Colin Kincaid
CS 106A

Practice Final
August 14, 2018

Practice Final Examination

Final Exam Time: Friday, August 17th, 12:15P.M.–3:15P.M.
Final Exam Location: Dinkelspiel Auditorium

Based on handouts by Nick Troccoli

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. We do not guarantee that the number of questions in this sample exam will match the number on the real exam, nor that every kind of problem shown here will exactly match the kinds of problems shown on the final exam (though the real exam will be generally similar overall).

The midterm exam is on your computer but closed-textbook and closed-notes. A "syntax reference sheet" will be provided during the exam (it is omitted here, but available on the course website). It will cover all material presented up to the midterm date itself. Please see the course website for a complete list of midterm exam details and logistics.

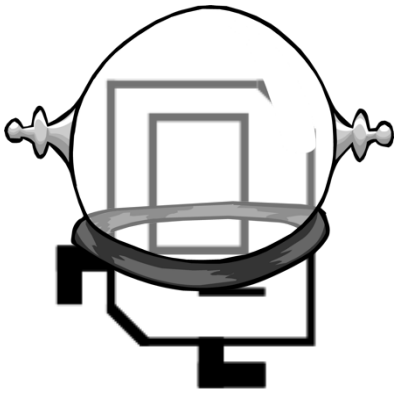
General instructions

Answer each of the questions included in the exam. If a problem asks you to write a method, you should write only that method, not a complete program. Type all of your answers directly on the *answer page provided for that specific problem*, 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 180. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem.

Unless otherwise indicated as part of the instructions for a specific problem, your code will not be graded on style – only on functionality. On the other hand, good style (comments, etc.) may help you to get partial credit if they help us determine what you were trying to do.

You have 3 hours to complete this exam. We recommend looking over all problems before starting. **Remember that you do not have to complete problems in order.**



The Adventures of Karel, Space Explorer

* Karel the Robot is not actually on the final exam. DO NOT write any Karel code on this exam!

TABLE OF CONTENTS

Chapter 1: Designing a spaceship [35 points]

In order for Karel to embark on its journey across the universe, it needs to design and create a spaceship! Help Karel by implementing the following class and method as described below.

Part 1 [25 points]

Write a class called `Spaceship` that models some characteristics of a Spaceship. Specifically, a `Spaceship` keeps track of the amount of food on board (in pounds), a list of names of visited planets (in order of visit), and the name of each crew member on board, as well as how much food (in pounds) each crew member consumes each day. *Note that you should store the crew members and their food consumption in a HashMap; not doing so will result in a substantial deduction.*

You create a `Spaceship` by specifying the amount of food initially on board, like so:

```
Spaceship myShip = new Spaceship(50);   // 50 pounds of food initially
```

A Spaceship should have the following public methods:

```
/** Boards a crew member with the given food intake.  This crew member will
   now consume food during trips. */
   public void board(String crewMemberName, int foodPerDay);

/** Unboards a crew member with the given name from the ship.  This crew
    member is no longer on the ship and no longer consumes food. */
   public void unboard(String crewMemberName);

/** Returns a String of visited planets, in order of visit.  The string
    should be formatted like "[Earth, Mars, Venus]" */
   public String getPlanetsVisited();

/** Attempts to fly to a planet, which takes the given number of days. */
   public boolean flyTo(String planetName, int daysRequired);
```

The most involved public method is `flyTo`. This method should do the following:

- It should first calculate if there is enough food on board to feed all crew members for the number of days required to get to this planet.
- If there **is** enough food, it should update the amount of food onboard to reflect that the ship has traveled to this planet, and it should add this planet to its list of visited planets. The method should also return `true` to indicate the trip was successful.
- If there **is not** enough food, then the method should return `false` to indicate the trip failed. It should not modify either the onboard food or the list of visited planets.

As an example, let's say on `myShip` we have onboard Nolan, who consumes 3 pounds of food daily, and Nick, who consumes 4 pounds of food daily. `myShip.flyTo("Venus", 7)` should return `true` because Nick and Nolan consume 7 pounds of food per day in total, which over 7 days is 7*7=49 pounds of food (and our ship has 50). We should therefore add `"Venus"` to our list of visited planets, and set the food onboard to now be 50-49=1.

On the other hand, `myShip.flyTo("Mars", 10)` should return `false` because we require 7*10=70 pounds of food to get to Mars, but our ship only has 50. We should not add `"Mars"` to our list of visited planets, nor change our onboard food.

