CS106A
Winter 2012-2013

Handout 24
February 20, 2013

Assignment 5: Array Algorithms[*]

Introduction

Arrays are a fundamental and versatile tool for representing, manipulating, and transforming data. In this assignment, you'll see how arrays can be applied to generate and manipulate music and images.

Like the first two assignments for the course, this assignment consists of three smaller programs. Part One of this assignment (**Steganography**) explores how arrays can be used to represent images and how simple transformations on images can be used to hide secret messages. Part Two of this assignment (**Tone Matrix**) lets you use arrays to play sounds and construct an awesome musical instrument. Part Three of this assignment (**Histogram Equalization**) shows how you can use arrays in a variety of contexts to manipulate photographs and recover meaningful data from overexposed or underexposed pictures.

By the time you've completed this assignment, you will have a much deeper understanding of how to use arrays to model and solve problems. You'll also be able to share secret messages with your friends, compose music, and fix all your old family photos.

Due Monday, March 4 at 3:15PM

* This is a *brand-new* assignment, and we're really excited to be rolling it out! If you have any suggestions on how to improve this assignment in future quarters, please don't hesitate to contact your section leader, Gil, or Keith with feedback!

Part One: Steganography[*]

Suppose that you want to send a secret message to someone else without other people being able to read it. One way that you could do so would be to *encrypt* the message in a way where only the recipient could *decrypt* it. To everyone else, the message would look like total gibberish. This approach to transmitting secret messages is called *cryptography*.

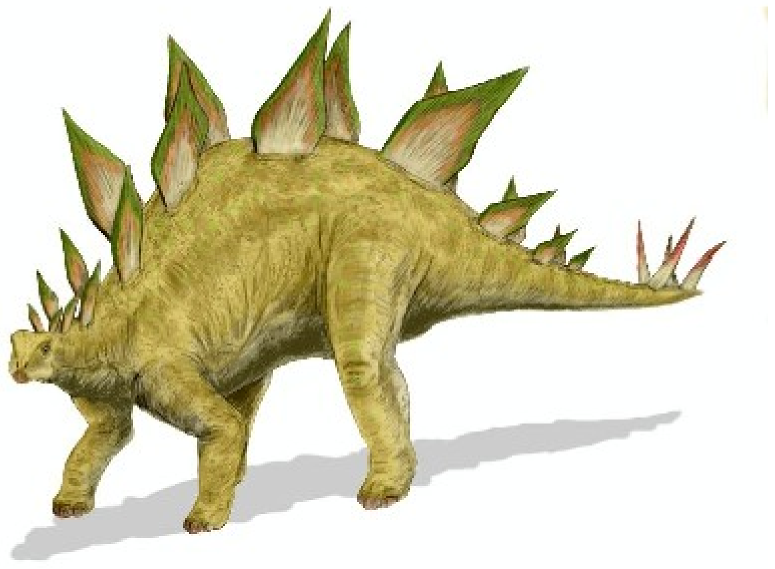
An entirely different way to deliver a secret message would be to send the message in a format that no one would suspect carries a secret message. For example, suppose that I were to send you the following note:

A**n**drew L**u**ck is quarter**b**ack for the Ind**ian**apol**i**s Colts. He plays foot**b**all **ex**cellently!

If you just read the message by itself, it seems to be a comment on Stanford's (awesome) former quarterback. However, if you look closer, you can see that some of the letters have been bolded. If you just read those letters, you get back the message **nubian ibex**, which was the secret message I was trying to send to you. Notice that the actual message to transmit isn't encrypted – anyone who knows where to look for the message can still read it – but it has been hidden so that an observer who didn't know to look for it wouldn't find it.

The art and science of sending secret messages by concealing the existence of the secret message is called ***steganography***. In this part of the assignment, you'll build a system for hiding messages in images so that you can send secret messages to your friends.

The specific type of steganography you'll be doing in this assignment is called *image steganography*. In image steganography, you will hide a black-and-white image (the secret message) inside of a full-color image in a way that no human eyes could ever detect. As an example of this, consider the following picture of a stegosaurus:



It looks like a totally normal picture, but surprisingly there is a second image embedded within it: a beautifully, artistically rendered and scientifically accurate picture of a Tyrannosaurus Rex:

[*] The steganography assignment was inspired by Julie Zelenski's steganography assignment given in a previous quarter's CS107. It also draws inspiration from Brent Heeringa and Thomas P. Murtagh's "Stego my Ego" assignment from Nifty Assignments.

How is this image steganographically hidden within the steganosaurus? The answer has to do with the way that computers represent images. Recall from lecture that each pixel in an image is represented as the intensity of its red, green, and blue components. Each of these components ranges from 0 to 255, with 0 meaning “none of this color” and 255 meaning “the maximum possible intensity of this color.”

Although every different combination of red, green, and blue produces a different color, the human eye is not sensitive enough to notice minor changes between closely-related RGB triplets. For example, the color magenta would be represented by the RGB triplet (255, 0, 255), with the red and blue components at maximum and the green component at zero. The related RGB triplet (254, 0, 255) is also an intensely magenta color. While (254, 0, 255) is not exactly the same color as (255, 0, 255), there is almost no perceptible difference between the two. If you were to look at them one after the other you'd be hard-pressed to find the difference between them. Our steganography system will use the fact that the human eye cannot tell the difference between two pixels of subtly different color, though the computer can distinguish them perfectly.

Our goal will be to hide secret black-and-white images within full-color images. To do this, we'll start off with two images: the secret black-and-white image, and the full-color master image. For simplicity, we'll assume that these images are always the same size as one another. We will then process the black-and-white image one pixel at a time, making minor adjustments to the corresponding pixels in the color image. The changes we will make are so small that no human will be able to detect that the color image has changed.

The specific approach we will use is the following. Beginning with the secret black-and-white image and the full-color master image, we will process pairs of corresponding pixels from the two images one at a time. We will then transform the colors in the master image as follows:

- If the pixel from the secret message is **black**, make the red channel of the color pixel **odd**.
- If the pixel from the secret message is **white**, make the red channel of the color pixel **even**.

For example, if the secret message pixel is black and the color image pixel has color (255, 0, 127), we would leave the color image pixel unchanged because its red channel (255) is already odd. However, if the pixel from the secret message was black and the color pixel had RGB value (100, 100, 100), we would change the pixel from the color image to have RGB value (101, 100, 100), which is a value close to the original pixel value but which has an odd number for its red channel. Similarly, if the pixel from the secret message was white and the color pixel had RGB value (0, 0, 0), we would leave the original pixel untouched because its red channel (0) is already even. However, if the secret pixel was white and color pixel had value (255, 255, 255), we would change the color pixel to (254, 255, 255), which is close to the original color but now has an even number in its red channel.

