Eric Roberts
CS 106A

Handout #63
March 5, 2010

Practice Final Examination #1

Review session: **Sunday, March 14, 7:00–9:00P.M. (Hewlett 201)**

Scheduled finals: **Monday, March 15, 12:15–3:15P.M. (Hewlett 200)**
Friday, March 19, 12:15–3:15P.M. (Hewlett 200)

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the final examination. A solution set to this practice examination will be handed out at the optional class on Monday. (CS 106A does not hold required classes during Dead Week.)

Time of the exam

The final exam is scheduled at two different times during exam period, as shown at the top of this handout. Both finals are in the regular classroom. You may take the exam at either of the two scheduled times and need not give advance notice of which exam you plan to take. If you are unable to take the exam at either of the scheduled times, please send an e-mail message to **`eroberts@cs`** stating the following:

- The reason you cannot take the exam at the scheduled time.
- A three-hour period on Monday or Tuesday of exam week at which you could take the exam. This time must be during the regular working day and must therefore start between 8:30 and 2:00.

In order to take an alternate exam, I must receive this message from you by 5:00P.M. on Wednesday, March 10. Replies will be sent by electronic mail on Friday, March 12.

Review session

Several section leaders will conduct a review session on Sunday evening from 7:00 to 9:00P.M. in Hewlett 201. We'll will announce the winners of the Adventure Contest and hold the random grand-prize drawing at the beginning of the review session.

Coverage

The exam covers the material presented in class through today, which means that you are responsible for the material in Chapters 1 through 13 of *The Art and Science of Java,* with the following exceptions:

- *Chapter 7*. There will not be any questions on the low-level representation of data as bits or any problems involving heap-stack diagrams.
- *Chapter 10*. In this chapter, you are responsible for the material in sections 10.1, 10.2, 10.3, 10.5, and 10.6. The only Swing interactors you will be expected to use are the ones from FacePamphlet: **`JLabel`**, **`JButton`**, and **`JTextField`**.
- *Chapter 11*. You are responsible for all the material in this chapter except for the bit-manipulation operators in section 11.7. Image-manipulation problems that use no bit operations (such as the **`FlipHorizontal`** problem from Section #6) are fair game.
- *Chapter 12*. You are responsible for the general idea of searching and sorting, as presented in sections 12.1 and 12.2, including the binary search and selection sort algorithms. The coverage of data files will be limited to reading a file line by line.
- *Chapter 13*. You should understand how to use the **`ArrayList`**, **`HashMap`**, and **`TreeMap`** classes, but need not understand their implementation.

General instructions

The instructions that will be used for the actual final look like this:

> Answer each of the questions given below. 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. The total number of points is 90. We intend that the number of points be roughly equivalent to the number of minutes someone who is completely on top of the material would spend on that problem. Even so, we realize that some of you will still feel time pressure. If you find yourself spending a lot more time on a question than its point value suggests, you might move on to another question to make sure that you don't run out of time before you've had a chance to work on all of them.
>
> 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 chapter number in which that definition appears.
>
> The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Note: To conserve trees, I have cut back on answer space for the practice exams. The actual final will have much more room for your answers and for any scratch work.

Please remember that the final is <u>open-book.</u>
Monday, March 15, 12:15–3:15p.m. (Hewlett 200)
Friday, March 19, 12:15–3:15p.m. (Hewlett 200)

Problem 1—Short answer (10 points)

1a) Suppose that the integer array **`list`** has been declared and initialized as follows:

```
private int[] list = { 10, 20, 30, 40, 50 };
```

This declaration sets up an array of five elements with the initial values shown in the diagram below:

`list`

10	20	30	40	50

Given this array, what is the effect of calling the method

```
mystery(list);
```

if **`mystery`** is defined as:

```
private void mystery(int[] array) {
   int tmp = array[array.length - 1];
   for (int i = 1; i < array.length; i++) {
      array[i] = array[i - 1];
   }
   array[0] = tmp;
}
```

Work through the method carefully and indicate your answer by filling in the boxes below to show the final contents of **`list`**:

