CS106A Handout 27S
Winter 2013-2014 February 28, 2014

Practice Second CS106A Midterm Solutions

Problem One: Oncogenes (18 Points)

```
private boolean isOncogene(ArrayList<String> healthySequences,
                           ArrayList<String> cancerSequences,
                           String candidate) {
    return fractionContaining(healthySequences, candidate) <
           fractionContaining(cancerSequences, candidate);
}

private double fractionContaining(ArrayList<String> sequences, String gene) {
    int result = 0;
    for (String sequence: sequences) {
        if (sequence.indexOf(gene) != -1) {
            result++;
        }
    }
    return (double)result / sequences.size();
}
```

Common mistakes included forgetting to include a cast to avoid performing integer division, using the wrong method to check if the gene was a substring of the overall sequence, and assuming that genes had to be exactly three characters long.

Problem Two: Stick Figure Factory (20 Points)

```
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 … */;

      private int headIndex = 0;
      private int bodyIndex = 0;

      public void init() {
          redrawAll();
          add(new JButton("Next Head"), SOUTH);
          add(new JButton("Next Body"), SOUTH);
          addActionListeners();
      }

      private void redrawAll() {
          removeAll();
          add(new GImage(HEADS[headIndex],  HEAD_X, HEAD_Y);
          add(new GImage(BODIES[bodyIndex], BODY_X, BODY_Y);
      }

      public void actionPerformed(ActionEvent e) {
          if (e.getActionCommand().equals("Next Head")) {
              headIndex = (headIndex + 1) % HEADS.length;
              redrawAll();
          } else if (e.getActionCommand().equals("Next Body")) {
              bodyIndex = (bodyIndex + 1) % BODIES.length;
              redrawAll();
          }
      }
}
```

Common mistakes included treating the **`HEADS`** and **`BODIES`** arrays as if they were arrays of **`GImage`**s rather than strings, minor off-by-one errors when wrapping around from the last head/body to the very first, and incorrectly looping inside **`actionPerformed`**.

Problem Three: The Neverending Birthday Party (26 Points)

```
import acm.program.*;
import acm.util.*;
public class NeverendingBirthdayParty extends ConsoleProgram {
    public void run() {
        RandomGenerator rgen = RandomGenerator.getInstance();
        boolean[] used = new boolean[366];
        int numLeft = 366;
        int numPeople = 0;

        while (numLeft > 0) {
            int birthday = rgen.nextInt(0, 365);
            if (!used[birthday]) {
                 numLeft--;
                 used[birthday] = true;
            }
            ++numPeople;
        }
        println("We needed " + numPeople + " in our group.");
    }
}
```

Note that there are *many* ways to do this – you could use a **`HashSet`** of the used birthdays so far, or store everything in an **`ArrayList`**, etc.

Common mistakes included only counting people who had unique birthdays (rather than everyone), accidentally stopping too early, and minor errors tracking which birthdays have been used.

Interestingly, it's possible to compute the exact value of the expected number of people necessary to have the neverending birthday party. It's approximately equal to $366 \ln 366 \approx 2160$. So even though CS106A is enormous this quarter, it's still much lower than the expected number of people you'd need.

Problem Four: Moving Day (32 Points)

```
private boolean canPlaceBox(boolean[][] truckLayout, int length, int width) {
    return canPlaceBoxOriented(truckLayout, length, width) ||
           canPlaceBoxOriented(truckLayout, width, length);
}

private boolean canPlaceBoxOriented(boolean[][] layout, int length, int width) {
    int numRows = layout.length;
    int numCols = layout[0].length;

    for (int row = 0; row < numRows - (length - 1); row++) {
        for (int col = 0; col < numCols - (width - 1); col++) {
            if (fitsHere(layout, row, col, length, width)) {
                return true;
            }
        }
    }
    return false;
}
private boolean fitsHere(boolean[][] layout, int row, int col,
                         int length, int width) {
    for (int i = row; i < row + length; i++) {
        for (int j = col; j < col + width; j++) {
            if (layout[i][j] == true) return false;
        }
    }
    return true;
}
```

The most common mistakes we saw were logic errors in which your method might accidentally report that the box fit when it does not. The following strategies do not accurately report whether the box will fit:

- Determining the length of the longest horizontal and vertical stretches of free space in the truck, then seeing if the box would fit in a rectangle made from these dimensions. This doesn't work because there is no guarantee that those open stretches are adjacent, or that the box they define is actually open space.
- Counting the number of open spaces in the truck and reporting whether this number is at least the size of the box. This doesn't check to see if the free spaces are actually arranged in a way that would let the box fit.

Other common mistakes included loop indexing errors, accidentally walking off of the truck layout array, and reporting that the box does not fit if a single location does not pan out, rather than if all locations do not pan out.

Problem Five: Animal Hipsters (24 Points)

```
private ArrayList<String>
findAnimalHipsters(HashMap<String, ArrayList<String>> network,
                   HashMap<String, String> favoriteAnimals) {
    ArrayList<String> result = new ArrayList<String>();
    for (String person: network.keySet()) {
        boolean isHipster = true;
        for (String friend: network.get(person)) {
            if (favoriteAnimals.get(person).equals(favoriteAnimals.get(friend)) {
                isHipster = false;
            }
        }
        if (isHipster) result.add(person);
    }
    return result;
}
```

Common mistakes included incorrectly looping over all the people in the graph, comparing strings using `==` instead of `.equals()`, mixing up the keys and values in the maps, and checking whether someone was a hipster by seeing if *anyone* in the network had the same favorite animal (and not just the person's friends).