Part 2 [10 points]

Now that we have designed a Spaceship, it's time to put it to use! Write a method called **`visitablePlanets`** that returns a list of planet names that we could visit given a certain amount of food. Specifically, the method accepts three parameters: the amount of food our spaceship starts with (`int`), a map of names of crewmembers to the amount of food they each consume daily, and a map from planet names we want to visit to the number of travel days required to visit that planet.

You should first create a new `Spaceship` with the given amount of food. Then, you should use the public methods of `Spaceship` to board all crew members and attempt to travel to the planets in the provided planet map. When iterating over the planet map's keys, you should assume that the keys (planet names) are in the order in which you want to travel to the planets. You should return the list of planets that the `Spaceship` can travel to given the amount of food it starts with.

For example, if the parameters had the values

```
startingFood = 50,
crewMap = {Rishi=2, Guy=2, Aleksander=3},
planetMap = {Venus=2, Mars=3, Saturn=10}
```

then **`visitablePlanets`** should return **"`[Venus, Mars]`"** because we have enough food to get to Venus (7*2 = 14 pounds required) and then Mars (another 7*3 = 21 pounds required) but not Saturn after that (we have only 50-14-21=15 pounds remaining, but need 7*10=70).

Chapter 2: alien communications [25 points]

After embarking on its space journey, Karel suddenly receives some cryptic communication from alien lifeforms. It seems, not having taken CS 106A, these aliens have sent Karel poorly-written Java code and a series of questions about Java! Karel immediately enlists your help to decipher and answer the questions in each of the parts below.

Part 1 [5 points each, 15 points total]

Write the output produced when the following method is passed each of the inputs listed below. It does not matter what order the key/value pairs appear in your answer, so long as you have the right overall set of key/value pairs. You may assume when you iterate over the map, it is iterated over in the same order the key/value pairs are listed below.

```
private void collectionMystery(HashMap<String, String> map,
                               ArrayList<String> list) {

  HashMap<String, String> map2 = new HashMap<>();
  for (String key : map.keySet()){
     boolean b = false;
     for (String str : list){
        if (map.get(key).equals(str)){
           b = true;
        }
     }
     if (b) {
        map2.put(map.get(key), key);
     }
  }
  println(map2);
}
```

a) map = {cow=moo, pig=oink, dog=bark, alien=boop},
   list = [moo, woof, meow, boop]
b) map = {dog=daisy, woof=daisy, karel=robot, space=robot},
   list = [daisy, robot, explorer]
c) map = {cat=meow, sheep=ba, lamb=ba, kitten=meow},
   list = [meow]

Part 2 [5 points each, 10 points total]

Write your answers to the questions below in your answer booklet – DO NOT WRITE HERE!

1) Give an example of where inheritance is useful in programming, and explain why it is useful in this example.
2) Explain why, when we pass primitives as parameters to a method and change them in that method, the changes do not persist outside of that method, but when we pass objects as parameters and change their properties, the changes do persist outside of that method.

Chapter 3: decoding messages [25 points]

The aliens now decide to send Karel messages as *arrays of 0s and 1s* (ints), which Karel has to decode into its original text form. Help Karel by implementing the following methods below. You may assume that the message digits are either 0 or 1.

*Constraints: for this problem, you may not create any **new** data structures (maps, lists, arrays).*

Part 1 [15 points]

The sent messages sometimes get *corrupted* when transmitted through space. Write a method named **`correctMessage`** that takes an **`int`** array of any length as a parameter representing a message, and *flips* its digits as needed to correct it (this method does not return anything).

To correct a message, you must ensure that *there are no sequences of more than three of the same digit in a row*. For instance, `[1, 1, 0]` is a valid message, but `[0, 1, 1, 1, 1]` is corrupted because there are four 1s in a row. You should go through the message from left to right and, whenever you see a sequence of more than three of the same digit in a row, *flip the fourth digit* (e.g. 0 -> 1 or 1 -> 0) to make it so that there are only three in a row.

For example, for the message `[0, 1, 1, 1, 1, 0]`, you should flip the 5th digit from a 1 to a 0, and the corrected message would be `[0, 1, 1, 1, 0, 0]`. Similarly, if you receive the message `[1, 1, 1, 1, 1]`, you should flip only the 4th digit (not the 5th) from a 1 to a 0, and the corrected message would be `[1, 1, 1, 0, 1]`.

