Mehran Sahami
CS106A

Handout #50
November 30, 2015

Additional Practice Problem Solutions

These additional problem solutions are courtesy of Keith Schwarz

Problem 1: Strings (15 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 four 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);
    if (word.indexOf(ch, i + 1) != -1)
      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);
    for (int j = i + 1; j < word.length();
         j++) {
      if (word.charAt(j) == ch) {
        return false;
      }
    }
  }
  return true;
}
```

Problem 2: Graphics and Interactivity (25 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 droplets, in milliseconds. */
    private static final double PAUSE_TIME = 1.0;

   /** 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);
      }
   }
```

Solutions continues on next page

```
   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);
      }
   }
}
```

Problem 3: The Never-ending Birthday Party (25 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.");
    }
}
```

Problem 4: Arrays (25 points)

```
/** Method: isMagicSquare
 * This method checks an n x n grid to see if it's a magic square.
 */
private boolean isMagicSquare(int[][] matrix, int n) {
  /* A 0 x 0 square is valid, in a weird way. */
   if (n == 0) return true;

  /* If we don't see all numbers in the range 1 to n^2, we can report
   * failure.                                                        */
   if (!allExpectedNumbersFound(matrix, n)) return false;

  /* Sum up the first row to get its value. */
   int expected = rowSum(matrix, 0, n);

  /* Check that all rows and columns have this value. */
   for (int i = 0; i < n; i++) {
      if (rowSum(matrix, i, n) != expected ||
          colSum(matrix, i, n) != expected)
         return false;
   }
   return true;
}

/** Method: allExpectedNumbersFound
 * This method returns whether all the numbers 1 … n^2 are present in the
 * given grid.
 */
private boolean allExpectedNumbersFound(int[][] square, int n) {
/* Make an array of n^2 + 1 booleans to track what numbers are found.  The
 * +1 is because the numbers range from 1 to n^2 and we have to ensure that
 * there's sufficient space.
 */
   boolean[] used = new boolean[n * n + 1];

   /* Iterate across the grid and ensure that we've seen everything. */
   for (int row = 0; row < n; row++) {
      for (int col = 0; col < n; col++) {
         /* Make sure the number is in range. */
         if (square[row][col] < 1 || square[row][col] > n * n)
            return false;

         /* Make sure it isn't used. */
         if (used[square[row][col]])
            return false;

         /* Mark the square used. */
         used[square[row][col]] = true;
      }
   }
   /* At this point, we know that all numbers are in range and there are
    * no duplicates, so everything is valid.
    */
   return true;
}
```

```
/** Method: rowSum
 * Returns the sum of the given row of the grid.
 */
private int rowSum(int[][] grid, int row, int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      sum += grid[row][i];
   }
   return sum;
}

/** Method: colSum
 * Returns the sum of the given column of the grid.
 */
private int colSum(int[][] grid, int col, int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      sum += grid[i][col];
   }
   return sum;
}
```