LAB 4

Create a project named **`lab4`** to store your work here. Framework code and this lab are available on moodle. This is the entirety of lab4—there won't be a *part B*—so you will have two weeks to complete this document. The problem write-ups are loooong, but the problems shouldn't be too hard.

Lab 4 is due on Monday, April 20th, before the start of lab on that day.

1. Processing images.

In lecture, we worked for a bit with the class **`RBGImage`**. Here, you are going to write more methods that transform images. The classes **`RGBImage`** and **`RGBImageViewer`** are available on moodle, and need to be in the same package. You'll also need a folder named **`images`** that contains jpeg files (other formats may work as well) directly under the project folder (not in **`src`**) in order to use the simple constructor:

```
RBGImage img = new RGBImage("seagull.jpg")  // opens seagull.jpg in the images folder
```

The zipped project in Moodle has the correct structure and some test images.

If you missed lecture, we noted that images are, essentially, a 2d array of pixels (dots of a single color). The color of a particular pixel can be broken down into three channels—red, green, and blue—each of which takes a value of 0 (off) to 255 (as bright as possible). If all values are 0, the color will be black; if all are 255, the color will be white. If red is 255 and the other channels are 0, the pixel will be fully red.

The class **`RGBImage`**, rather than being backed by a 2d array of pixels, contains three separate 2D arrays, one for each color channel. You can see them defined starting on line 20:

```
private int red[][];
private int green[][];
private int blue[][];
```

These arrays hold values between 0 and 255 inclusive. The outer array holds rows, the inner columns. So, **`red[0][100]`** will hold a **`int`** between 0 and 255 for the pixel 100 positions down on the left side of the image.

We looked at the **`flipVertical()`** method, and wrote **`makeGreyscale()`** (by making the R,G, and B values equal, the pixel will be grey) and **`threshHolding()`** (this turns each pixel either fully black or fully white, depending on how bright it started out as). Try them out by modifying and running the **`TestRGBImage`** class.

Finish the **`mirrorHorizontal`**, **`contrastStretch`**, and **`addBorder`** methods inside **`RBGImage`**. Instructions are in comments above method stubs.

Make sure to add a call to **`refresh`** at the end of your methods or you won't see any effect!

Modify and use the class **`TestRGBImage`** to test your methods on various images.

2. Spatial Filtering

Spatial filtering provides a way of doing a variety of image processing tasks, such as smoothing and sharpening, by applying different spatial filters through one common method.

A spatial filter is a small, square, two-dimensional array of **`double`**s. The width and height of this matrix is odd (e.g., 3 cells by 3 cells, 5 cells by 5 cells, etc.), such that there is a specific cell at the center of the filter. Generally, the numbers in the cells of a filter should sum to `1`, although `0` is not uncommon. To this end, a filter might contain some negative values and some positive values, or it might contain all positive numbers between 0 and 1.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

To process a spatial filter for a particular pixel, line up the filter so that the center cell corresponds to that pixel, and the other cells correspond to nearby neighbor pixels. For each filter cell, multiply a color value of the pixel with the corresponding filter cell, sum all of these up, and set that result as the pixel's value for that color. This process happens separately for each color channel.

Two issues will arise:

- The result of the spatial filter may be less than `0` or greater than `255`. Make sure anything out of range gets set to the appropriate maximum or minimum value.
- When at the edges of an image, there won't be pixels to correspond to each cell in the filter. Two options: don't apply the filter to a pixel unless all cells correspond to a neighbor pixel, or apply the filter to only the relevant cells.

Write a method **`spatialFilter`** in **`RGBImage`** that takes in a 2D array of doubles and applies that filter to the image. Use helper methods: don't write code that has 4 nested **`for`** loops!

Test images with the following 3x3 filters (or, 5x5 versions you create yourself) in a test class:

Smoothing

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Sharpening

0	-1	0
-1	5	-1
0	-1	0

Vertical Edge Detection

-1	0	1
-1	0	1
-1	0	1

Horizontal Edge Detection

-1	-1	-1
0	0	0
1	1	1

3. Reversing strings

This question will take a little while to set up!

Consider a recursive solution to a method **`String reverse (String s)`** —that is, a method that calls itself on a *smaller* problem and doesn't use **`StringBuilder`** or a loop to perform the reversal.

Here are some naïve ways of writing reverse for inputs of a specific length:

```
// returns the reverse of a one-length string.  Easy!
public String reverse1 (String s) {
    return s;
}

//returns the reverse of a two-length string.
public String reverse2 (String s) {
    return (s.substring(1,2) + s.substring(0,1));
}

//returns the reverse of a three-length string.
public String reverse3 (String s) {
    return (s.substring(2,3) + s.substring(1,2) + s.substring(0,1));
}
```

These operate correctly, but, as you can see in the solution below, can be tedious to write:

```
public String reverse16 (String s) {
    // This is the *bad* way to write reverse!
    return s.substring(15,16) +
            s.substring(14,15) +
            s.substring(13,14) +
            s.substring(12,13) +
            s.substring(11,12) +
            s.substring(10,11) +
            s.substring(9,10) +
            s.substring(8,9) +
            s.substring(7,8) +
            s.substring(6,7) +
            s.substring(5,6) +
            s.substring(4,5) +
            s.substring(3,4) +
            s.substring(2,3) +
            s.substring(1,2) +
            s.substring(0,1);
}
```

Ug. Let's do a better job on **`reverse17`**. Use the class **`ReverseMethods`**, which contains all the code shown so far, to do your work.

1. Write **`reverse17`** such that it uses **`reverse16`** to compose the correct answer. That is, given a working version of **`reverse16`**, how can **`reverse17`** use it to make its solution easy? The body of **`reverse17`** should be a single line long!

You'll have two helper methods at your disposal: **`allButFirst(String s)`**, which returns a string missing the first character of the input **`s`** (and which will error if given an empty string!), and **`allButLast(String s)`**, which returns a string of all the characters in **`s`** but the last.

There are two 'correct' ways to write **`reverse17`**; each way will use one of the helper methods above.

2. Now, you are ready to write **`reverse`**, which combines all the specific length reverse methods into a whole. Reverse takes a string of any length and returns its reverse.

   Remember, recursive methods make a call to a clone of themselves that solves a slightly easier problem… similar to the way that your solution to **`reverse17`** (hopefully) used **`reverse16`**.

   Again, there are two 'correct' ways to write this; each way will use one of the two helper methods (**`allButFirst`** and **`allButLast`**).

Remember, no loops! Your resulting code should be *very* simple.

4. Counting digits

Consider a method **`digitCount`** which takes an **`int`** value and returns an **`int`** that is the number of digits in the input.

```
digitCount(5)    // returns 1
digitCount(10)   // returns 2
digitCount(82)   // returns 2
digitCount(9912) // returns 4
```

The following recursive implementation has a bug:

```
public static int digitCount (int value) {
    if (value == 0) {
        return 1;
    } else {
        return 1 + digitCount(value/10);
    }
}
```

Fix the bug, and write a **`TestDigitCount`** class that makes sure **`digitCount`** works for a range of inputs. Fixing the bug should take a change to only one line!