As an example, suppose that we have a black-and-white image and a color image, which are represented below (the red channel in the color image has been highlighted):

□	■	■	■	□		**255**, 0, 100	**137**, 42, 10	**106**, 103, 4	**27**, 18, 28	**31**, 41, 59
■	□	□	□	■		**86**, 75, 30	**123**, 58, 13	**0**, 255, 0	**161**, 08, 0	**45**, 90, 45
■	□	■	□	■		**66**, 99, 10	**11**, 5, 9	**20**, 8, 0	**100**, 50, 25	**1**, 10, 100
■	□	□	□	■		**0**, 0, 0	**255**, 0, 0	**123**, 57, 11	**0**, 0, 255	**70**, 70, 70
□	■	■	■	□		**83**, 69, 69	**89**, 79, 85	**154**, 161, 1	**140**, 144, 2	**124**, 145, 3
Black-and-White Image						Original Color Image				

To embed the black-and-white image inside the color image, we would adjust the red channels of the color image as follows:

□	■	■	■	□		**254**, 0, 100	**137**, 42, 10	**107**, 103, 4	**27**, 18, 28	**30**, 41, 59
■	□	□	□	■		**87**, 75, 30	**122**, 58, 13	**0**, 255, 0	**160**, 08, 0	**45**, 90, 45
■	□	■	□	■		**67**, 99, 10	**10**, 5, 9	**21**, 8, 0	**100**, 50, 25	**1**, 10, 100
■	□	□	□	■		**1**, 0, 0	**254**, 0, 0	**122**, 57, 11	**0**, 0, 255	**71**, 70, 70
□	■	■	■	□		**82**, 69, 69	**89**, 79, 85	**155**, 161, 1	**141**, 144, 2	**124**, 145, 3
Black-and-White Image						Modified Color Image				

Take a few minutes to make sure you understand how the colors from the original image were transformed to produce this new image.

Once we have encoded our secret message within the original image, we can easily recover it by looking at all the pixels of the color image one at a time. For each pixel in the color image, if its red channel is an odd number, the corresponding pixel in the black-and-white image must be black. Otherwise, the corresponding pixel must be white.

The Assignment

Your job in Part One of this assignment is to implement the methods from the **`SteganographyLogic`** class. These methods are responsible for hiding and finding hidden messages in images.

In our framework, the color image will be represented as a **`GImage`**. As mentioned in lecture, you can get a 2D array of the pixels of a **`GImage`** by calling the **`getPixelArray()`** method. From there, you can extract the red, green, and blue components of any individual pixel by passing that pixel into the methods **`GImage.getRed`**, **`GImage.getGreen`**, and **`GImage.getBlue`**. Given red, green, and blue channels of a pixel, you can create a pixel of that color using the **`GImage.createRGBPixel`** method.

The black-and-white image will be represented as a **`boolean[][]`** two-dimensional array. White pixels are represented as **`false`**, while black pixels are represented as **`true`**. This means that

If the secret pixel is ***black***, it is represented as ***`true`***, and you should make the red channel ***odd***.

If the secret pixel is ***white***, it is represented as ***`false`***, and you should make the red channel ***even***.

The **`SteganographyLogic`** class is reprinted below:

```java
public class SteganographyLogic {
      /**
       * Given a GImage containing a hidden message, finds the hidden message
       * contained within it and returns a boolean array containing that message.
       * <p>
       * A message has been hidden in the input image as follows.  For each pixel
       * in the image, if that pixel has a red component that is an even number,
       * the message value at that pixel is false.  If the red component is an odd
       * number, the message value at that pixel is true.
       *
       * @param source The image containing the hidden message.
       * @return The hidden message, expressed as a boolean array.
       */
      public static boolean[][] findMessage(GImage source) {
            /* TODO: Implement this! */
      }

      /**
       * Hides the given message inside the specified image.
       * <p>
       * The image will be given to you as a GImage of some size, and the message
       * will be specified as a boolean array of pixels, where each white pixel is
       * denoted false and each black pixel is denoted true.
       * <p>
       * The message should be hidden in the image by adjusting the red channel of
       * all the pixels in the original image.  For each pixel in the original
       * image, you should make the red channel an even number if the message
       * color is white at that position, and odd otherwise.
       * <p>
       * You can assume that the dimensions of the message and the image are the
       * same.
       * <p>
       *
       * @param message The message to hide.
       * @param source The source image.
       * @return A GImage whose pixels have the message hidden within it.
       */
      public static GImage hideMessage(boolean[][] message, GImage source) {
            /* TODO: Implement this! */
      }
}
```

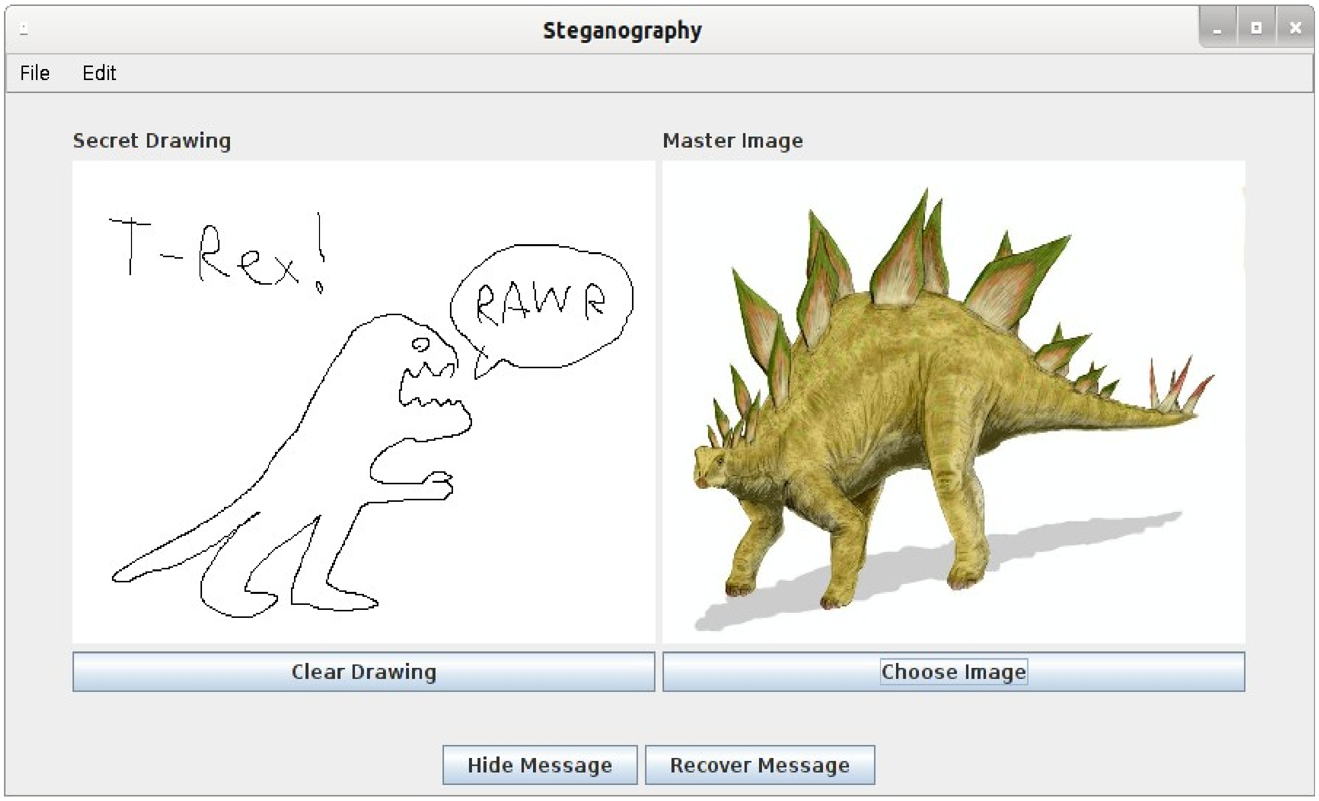
This class contains two methods that you will need to implement. The first, **`findMessage`**, takes as input a **`GImage`** containing a secret message that has been hidden using the algorithm described on the previous page. Your task is to recover the hidden message from that image by returning a two-dimensional array of **`boolean`**s corresponding to the white and black pixels of the hidden message. The second, **`hideMessage`**, takes in a 2D array of **`boolean`**s representing the secret message and a **`GImage`** in which the secret message should be hidden, then uses the algorithm described earlier to hide that message in a **`GImage`**.