You should go from left to right, and if a flipped digit causes *another* sequence of more than three of the same digit, you should correct that too. For instance, for the message `[1, 1, 1, 1, 0, 0, 0, 1]`, working from left to right we should first flip the 4th digit from a 1 to a 0 to get `[1, 1, 1, 0, 0, 0, 0, 1]`. Then, we get to the 7th digit and see that this causes four 0s in a row, so we flip it from a 0 to a 1 to get `[1, 1, 1, 0, 0, 0, 1, 1]`. We get to the end without finding additional sequences of more than three of the same digit, so we are done.

Part 2 [10 points]

Now write a method named **`decodeMessage`** that decodes a corrected message into text. The method accepts three parameters: the **`int`** array of any length representing a corrected message, an **`int digitsPerChar`** representing the number of digits that correspond to a single character, and a map from `String` to `Character` mapping digit sequences to characters.

To decode a message, break the message into **`digitsPerChar`**–size chunks and convert each chunk to a character. To do this, look up each chunk (in String form; e.g. 0, 1, 0, 1 is **`"0101"`**) in the given map which maps digit sequences to single characters. You may assume that every sequence of digits is in the map, and that the number of digits is divisible by **`digitsPerChar`**. Your method should return the fully-decoded message as a string.

For example, for the message `[0, 1, 1, 1]`, `digitsPerChar=2`, and the map `{01=a, 11=b}`, the first chunk of 2, the string **`"01"`**, corresponds to **`'a'`** in the map. The second chunk of 2, the string **`"11"`**, corresponds to **`'b'`** in the map. Our method would therefore return the string **`"ab"`**.

Chapter 4: analyzing alien dna [25 points]

After decoding the alien messages, Karel managed to successfully navigate to the aliens' home planet. Of course, being an avid scientist, Karel wasted no time taking an alien DNA sample to analyze. Alien DNA, which is made up of "bases" (the letters B, O, R, K), can easily be represented as a String. For instance, the string `"KRBOORBOKRB"` is a valid string of alien DNA. A **k-mer** is a k-length substring of bases in that DNA. For instance, here are all the 3-mers in this sequence:

```
"KRBOORBOKRB"
-------------
 KRB
  RBO
   BOO
    OOR
     ORB
      RBO
       BOK
        OKR
         KRB
```

Conversely, in this example, `"BBB"` is *not* a 3-mer because it is not a substring of this DNA sequence. An example of a 4-mer in this string is `"BOKR"`.

Write a method named **`mostFrequentKmer`** that returns the k-mer found most often in a given strand of alien DNA. The method accepts two parameters: the DNA string, and the value of **k** (an `int`). As an example, if we are given the DNA string `"BOBOK"` and `k=2`, then the 2-mer found most often is the string `"BO"` because it appears twice, while other 2-mers (`"OB"`, `"OK"`) appear only once. If there is a tie for the k-mer found most often, you may return any one of the most frequent k-mers.

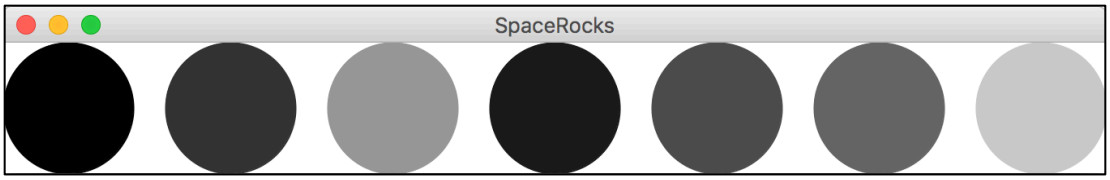
Constraints: you may create no more than 1 additional data structure (map, list, array).

Chapter 5: space rocks! [25 points]

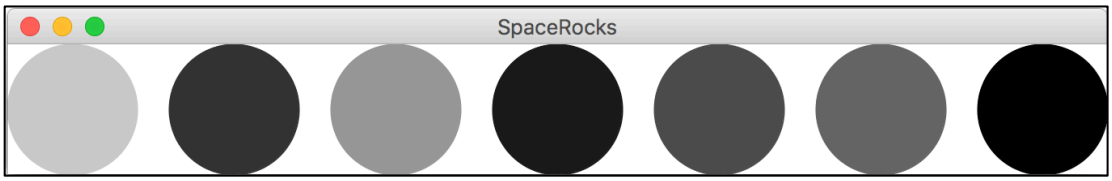
While on this alien planet, Karel figures it's also good to collect samples of space rocks. Help Karel by writing a complete graphical program named **`SpaceRocks`** to help organize and discard space rock samples. When writing this program, you should make use of the following provided constants (you do not need to define these in your answer):

```
private static final int NUM_ROCKS = 8;      // number of space rocks to display
private static final int ROCK_DIAMETER = 80;    // width/height of a space rock
private static final int ROCK_SPACING = 20; // pixels between each space rock
```