`list`

1b) Suppose that you have been assigned to take over a project from another programmer who has just been dismissed for writing buggy code. One of the methods you have been asked to rewrite has the following comment and prototype:

```
/* Method: insertValue(value, array) */
/**
 * Inserts a new value into a sorted array of integers by
 * creating a new array that is one element larger than the
 * original array and then copying the old elements into the
 * new array, inserting the specified value at its proper
 * position.
 *
 * @param value The value to be inserted into the array
 * @param array An array sorted in ascending order
 * @return A newly allocated array with value inserted in order
 */
  private int[] insertValue(int value, int[] array) {
     int[] result = new int[array.length + 1];

     for (int i = 0; i < result.length; i++) {
        result[i] = array[i];
     }

     int pos = 0;
     for (int i = 0; i < array.length; i++) {
        if (value > array[i]) {
           pos = i;
           break;
        }
     }

     for (int i = result.length; i >= pos; i--) {
        result[i] = result[i - 1];
     }

     result[pos] = value;

     return result;
  }
```

Unfortunately, the code contains several bugs. Circle the bugs in the implementation and write a short sentence explaining the precise nature of each problem you identify.

Problem 2—Using the `acm.graphics` library (15 points)

Although the definition of filling for an arc (see page 314 in the text) is not necessarily what you would want for all applications, it turns out to be perfect for the problem of displaying a traditional pie chart. Your job in this problem is to write a method

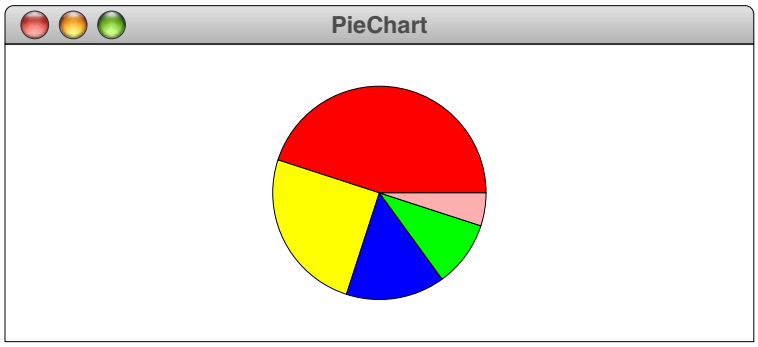
```
private GCompound createPieChart(double r, double[] data)
```

that creates a **`GCompound`** object that displays a pie chart for the specified data values. The parameter **`r`** represents the radius of the circle in which the pie chart is displayed, and **`data`** is the array of data values you want to plot in the chart.

The operation of the **`createPieChart`** method is easiest to illustrate by example. If you execute the **`run`** method

```
public void run() {
   double[] data = { 45, 25, 15, 10, 5 };
   GCompound pieChart = createPieChart(50, data);
   add(pieChart, getWidth() / 2, getHeight() / 2);
}
```

your program should generate the following pie chart in the center of the window:



The red wedge corresponds to the 45 in the data array and extends 45% around the circle, which is not quite halfway. The yellow wedge then picks up where the red wedge left off and extends for 25% of a complete circle. The blue wedge takes up 15%, the green wedge takes up 10%, and the pink wedge the remaining 5%.

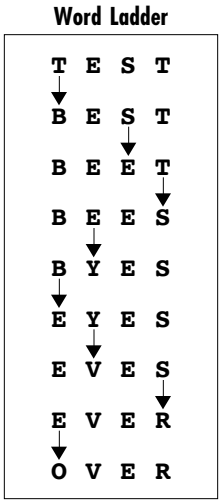
As you write your solution to this problem, you should keep the following points in mind:

- The values in the array are not necessarily percentages. What you need to do in your implementation is to divide each data value by the sum of the elements to determine what fraction of the complete circle each value represents.
- The colors of each wedge are specified in the following constant array:

  ```
  private static Color[] WEDGE_COLORS = {
     Color.RED, Color.YELLOW, Color.BLUE, Color.GREEN,
     Color.PINK, Color.CYAN, Color.MAGENTA, Color.ORANGE
  };
  ```

  If you have more wedges than colors, you should just start the sequence over, so that the eighth wedge would be red, the ninth yellow, and so on.
- The reference point of the **`GCompound`** returned by **`createPieChart`** must be the center of the circle.