We have provided you a **`Steganography`** program that uses your implementation of the methods described above to hide and recover secret messages. When you first run the program, you will see two areas, one labeled "Secret Drawing" and the other labeled "Master Image." You can draw a picture in the region labeled "Secret Drawing," and can use the "Choose Image" button to pick an image into which you will embed that secret drawing.

If you load an image that contains an hidden message and click the "Recover Message" button, the program will call your **`findMessage`** function and show the secret message in the panel marked "Secret Drawing." If the original image didn't contain a secret message, then you're likely to get back garbage patterns, since your **`findMessage`** function will still read back the red channels and try to reconstruct the image.

If you load an image and click the "Hide Message" button, the program will call your **`hideMessage`** function to hide your secret drawing inside the master image. It will then let you save the resulting image to a file so that you can share it with your friends or recover the secret message later. **Be careful when saving images, since the program will let you overwrite existing files**.

Here is a screenshot of this program in action:



Advice, Tips, and Tricks

For this portion of the assignment, we strongly suggest starting off by implementing the **`findMessage`** function and testing it out on the sample images that we've provided for you. That way, you can confirm that your logic for decoding messages works properly before you test it out in conjunction with your **`hideMessage`** function.

When implementing **`hideMessage`**, make sure that you never try increasing the value of the red channel above 255 or decreasing it below 0. If you do, the values will "wrap around" and start producing images with unusual bright or dark spots in them. If you plan out in advance how you will change the red channel values to make them even or odd, you can find a way to do so that never will overflow or underflow the channel values.

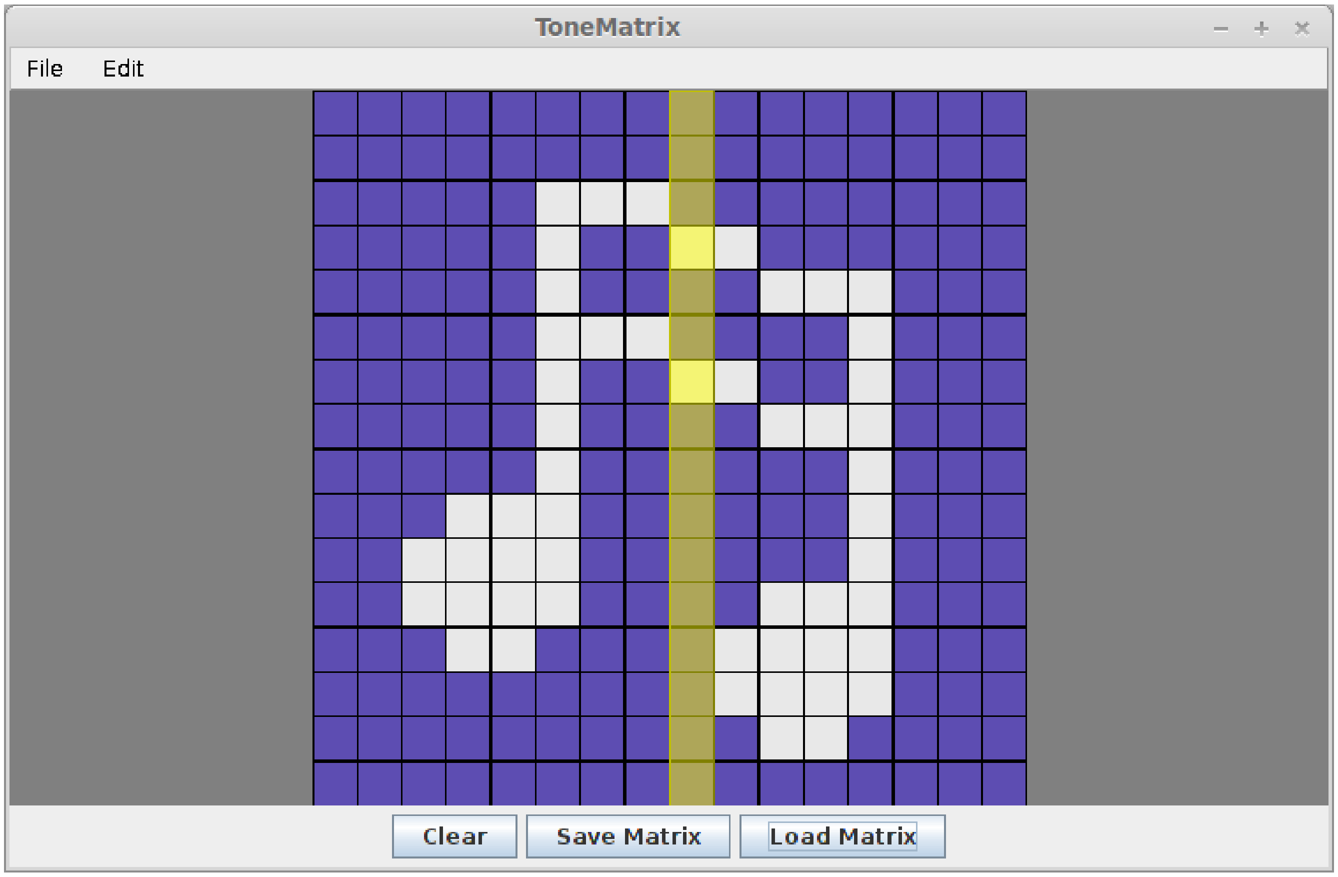
Part Two: Tone Matrix[*]

In this part of the assignment, you'll build a really nifty piece of software that combines music and visuals – the Tone Matrix!

