up, initially it shows a stick figure made of two **`GImage`**s – one image for the head and one for the body. There are also two buttons in the south border, one labeled "Next Head" and one labeled "Next Body." When the user clicks the "Next Head" button, the program changes which head is displayed to the next head in the list, wrapping back around to the start if there are no heads left. Similarly, clicking the "Next Body" button will cause the program to change which body is currently being displayed to the next in the sequence, wrapping around if there are no bodies left.

For example, here is a series of screenshots representing the program at startup, the result of clicking the "Next Head" button, and the result of then clicking the "Next Body" button:



(Continued on next page)

In writing this program, you should assume the following:

- The head image should have its upper left corner at position (**HEAD_X**, **HEAD_Y**). Similarly, the body image should have its upper left corner at position (**BODY_X**, **BODY_Y**).
- You do not need to resize the **GImage**s – their default sizes are perfectly fine.
- The head and body images should initially be the images whose filenames are stored at index 0 in the **HEADS** and **BODIES** arrays.
- The **HEADS** and **BODIES** arrays are nonempty, but may have different sizes.
- If you want to change the displayed image by adding a new **GImage**, you should remove the old **GImage** first.

Write your solution below.

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

import java.awt.event.*;
import javax.swing.*;

public class StickFigureFactory extends GraphicsProgram {
      private static final String[] HEADS  = { /* … omitted … */ };
      private static final String[] BODIES = { /* … omitted … */ };

      private static final double HEAD_X = /* … omitted … */;
      private static final double HEAD_Y = /* … omitted … */;
      private static final double BODY_X = /* … omitted … */;
      private static final double BODY_Y = /* … omitted … */;
```

Problem Three: The Neverending Birthday Party (26 Points)

Suppose you want to hold a neverending birthday party, where every day of the year someone at the party has a birthday. How many people do you need to get together to have such a party?

Your task in this program is to write a program that simulates building a group of people one person at a time. Each person is presumed to have a birthday that is randomly chosen from all possible birthdays. Once it becomes the case that each day of the year, someone in your group has a birthday, your program should print out how many people are in the group, then should exit.

In writing your solution, you should assume the following:

- There are 366 possible birthdays (this includes February 29).
- All birthdays are equally likely, including February 29.

You might find it useful to represent birthdays as integers between 0 and 365, inclusive.

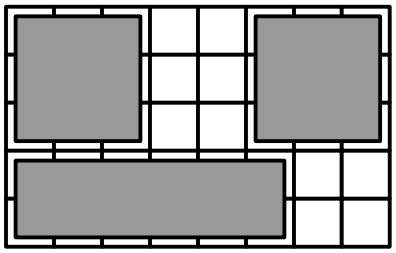
```
import acm.program.*;

public class NeverendingBirthdayParty extends ConsoleProgram {
```

Problem Four: Moving Day (32 Points)

Suppose that you have a bunch of rectangular boxes that you are trying to fit into a rectangular truck. You have already packed most of your boxes into the truck, and you are trying to see if you can fit one more box in.

For simplicity, we'll assume that the length and width of each box is a whole number of feet, and that the length and width of the truck is a whole number of feet. Given this setup, we can represent the available space in the truck as a two-dimensional array of **`boolean`**s, one for each square foot of space in the truck. If the location is occupied, the boolean is set to **`true`**, and if the location is unoccupied, the location is **`false`**. For example, below is one possible arrangement of boxes in the truck and its corresponding array representation:



Truck Layout

T	T	T	F	F	T	T	T
T	T	T	F	F	T	T	T
T	T	T	F	F	T	T	T
T	T	T	T	T	T	F	F
T	T	T	T	T	T	F	F

Array Representation

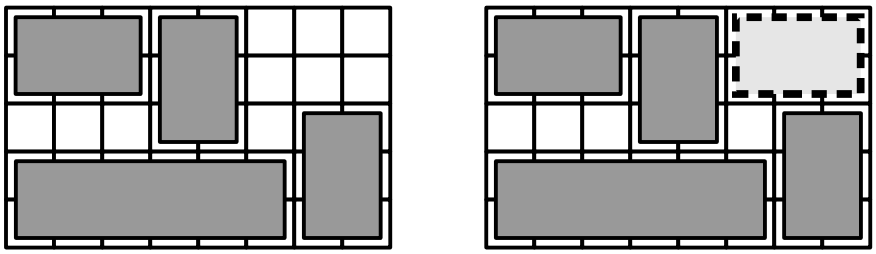
Your job is to write a method