Problem 3—Strings (15 points)



A **word-ladder puzzle** is one in which you try to connect two given words using a sequence of English words such that each word differs from the previous word in the list only in one letter position. For example, the figure at the right shows a word ladder that turns the word **`TEST`** into the word **`OVER`** using eight single-letter steps.

In this problem, your job is to write a program that checks the correctness of a word ladder entered by the user. (In CS 106B, you will learn how to write a program that finds word ladders.) Your program should read in a sequence of words and make sure that each word in the sequence follows the rules for word ladders, which means that each line entered by the user must

1. Be a legitimate English word
2. Have the same number of characters as the preceding word
3. Differ from its predecessor in exactly one character position

Implementing the first condition requires that you have some sort of dictionary available, which is well beyond the scope of a 15-minute problem. You may therefore assume the existence of a **`Lexicon`** class that exports the following method

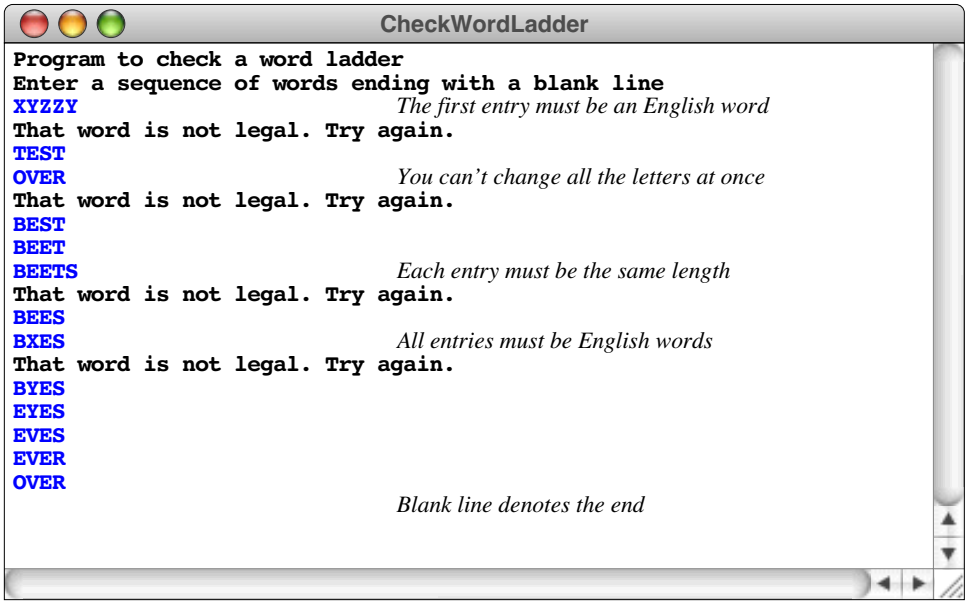
```
public boolean isEnglishWord(String str)
```

which takes a word and returns **`true`** if that word is in the lexicon. You may also assume that you have access to such a dictionary via the following instance variable declaration:

```
private Lexicon lexicon = new Lexicon("Dictionary.txt");
```

All words in the lexicon are in upper case.

If the user enters a word that is not legal in the word ladder, your program should print out a message to that effect and let the user enter another word. It should stop reading words when the user enters a blank line. Thus, your program should be able to duplicate the following sample run (the italicized messages don't appear but are there to explain what's happening):



Problem 4—Arrays (10 points)

If you spent any time at all playing your Yahtzee program, you know that it would be wonderful if there were a two-pair category for all the times you just missed getting a full house. Write a Java method

```
private boolean checkTwoPairCategory(int[] dice)
```

that takes an array of five die values and returns **`true`** if and only if the values in the **`dice`** array contain two pairs and a fifth value that does not match the others.