The best way to build up an understanding of the Tone Matrix is to try out the sample version of the program, which is available on the course website.

The Tone Matrix is a 16 × 16 grid of lights, all of which are initially turned off. Each of the lights represents a musical note at a specific point in time. Lights further to the left are played before the lights further to the right. The height of a light determines which note is played: lights toward the bottom of the Tone Matrix have a lower pitch than lights toward the top of the Tone Matrix. All lights on the same row play the same note and correspond to playing that note at different points in time. If multiple lights are turned on in the same column, they will be played simultaneously.

You can see this here:



When you start up the Tone Matrix application, you will see a blank grid with a vertical line sweeping from the left to the right. As you click on the lights to turn them on, the Tone Matrix will start to play music whenever the sweeping line passes over those lights.

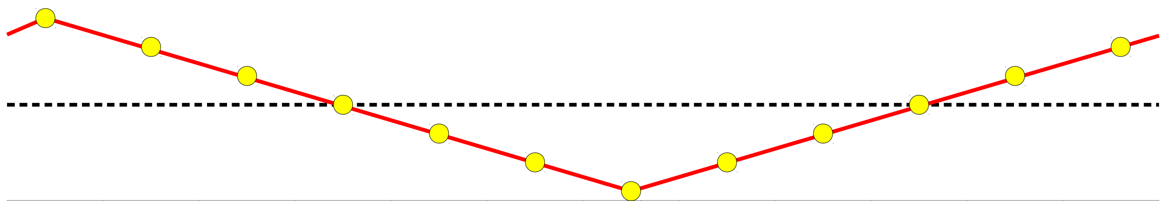
Your job in this assignment is to generate the sound that will be played by the Tone Matrix. We'll handle all of the logic necessary to display the Tone Matrix, draw the sweeping line, and interact with the user. You will be responsible for converting the lights in the Tone Matrix into sound samples that will be sent to the speakers.

[*] The Tone Matrix assignment was inspired by André Michelle's most amazing ToneMatrix program, which is available online at http://tonematrix.audiotool.com/. The starter file uses a modified version of Kevin Wayne and Robert Sedgewick's **`StdAudio.java`** file, developed at Princeton University, to generate sound.

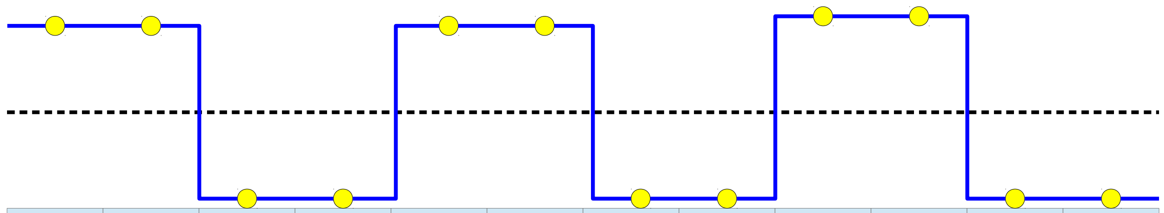
Recall from lecture that computers can represent sound as an array of the intensities of the sound over time. To play sound, the computer can either play an existing sound (by loading the sound from a file on disk), or it can generate a new sound from scratch (by generating the sound wave algorithmically).

The Tone Matrix plays music by using a combination of these two approaches. Specifically, the Tone Matrix generates music by taking 16 existing sounds (one for each note that can be played) and constructing new sounds by combining them together based on which lights are lit up. For example, if a column in the Tone Matrix had lights two, four, six, and eight lit up, then the Tone Matrix would generate a sound sample formed by combining together the second, fourth, sixth, and eighth existing samples, then playing the newly-generated sound sample.

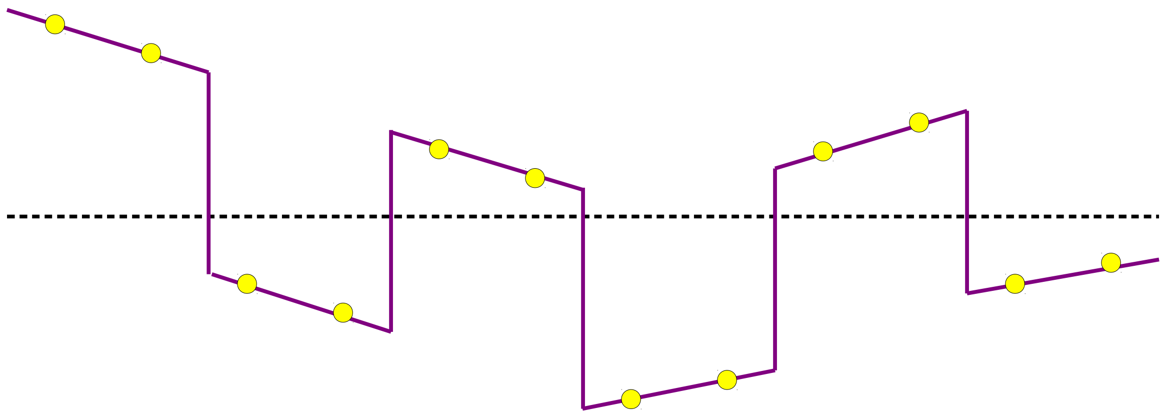
How exactly would we combine together multiple sound samples? One interesting observation is that if you take two different sound samples (represented as arrays of **`double`**s between -1 and +1) and then add up the values in those arrays pairwise, you end up with a new sound sample corresponding to both sounds playing at once. Below is an example of this – the top two waves are the input waves, and the bottom wave is the wave produced by playing both waves at the same time:



1.00	0.67	0.33	0	-0.33	-0.67	-1.00	-0.67	-0.33	0	0.33	0.67



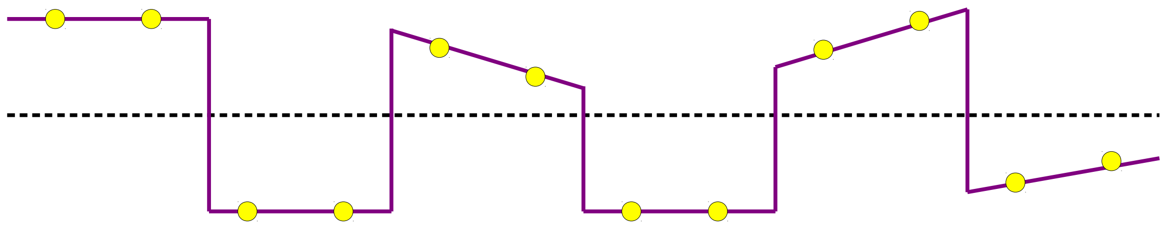
1.00	1.00	-1.00	-1.00	1.00	1.00	-1.00	-1.00	1.00	1.00	-1.00	-1.00



2.00	1.67	-0.67	-1.0	0.67	0.33	-2.0	-1.67	0.67	1.00	-0.67	-0.33

There is one minor problem with this approach. The speakers have a maximum intensity that they can physically produce. Our sound libraries represent this displacement using the values ±1. If you try to play a sound sample containing values outside the range [-1, +1], then the speakers will cap any values above +1 at +1 and any values below -1 at -1. As a result, the sound can come out distorted.