```
private boolean canPlaceBox(boolean[][] truckLayout, int length, int width)
```

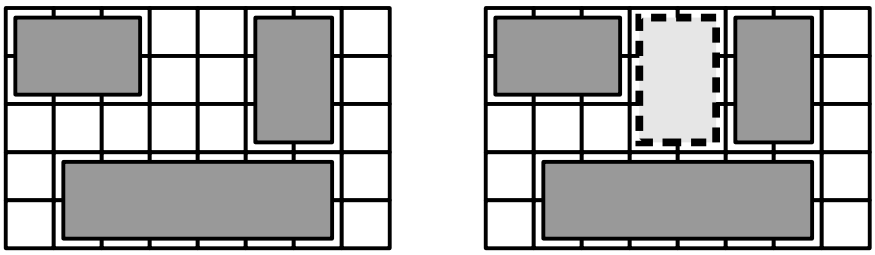
that accepts as input the current layout of the truck (represented as a **`boolean[][]`**) and the dimensions of a new box (given by its length and width, in feet), then returns whether if there is space left in the truck to place this box without moving any of the other boxes. You can assume the following:

- Boxes cannot be stacked on top of one another.
- The box must have its sides parallel to the sides of the truck, but may be rotated 90º.
- You cannot move or rotate any of the existing boxes in the truck.
- The dimensions of the new box will be an arbitrary positive number of feet, and could potentially be larger than the truck itself.
- You should not update the **`truckLayout`** array. You just need to return whether it's possible to place the box and don't need to update the representation.

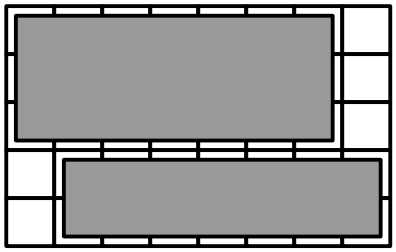
For example, given the following configuration and a 3' × 2' box, your method should return **`true`** because the box can be placed at the indicated location:



Similarly, given the following configuration and a 3' × 2' box, your method should return **`true`** because the box can be rotated and placed into the specified position:

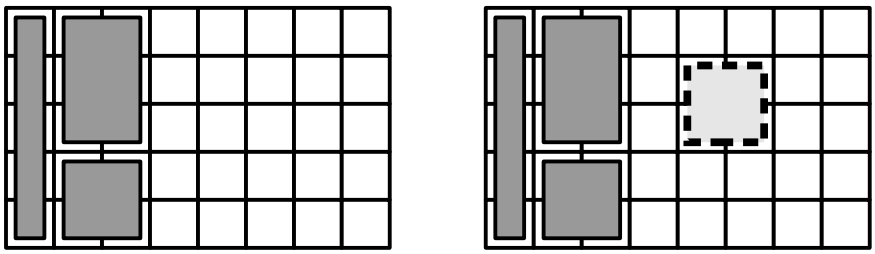


However, given the following configuration and a 1' × 5' box, your method should return **`false`** because the box will not fit, even if it is rotated:



(Note that the box could be made to fit by moving one of the other boxes over, though for the purposes of this problem we're only curious whether you can place the box without moving or rotating any other boxes.)

Finally, given the following configuration and a 2' × 2' box, your method should return **`true`** because the box can fit in many different places, such as at the indicated position:

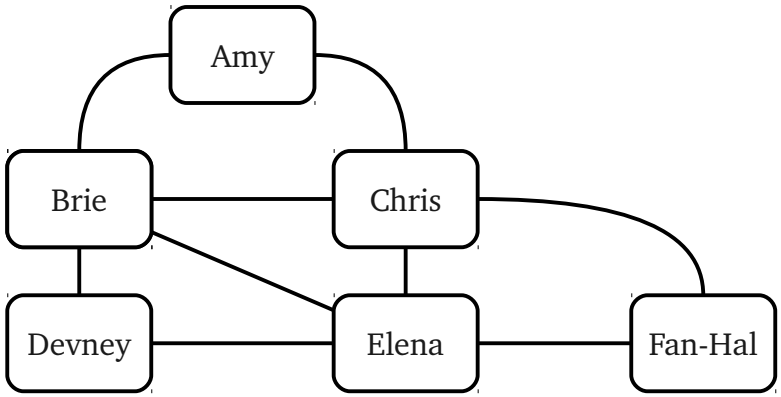


In other words, the box does not have to be "snug" up against other boxes. It just has to fit in the truck.

```
private boolean canPlaceBox(boolean[][] truckLayout, int length, int width) {
```

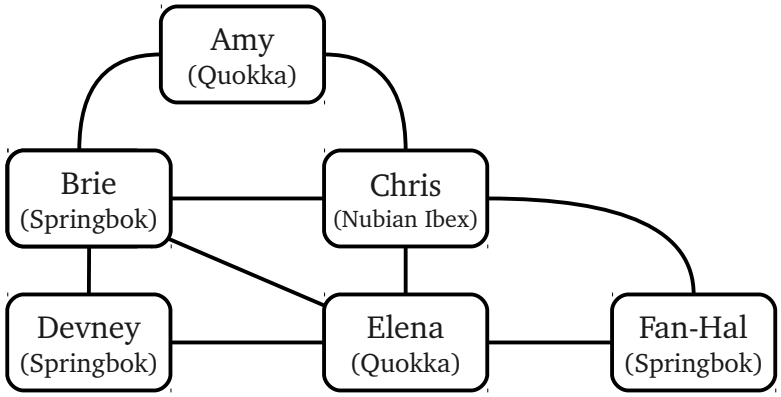
Problem Five: Animal Hipsters (24 Points)

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). Even though 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 HashMap`**s 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.

Write your method in the space below.

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