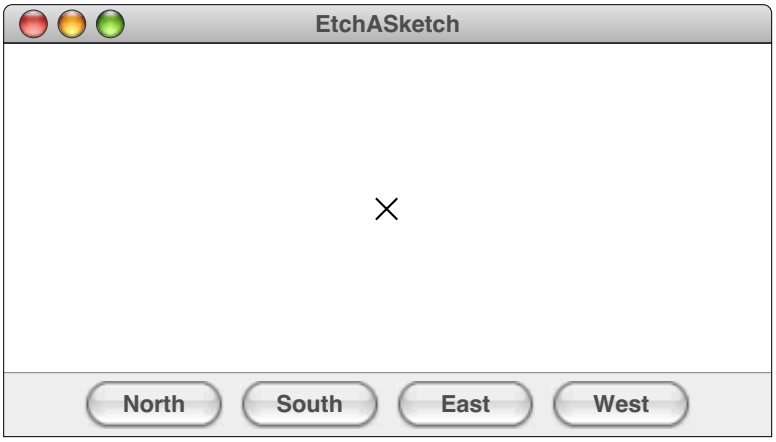
In writing your answer to this problem, you should keep the following points in mind:

- You do not have to check that the array contains five elements or that the values in the array fall between 1 and 6.
- You may not change or reorder the values in the array you have been passed by the caller. You may, however, create temporary arrays as part of your implementation.
- A set of dice fits the two-pair category only when the paired values are different and the fifth value doesn't match any of the others.

Problem 5—Building graphical user interfaces (15 points)

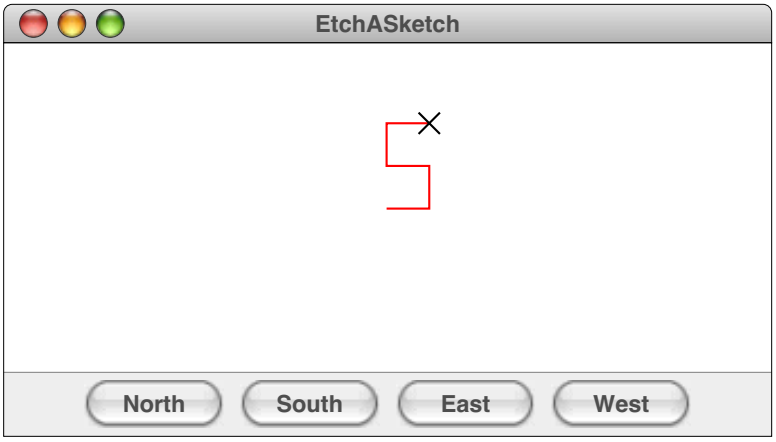
Write a **`GraphicsProgram`** that does the following:

1. Add buttons to the south region labeled **`"North"`**, **`"South"`**, **`"East"`**, and **`"West"`**.
2. Create an X-shaped cross ten pixels wide and ten pixels high.
3. Adds the cross so that its center is at the center of the graphics canvas. Once you have completed these steps, the display should look like this:

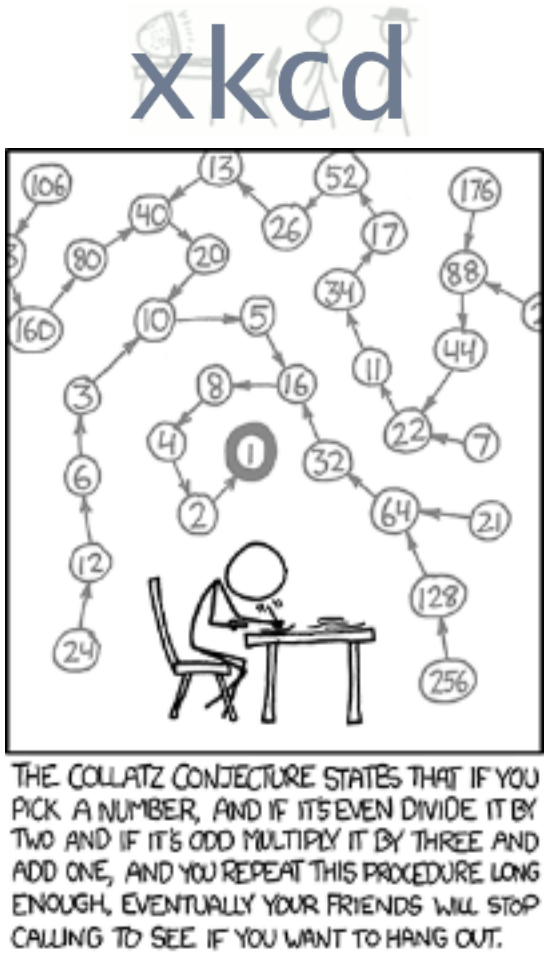


4. Implement the actions for the button so that clicking on any of these buttons moves the cross 20 pixels in the specified direction. At the same time, your code should add a red **`GLine`** that connects the old and new locations of the pen.