For example, if you tried to directly play the above sound using our sound libraries, it would be the same as playing this sound:



1.00	1.00	-0.67	-1.0	0.67	0.33	-1.00	-1.00	0.67	1.00	-0.67	-0.33

This is completely different from the sound that is intended, and will sound distorted.

To address this, we can *normalize* the sound wave by "squashing" it to fit inside of the range [-1, +1]. To accomplish this, we can find the maximum intensity of the sound at any point (where the *intensity* of the sound at a single point in time is the absolute value of the sample at that point), then dividing all of the sample values in the sound by this maximum value. This forces all the samples in the sound wave to be within the range [-1, +1]. If no sound is to be played, then you can skip the normalization step, since the sound you'll be generating have value 0 everywhere.

The Assignment

Your main task is to implement the following method in the **`ToneMatrixLogic`** class:

```
public double[] matrixToMusic(boolean[][] toneMatrix, int column, double[][] samples)
```

This method takes in three parameters:

- **`boolean[][] toneMatrix`**. This is a two-dimensional array representing the lights in the matrix. **`true`** values correspond to lights that are on, while **`false`** values correspond to lights that are off. The Tone Matrix is stored such that you would look up the entry at position (row, col) by reading position **`toneMatrix[row][col]`**.
- **`int column`**. This integer holds the index of the column that is currently being swept across in the Tone Matrix. Your goal for this function will be to generate a sound sample for this particular column, and you can ignore all the other columns in the matrix.
- **`double[][] samples`**. This two-dimensional array is actually an "array of arrays." Each entry of the **`samples`** array is itself an array of doubles (a **`double[]`**) that holds the sound sample corresponding to a particular row in the Tone Matrix. For example, **`samples[0]`** is the sound sample that would be played by lights in row 0 of the Tone Matrix, **`samples[1]`** is the sound sample that would be played by lights in row 1 of the Tone Matrix, etc. (recall that all lights in the same row as one another all play the same sound). All of these samples have length equal to the constant **`ToneMatrixConstants.sampleSize()`**.

Your method should use the values of **`toneMatrix`** and **`column`** to determine which sounds to play, then construct a sound sample by adding together the appropriate sounds from **`samples`** and normalizing them. The code that we have provided you will automatically call the **`matrixToMusic`** function at the appropriate rate and play the sounds that you generate, so you don't need to actually play the sound clip you create.

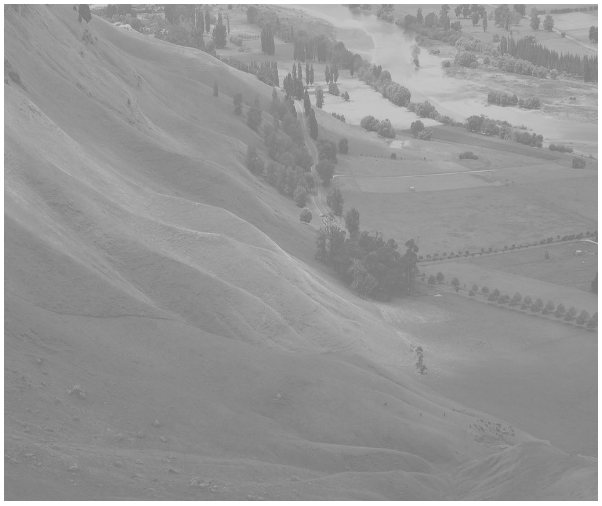
Advice, Tips, and Tricks

We strongly suggest that you start off by only testing your Tone Matrix when there is at most one note playing per column. That way, you don't need to worry about normalizing the sound waves. You can test your Tone Matrix on the file **`diagonal.matrix`**, which will play every note in the matrix one after the other. For now, don't worry about getting multiple notes playing at the same time.

Once you can get individual notes working correctly, you can move on to getting multiple notes playing at the same time. To do this, you will need to combine together multiple sounds, then normalize the result. If you are successfully combining together multiple sounds but have not gotten normalization implemented, then you will probably hear some distorted notes or strange noises when multiple notes play at the same time. As soon as you have normalization working, the sound should come out beautifully, and you can start to play around with your Tone Matrix!

Watch out for the case where no sounds are being played. In this case, you should not try to normalize your sound clip, since this would cause you to divide each entry in the sound clip (all of which will be zero) by the maximum intensity (also zero).

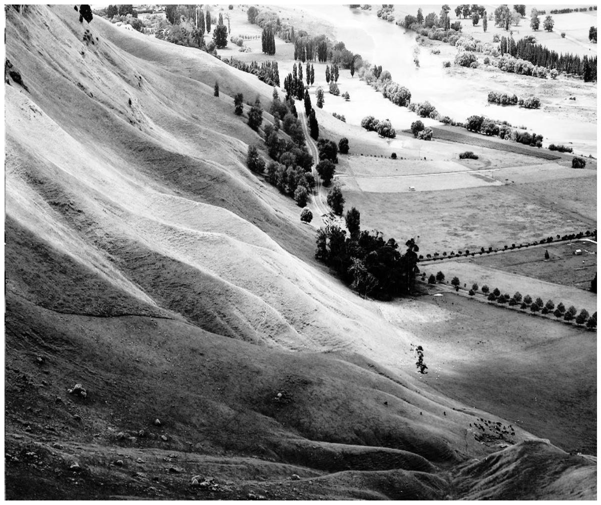
Part Three: Histogram Equalization



Consider the image of a countryside to the right of this paragraph.[*] Notice that this picture seems hazy, and that there is not much contrast in the image. It would be nicer if we could sharpen the contrast in this picture to make individual features pop out more. Ideally, we could sharpen the contrast in this picture to reveal more details. Doing so might give us back a picture like the one below the initial, washed-out image.

In this final part of the assignment, you will implement an algorithm called ***histogram equalization*** that sharpens the contrast in an image, often leading to much clearer pictures.