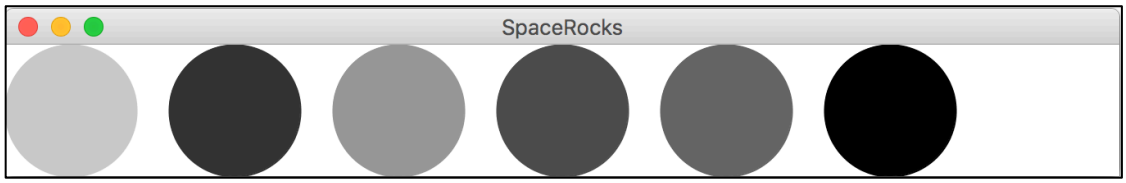
When you start the program, the canvas should display the space rocks as `GOvals` with **random colors** (note that these colors are displayed in grayscale here). The rocks should be positioned starting at the left of the canvas - (0, 0). You do not need to worry about setting the canvas size.



If the user clicks on one space rock, followed by another space rock, those two space rocks should *swap positions* in the row of rocks. For instance, if the user clicks on the first (darkest) rock, followed by the last (lightest) rock, they would swap and the display would look like so:



If the user clicks on one space rock, and *again* on the same space rock, that rock should be **removed**, and **all rocks to its right should shift to the left** so that the rocks are still correctly padded. For instance, if the user clicks twice on the 4th rock from the left (dark), you should remove that rock and shift to the left the three rocks to its right to fill the empty space, like so:



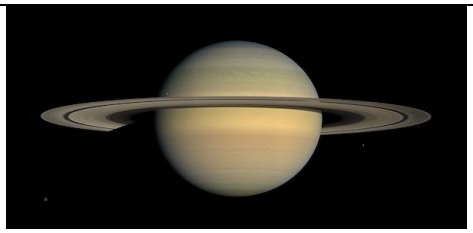
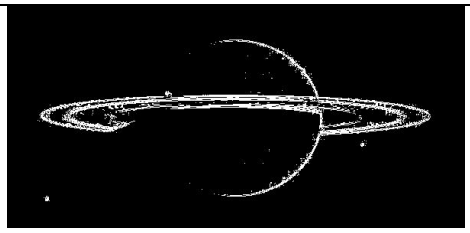
You may assume that the user will only click on the rocks (and not any blank spaces).

Hint: to make the program simpler, we recommend writing a method like `repositionRocks` *that updates the position of all rocks still present.*

Constraints: you may create no more than 1 additional data structure (map, list, array). You also ***may not change*** *the color of any rocks after they are initialized to a random color.*

Chapter 6: images of outer space [45 points]

Along its journey through the universe, Karel has captured pictures of various artifacts. One common image algorithm to extract information about these images is called *edge detection*. Edge detection makes an image of the same size as an original, where pixels along **edges** within the image are colored white, and all others are colored black. For instance, running the edge detection algorithm on the image on the left would produce the image on the right:

Source image	*Edge Detection output image*
	

Part 1 [30 points]

Write a method named **`detectEdges`** that takes as a parameter a source `GImage` (sized at least 1x1 pixels) and returns a new `GImage` of the same size that displays the edges within that image.

The general idea behind the edge detection algorithm is that we consider a pixel part of an "edge" if it differs significantly from its neighbors. The way we calculate this is similar to how you implemented **blur** for the ImageShop assignment. For a given pixel *(r, c)* located at row *r* and column *c* in the source image, we calculate the average luminosity (rounded down to the nearest integer) of the pixels at locations *(r-1, c-1)* through *(r+1, c+1), not including (r, c)*. For example, in the diagram below, for the pixel (row 1, column 2), you would calculate the average luminosity of the eight pixels (0, 1), (0, 2), (0, 3), (1, 1), (1, 3), (2, 1), (2, 2), and (2, 3).