Keep in mind that each button click adds a new **`GLine`** that starts where the previous one left off. The result is therefore a line that charts the path of the cross as it moves in response to the buttons. For example, if you clicked **`East`**, **`North`**, **`West`**, **`North`**, and **`East`** in that order, the screen would show a Stanford "S" like this:



Problem 6—Using Java collections (15 points)



In the **xkcd** strip that appeared yesterday, Randall Monroe makes reference to the hailstone sequence that you learned about back in Assignment #2 (the *Collatz Conjecture* to which he refers is the as-yet-unresolved claim that this process always terminates). As the cartoon makes clear, many of the sequences include common patterns. For example, all hailstone sequences (except for 1, 2, 4, and 8, of course) go through the value 16 and therefore share the last four elements of the path. If you've followed through this path once, you don't need to follow it again.

Keeping track of these common patterns can make calculating the length of hailstone sequences vastly more efficient. Suppose, for example, that you are calculating the number of steps in all the hailstone sequences between 1 and 100. What happens when you get to 24, which appears at the lower left corner of the cartoon? By the time you get to this point, you've already figured out that it takes nine steps to get from 12 down to 1, so that 24 must take ten steps, given that 12 is just one step away from 24.

It's even more interesting to follow what happens with 7, which is in the middle of the right hand side. Here, figuring out the number of steps for 7 entails running through the number 22, 11, 34, 17, 52, 26, 13, 40, and 20 before you come to 10, which you've already encountered in counting the number of steps from 3 and 6. At this point, you have learned an enormous amount. Given that it takes six steps to reach 1 starting at 10, you know that 20 takes seven steps, 40 takes eight, 13 takes nine, and so forth, backwards along the list of numbers in the chain, until you determine that there are sixteen steps in the hailstone chain between 7 and 1. If you kept track of these values in, for example, a **HashMap<Integer,Integer>**, you'd already know the answers for the other values in the chain. Keeping track of previously computed values so that you can use them again without having to recompute them is called ***caching***.

Write a function

```
private int countSteps(int n, HashMap<Integer,Integer> cache) {
```

that determines the number of steps in the hailstone sequence starting at **n**. The second parameter is a **HashMap** containing previously computed values. Your implementation should look up each value it encounters during the process to check whether the answer from that point is already known. If so, it should use that previously computed value not only to compute the current result, but also to add the counts for all the intermediate steps to the **HashMap**. Thus, if you call **countSteps** with 7 as the first parameter, your code should add the counts for 20, 40, 13, 26, 52, 17, 34, 11, 22, and 7 to the **HashMap** before returning the answer.

Problem 7—Essay: Extensions to the assignments (10 points)

In Crowther's original version of Adventure, the solution to one of the puzzles involves a magic word that automatically returns a treasure from wherever it happens to be to its initial location. In this problem, your job is to discuss the changes that would be necessary in the Adventure implementation to implement a similar feature in the game from Assignment #6. This question requires an essay answer, and you should not feel compelled to write any actual code unless you feel that doing so is the best way to convey your ideas. You should, however, indicate by name what classes and methods need to change.

To make this question more concrete, suppose that you have been asked to add a command

> **ZAP** *object*

that has the effect of returning the specified object to its initial room. Like **LOOK**, **QUIT**, **INVENTORY**, and the other action commands, the **ZAP** command can be executed in any room. Unlike the **TAKE** command, however, the object is not ordinarily in that room, but can be anywhere in the cave, including the player's inventory.

Before you get the idea that this question is too easy, note that there is no way in the current design to determine where an object is currently located. Given a room, you can find the objects in that room, and you have presumably added an array list somewhere to keep track of the player's inventory. Adding the object to the list in its starting room is relatively easy, but taking it out of its current location is hard. You need to think about the changes you need to make in the overall design to implement this feature.