Luminance



Inside the computer, colors are usually represented as RGB triplets, with each component ranging from 0 to 255. An RGB triplet encodes the intensity of the red, green, and blue channels of some particular color. For example, (255, 0, 0) is an intense red color, (0, 255, 0) is an intense green color, and (0, 0, 255) is an intense blue color. However, if you were to look at three squares of these colors side-by-side, you would not see them as having the same brightness. The human eye perceives some colors as brighter than others, much in the same way that it perceives tones of certain frequencies as louder than others. Because of this, the green square would appear dramatically brighter than the red and blue

* Images from http://en.wikipedia.org/wiki/File:Unequalized_Hawkes_Bay_NZ.jpg and http://en.wikipedia.org/wiki/File:Equalized_Hawkes_Bay_NZ.jpg

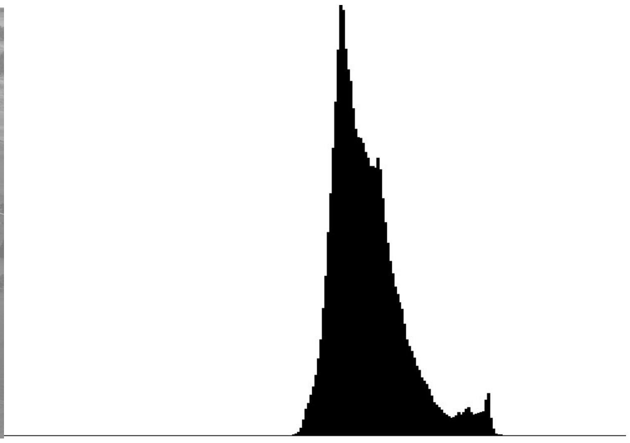
squares and the red square would appear marginally brighter than the blue square. Given an RGB triplet, it is possible to compute a *luminance* value that represents, intuitively, the perceived brightness of a color. Like RGB values, luminance values range from 0 to 255, inclusive.

Manipulating an image by changing its RGB triplets will allow you to transform an image by changing the intensities of different color channels. In this part of the assignment, you will instead do computations over the *luminances* of the pixels in an image rather than the individual color components. This will change the apparent brightness of different parts of the image, which makes it possible to increase or decrease the contrast within that image.

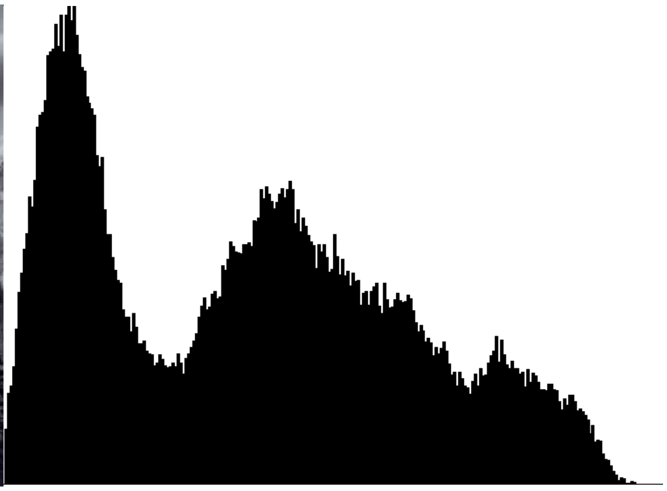
Image Histograms

Given an image, there may be multiple different pixels that all have the same luminance. An *image histogram* is a representation of the distribution of luminance throughout that image. Specifically, the histogram is an array of 256 integers – one for each possible luminance – where each entry in the array represents the number of pixels in the image with that luminance. For example, the zeroth entry of the array represents the number of pixels in the image with luminance zero, the first entry of the array represents the number of pixels in the image with luminance one, the second entry of the array represents the number of pixels in the image with luminance two, etc.

Looking at an image's histogram tells you a lot about the distribution of brightness throughout the image. For example, here is the original picture of the countryside, along with its image histogram:



Compare this to a picture with more contrast, along with its histogram:[*]



* Image taken from http://anseladams.com/wp-content/uploads/2012/03/1901006-2-412x300.jpg

Images with low contrast tend to have histograms more tightly clustered around a small number of values, while images with higher contrast tend to have histograms that are more spread out throughout the full possible range of values. In this part of the assignment, you will implement an algorithm that tries to increase an image's contrast by spreading out its histogram through a wider range of luminances.

Related to the image histogram is the *cumulative histogram* for an image. Like the image histogram, the cumulative histogram is an array of 256 values – one for each possible luminance. Unlike the image histogram, which is computed directly from the image, the cumulative histogram is computed purely from the image histogram. The cumulative histogram is defined as follows: if ***H*** is the image histogram and ***C*** is the cumulative histogram, then

$$C[n] = H[0] + H[1] + \dots + H[n]$$

For example, the zeroth entry of the cumulative histogram is the zeroth term of the image histogram, the first entry of the cumulative histogram is the sum of the zeroth and first terms of the image histogram, and second entry of the cumulative histogram is the sum of the zeroth, first, and second terms of the image histogram, etc. As an example, if the first few terms of the image histogram were

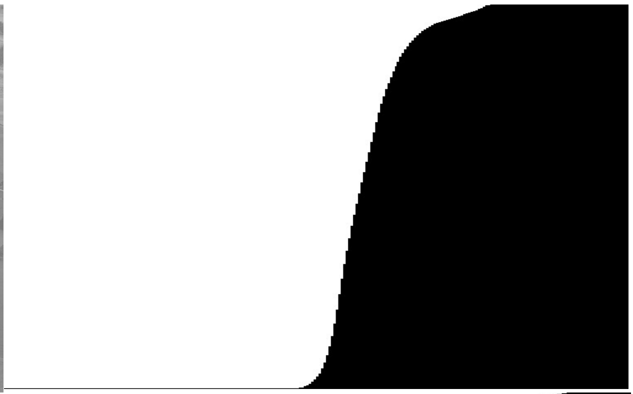
2, 3, 5, 7, 11, 13, …

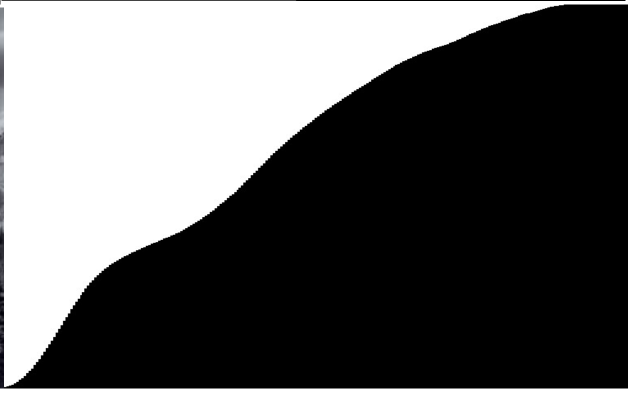
Then the first few terms of the corresponding cumulative histogram would be

2, 5, 10, 17, 28, 41, …

One way to get an appreciation for the cumulative histogram is as follows. Given the image histogram, the *n*th entry of that histogram describes the total number of pixels with luminance exactly *n*. Given the cumulative histogram, the *n*th entry of that histogram describes the total number of pixels with luminance *less than or equal to n*.