	0	*1*	*2*	*3*	*4*
0	(14, 97, 63)	(84, 22, 99)	(74, 38, 69)	(16, 17, 18)	(85, 75, 75)
1	(21, 18, 45)	(66, 53, 88)	**(32, 67, 12)**	(95, 65, 35)	(6, 0, 2)
2	(37, 29, 61)	(28, 49, 31)	(47, 21, 94)	(31, 41, 51)	(246, 84, 13)
3	(82, 33, 90)	(42, 43, 44)	(15, 80, 50)	(60, 40, 12)	(188, 45, 1)

Note that if a pixel is near the sides of the image, it may have fewer neighbors. Once you have the average neighbor luminosity, you subtract it from that pixel's own luminosity, and take the absolute value to ensure the difference is positive. This code would look something like:

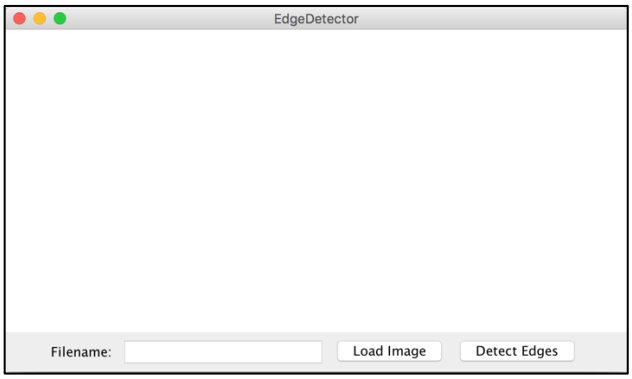
```
int luminosityDifference = Math.abs(ownLuminosity - neighborAvgLuminosity);
```

If this difference is **greater than the provided constant `THRESHOLD`** (you do not have to define this constant), then this pixel should be white (r=255 ,g=255, b=255) in the new image. Otherwise, it should be black (r=0 ,g=0, b=0). You may assume the method `computeLuminosity(int r, int g, int b)`, which returns the luminosity for the given RGB values, has been written for you (you do not have to write it).

*Constraints: You may not create any **new** data structures except for at most one 2D array. (Asking for the pixel array of an existing image does not count; writing '**new**' is what counts.)*

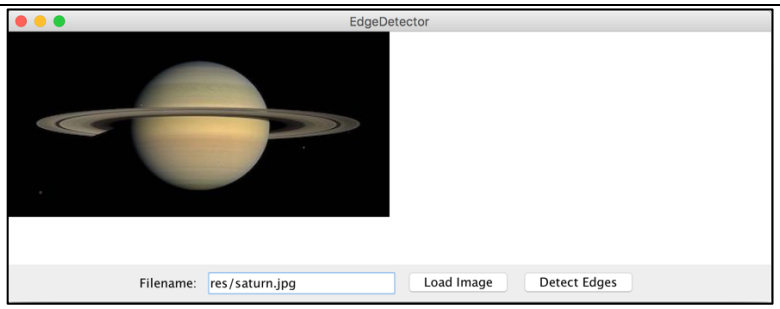
Part 2 [15 points]

Write a complete graphical program named **EdgeDetector** that implements a graphical user interface for loading and displaying an image, and running edge detection on that image. When you start the program, the user interface has a blank central canvas and a bottom region with the following graphical components: a label displaying the text "Filename: ", a text field of width **16** for the user to input a filename, a "Load Image" button, and a "Detect Edges" button.

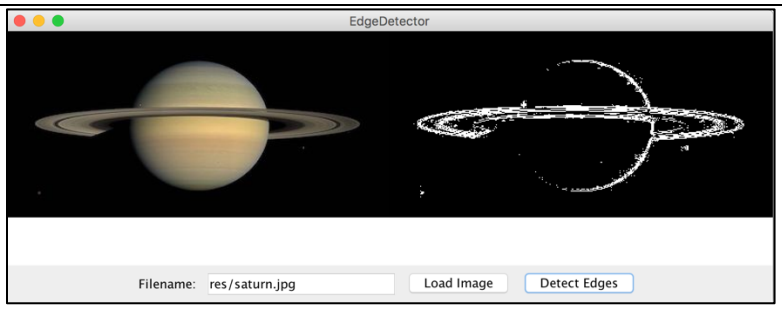


When the user enters a filename (e.g. **"`res/saturn.jpg`"**) and clicks the "Load Image" button or presses ENTER in the text field, that image should be added to the canvas with its upper-left corner at (0, 0). When the user clicks the "Detect Edges" button, you should call `detectEdges` on the loaded-in image and display the output immediately to the right of the original image.

After typing "res/saturn.jpg" and clicking "Load Image" (or hitting ENTER)



After clicking "Detect Edges"



You may assume that the entered filename is valid, and do not have to worry about edge cases such as the user clicking "Load Image" multiple times, "Detect Image" multiple times, or the user clicking "Detect Image" without first loading an image. You also do not need to worry about setting the canvas size.