Below are the cumulative histograms for the two above images. Notice how the low-contrast image has a sharp transition in its cumulative histogram, while the normal-contrast image tends to have a smoother increase over time.





The Histogram Equalization Algorithm

We are now ready to actually describe the histogram equalization algorithm.

Suppose that we have a pixel in the original image whose luminance is 106. Since the maximum possible luminance for a pixel is 255, this means that the "relative" luminance of this image is 106 / 255 ≈ 41.5%, meaning that this pixel's luminance is roughly 41.5% of the maximum possible luminance. Assuming that all intensities were distributed uniformly throughout the image, we would expect this pixel to have a brightness that is greater than 41.5% of the pixels in the image.

Similarly, suppose that we find a pixel in the original image whose luminance is 222. The relative luminance of this pixel is 222 / 255 ≈ 87.1%, so we would expect that (in a uniform distribution of intensities) that this pixel would be brighter than 87.1% of the pixels in the image.

The histogram equalization algorithm works by trying to change the intensities of the pixels in the original image such that if a pixel is supposed to be brighter than *X*% of the total pixels in the image, then it is mapped to an luminance that will make it brighter than as close to *X*% of the total pixels as possible. This turns out to be not nearly as hard as it might seem, especially if you have the cumulative histogram for the image.

Here's the key idea behind the algorithm. Suppose that an original pixel in the image has luminance *L*. If you look up the *L*th entry in the cumulative histogram for the image, you will get back the total number of pixels in the image that have luminance *L* or less. We can convert this into a fraction of pixels in the image with luminance *L* or less by dividing by the total number of pixels in the image:

$$fractionSmaller = \frac{cumulativeHistogram[L]}{totalPixels}$$

Once we have the fraction of pixels that have intensities less than or equal to the current luminance, we can scale this number (which is currently between 0 and 1) so that it is between 0 and 255, which gives a valid luminance value. The overall calculation is the following:

$$newLuminance = \frac{\text{MAX\_LUMINANCE} \cdot cumulativeHistogram[L]}{totalPixels}$$

The histogram equalization algorithm is therefore given by the following:

1. Compute the histogram for the original image.
2. Compute the cumulative histogram from the image histogram.
3. Replace each luminance value in the original image using the above formula.

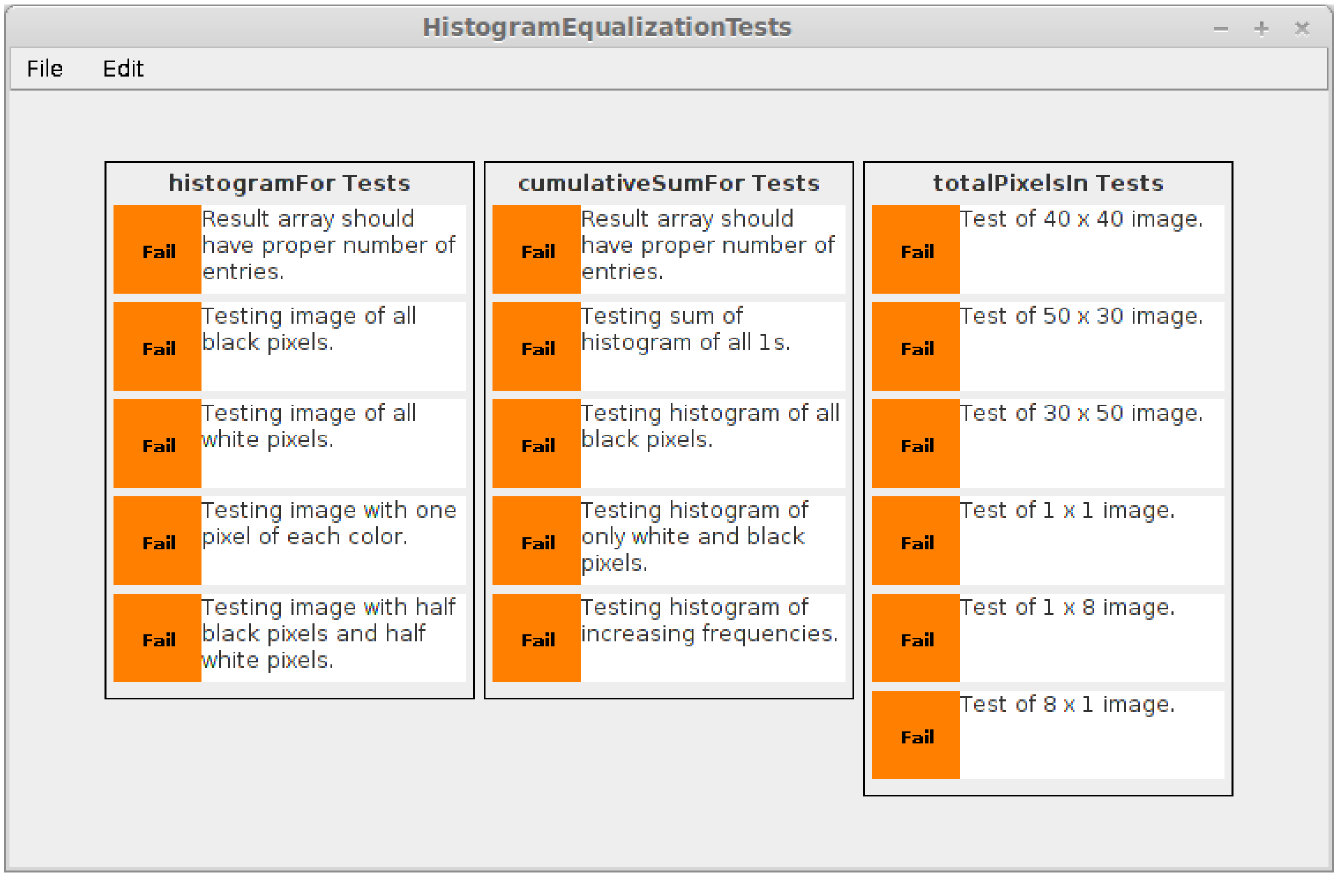
The Assignment

Your job is to implement these methods in the **`HistogramEqualizationLogic`** class:

```java
public class HistogramEqualizationLogic {
      /**
       * Given the luminances of the pixels in an image, returns a histogram of
       * the frequencies of those luminances.
       * <p>
       * You can assume that pixel luminances range from 0 to MAX_LUMINANCE,
       * inclusive.
       *
       * @param luminances The luminances in the picture.
       * @return A histogram of those luminances.
       */
      public static int[] histogramFor(int[][] luminances) {
            /* TODO: Implement this! */
      }
      /**
       * Given a histogram of the luminances in an image, returns an array of the
       * cumulative frequencies of that image.  Each entry of this array should be
       * equal to the sum of all the array entries up to and including its index
       * in the input histogram array.
       * <p>
       * For example, given the array [1, 2, 3, 4, 5], the result should be
       * [1, 3, 6, 10, 15].
       *
       * @param histogram The input histogram.
       * @return The cumulative frequency array.
       */
      public static int[] cumulativeSumFor(int[] histogram) {
            /* TODO: Implement this! */
      }
      /**
       * Returns the total number of pixels in the given image.
       *
       * @param luminances A matrix of the luminances within an image.
       * @return The total number of pixels in that image.
       */
      public static int totalPixelsIn(int[][] luminances) {
            /* TODO: Implement this! */
      }
      /**
       * Applies the histogram equalization algorithm to the given image,
       * represented by a matrix of its luminances.
       * <p>
       * You are strongly encouraged to use the three methods you have implemented
       * above in order to implement this method.
       *
       * @param luminances The luminances of the input image.
       * @return The luminances of the image formed by applying histogram
       *         equalization.
       */
      public static int[][] equalize(int[][] luminances) {
            /* TODO: Implement this! */
      }
}
```

We recommend implementing the methods in this class in the order in which they appear.

To help you test out your implementation, we have provided you with a test harness that will run your methods on a variety of different inputs. If you run this program without implementing any of the methods, you will see a window like this:



histogramFor Tests

- Fail — Result array should have proper number of entries.
- Fail — Testing image of all black pixels.
- Fail — Testing image of all white pixels.
- Fail — Testing image with one pixel of each color.
- Fail — Testing image with half black pixels and half white pixels.

cumulativeSumFor Tests

- Fail — Result array should have proper number of entries.
- Fail — Testing sum of histogram of all 1s.
- Fail — Testing histogram of all black pixels.
- Fail — Testing histogram of only white and black pixels.
- Fail — Testing histogram of increasing frequencies.

totalPixelsIn Tests

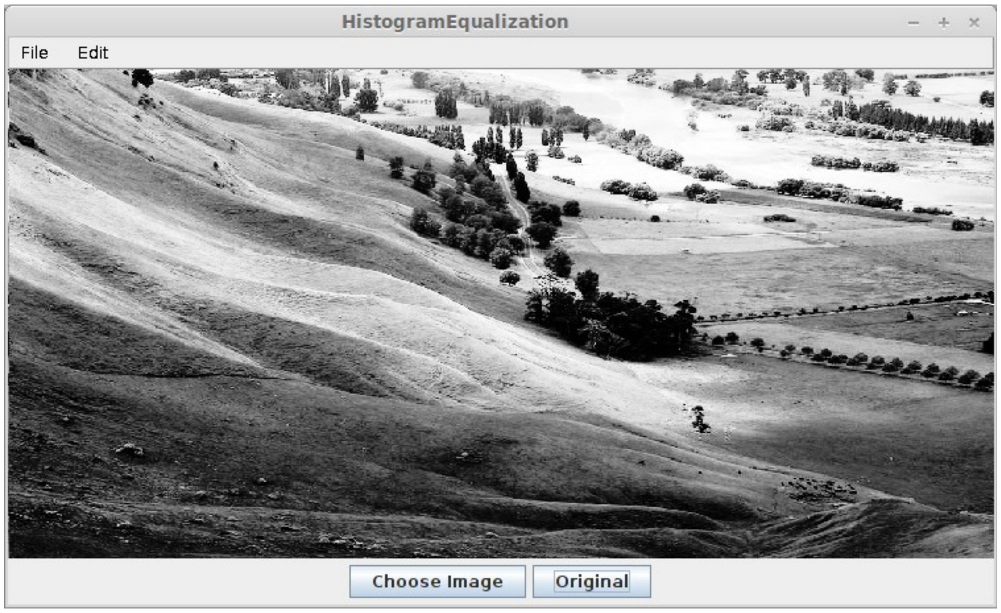
- Fail — Test of 40 x 40 image.
- Fail — Test of 50 x 30 image.
- Fail — Test of 30 x 50 image.
- Fail — Test of 1 x 1 image.
- Fail — Test of 1 x 8 image.
- Fail — Test of 8 x 1 image.

Each of the colored rectangles represents a single test case that we have written to check whether your implementation works. If your implementation of any of the initial methods is incorrect, there is a good chance that it will give back an incorrect answer for one of these tests. Consequently, a test failure indicates that you probably have a bug somewhere in the indicated method. On the other hand, if all the tests pass, that probably (but not definitely) means that your implementation is working correctly.

The result of each test is color-coded as follows:

- **Green:** This indicates that the test passed successfully. You should aim to make all tests green!
- **Yellow:** This indicates that the test is still running. Normally, tests should complete quickly, so if you see this rectangle it likely means that your code contains an infinite loop.
- **Orange:** This indicates that the test completed, but that your method did not pass the test.
- **Red:** This indicates that the test failed to complete. This probably means that your method caused an error before returning. You can click on the red rectangle to see exactly what exception was generated.

The tests in this program only cover the first three methods. You can check whether the **`equalize`** method works by running the **`HistogramEqualization`** program we've provided, which will use your **`equalize`** method to adjust the contrast in an image. A snapshot of this program is shown below:



Advice, Tips, and Tricks

This part of the assignment is probably the most complicated, with more math required than the other steps. A small math error in one step can easily lead to grossly incorrect values later on, causing strangely distorted results.

To combat this, we *strongly* recommend using our testing infrastructure when writing this program and not trying to write the entire program in one sitting without testing it. Build each method independently and test them as you go. Once you think you have everything working, then try to to write the final method, which glues everything together.

Also, be careful with integer division and casting in the last step. You will be dividing **`int`**s by one another, and it is extremely easy to accidentally get 0 for your answers by using integer division and rounding down.

Note that the histogram equalization algorithm that you will be implementing will always convert color images to grayscale images in order to show off the high contrast. Don't worry if the colors aren't showing up correctly; they're not expected to.

Possible Extensions

There are a huge number of possible extensions for this assignment. Here are a few suggestions to help give you some ideas:

- **Steganography**: You could consider using all three color channels to store hidden information, not just the red channel. This would let you hide images inside the color image that are larger than that original image, or which are not just black and white. You could also try to find a way to encode text information inside of a color picture by using multiple different pixels to store a single character.
- **Tone Matrix**: What kind of music could you make if you could turn the lights on in different colors, where each color played a different instrument? Or what if you changed the sound samples so that the tone matrix sounded like a concert piano?
- **Histogram Equalization**: Could you try balancing the *colors* in the image, not just the luminance? Can you brighten or darken the image by shifting the histogram upward or downward?

Some of these extensions might require you to make changes to some of the starter files that we provide. You can download the complete source for each program from the course website and use modify them to your heart's content. We have tried to make these programs as easy to read as possible, though they do use some language features we haven't discussed yet. You might want to read over Chapter 10 of *The Art and Science of Java* to learn more.

Good luck!