Handout 20
February 18, 2015

Assignment 6: Array Algorithms

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

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 (**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. Part Three of this assignment (**Tone Matrix**) lets you use arrays to play sounds and construct an impressive musical instrument.

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. We hope you have a lot of fun working through these problems and playing around with the results!

Due Friday, February 27 at 3:15PM

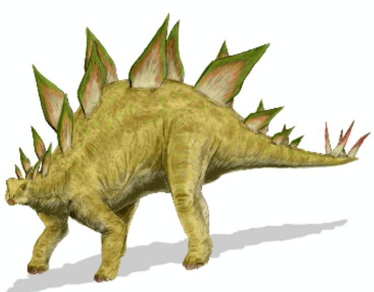
Part One: Steganography[*]

Suppose that you want to send a secret message to someone else without other people being able to read it. One way to do this would be to hide the message inside something innocuous so that no one would think to look for it. For example, suppose that I send you the following note:

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

This message looks like a comment on Stanford's (wonderful) 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 **nubian ibex**, which was the secret message I was trying to send to you. The secret message isn't encrypted – anyone who knows where to look for the message can still read it – but has been hidden so that people might not even notice it exists. The art and science of concealing hidden messages is called *steganography*. In this part of the assignment, you'll implement a system of image steganography to conceal messages in pictures.

Your program will let the user hide black-and-white images (the secret message) inside of full-color images. As an example, consider the picture of a stegosaurus to the left. This looks like a totally normal picture, but surprisingly there is a second image embedded within it: a beautifully, artistically rendered, scientifically accurate picture of a Tyrannosaurus Rex, which is shown to the right. This secret image was embedded in the main image by slightly modifying pixel colors in the original image in a way that is virtually impossible to notice with the naked eye.



Each pixel in an image is represented by the intensity of its red, green, and blue components. Although every combination of red, green, and blue produces a different color, the human eye is not sensitive enough to distinguish all of these colors. For example, the color magenta has RGB values (255, 0, 255), with the red and blue components at maximum and the green component at zero. The related triplet (254, 0, 255) is also an intensely magenta color. While (254, 0, 255) is not exactly the same color as (255, 0, 255), they are so similar that the human eye can't tell them apart.

The image steganography system you'll be implementing works as follows. You'll start off with two images: a secret black-and-white image, and the full-color master image. For simplicity, you can assume that these images are always the same size as one another. You will then process the black-and-white image one pixel at a time, making minor adjustments to the corresponding pixels in the color image to encode whether the secret pixel is black or white. Specifically you will do the following:

- 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***.

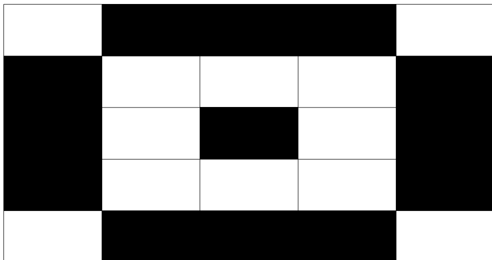
For example, if the secret message pixel is black and the color image pixel has color (255, 0, 127), you will leave the color image pixel unchanged because its red channel (255) is already odd. However, if the pixel from the secret message was black and the color pixel had RGB value (100, 100, 100), you should change the pixel from the color image to have RGB value (101, 100, 100), which is close to the original pixel value but has an odd red channel value. If the secret message pixel was white and the

---

* 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.

color pixel has RGB value (0, 0, 0), you would leave the original pixel untouched because its red channel (0) is already even. However, if the secret pixel was white and color pixel has value (255, 255, 255), you would change the color pixel to (254, 255, 255), which is close to the original color but with an even number in its red channel. These changes are so subtle that they're completely imperceptible and the resulting image will look perfectly normal, but the changes are prominent enough for the computer to use them to reconstruct the hidden message later on.

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):

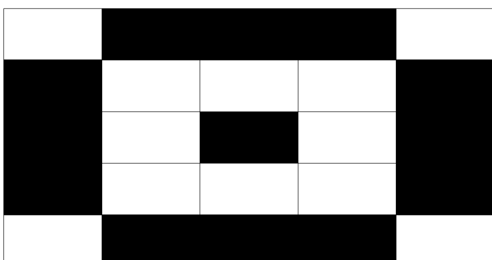


Black-and-White Image

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

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:



Black-and-White Image

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

Modified Color 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 in 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`, while the black-and-white image will be represented as a `boolean[][]`. 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:

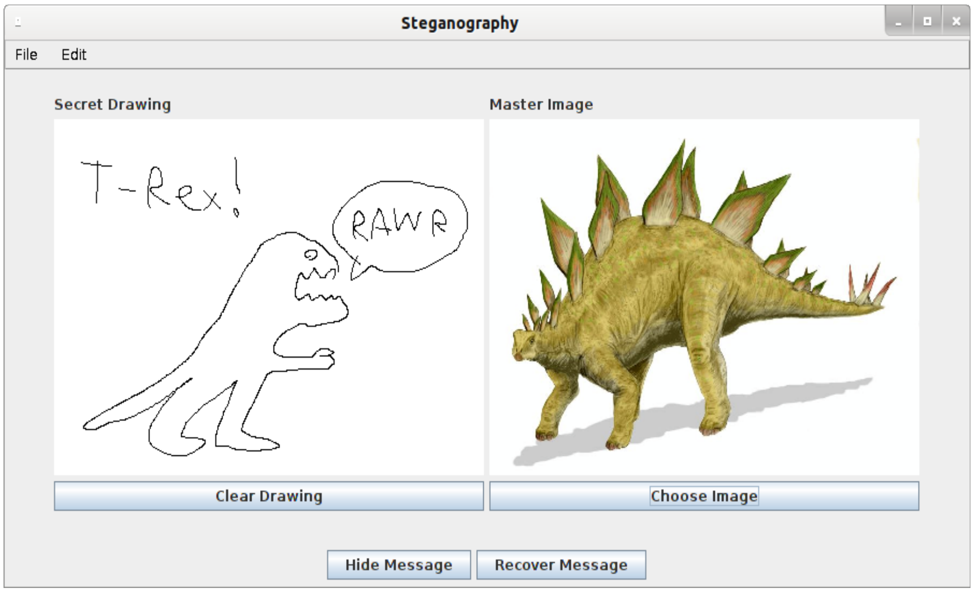
```java
public class SteganographyLogic {
	/**
	 * Given a GImage containing a hidden message, finds the hidden message
	 * contained within it and returns a boolean array containing that message.
	 * <p>
	 * A message has been hidden in the input image as follows. For each pixel
	 * in the image, if 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 by returning a `boolean[][]` containing the pixels of the hidden image. The second, `hideMessage`, takes in a `boolean[][]` 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 inside the `GImage`.

We have provided you a `Steganography` program that uses your implementation of these methods 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 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.*

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.

Here is a screenshot of this program in action:



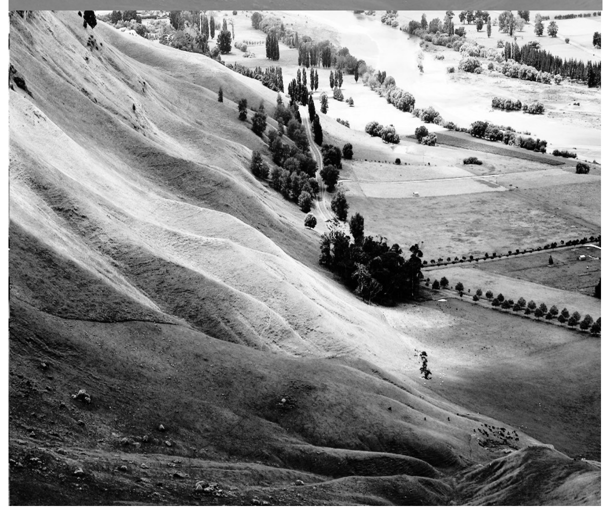
Advice, Tips, and Tricks

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 don't increase the red channel above 255 or decrease it below 0. If you do, the red channel will "wrap around," with values below 0 wrapping back around up to 255 and values above 255 wrapping down to 0. If you do this, you might see bright red or bright cyan patterns in the resulting image due to the red channel flipping between low and high values. If you choose the right way of changing whether a number is odd or even, you won't need to include any special cases to handle this.

Part Two: Histogram Equalization

Consider the image of a countryside to the right of this paragraph.[*] The image is hazy because there isn't much contrast. It would be nice if 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 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 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 because the human eye perceives some colors as brighter than others. Because of this, the green square would appear much brighter than the red and blue 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, with 0 meaning "completely dark" and 255 meaning "completely bright." In this part of the assignment, you will compute over the *luminances* of the pixels in an image rather than the individual color components. This will let you change the apparent brightness of the image, increasing the contrast.

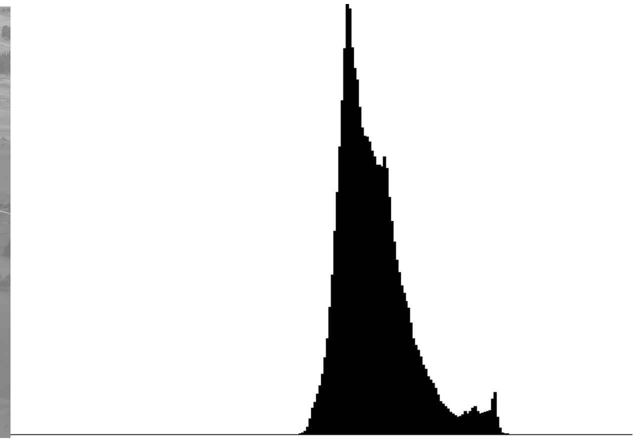
Image Histograms

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.

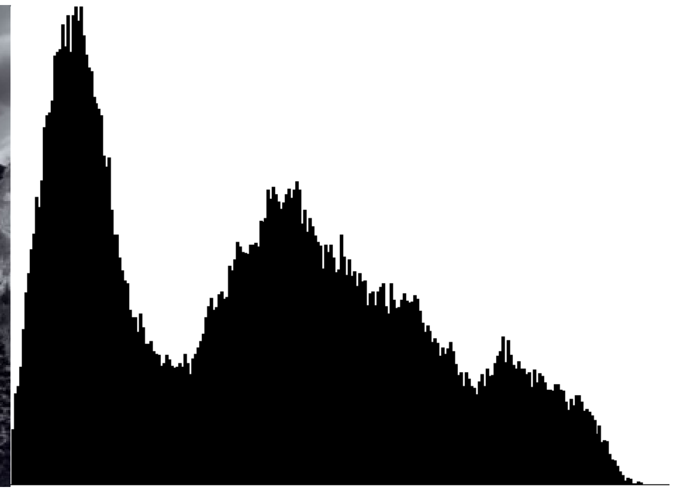
Looking at an image's histogram tells you a lot about the distribution of brightness throughout the image. For example, here is the original picture of the countryside, along with its image histogram:

---

* 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



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



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.

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 original image, the cumulative histogram is computed purely from the image's 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, …

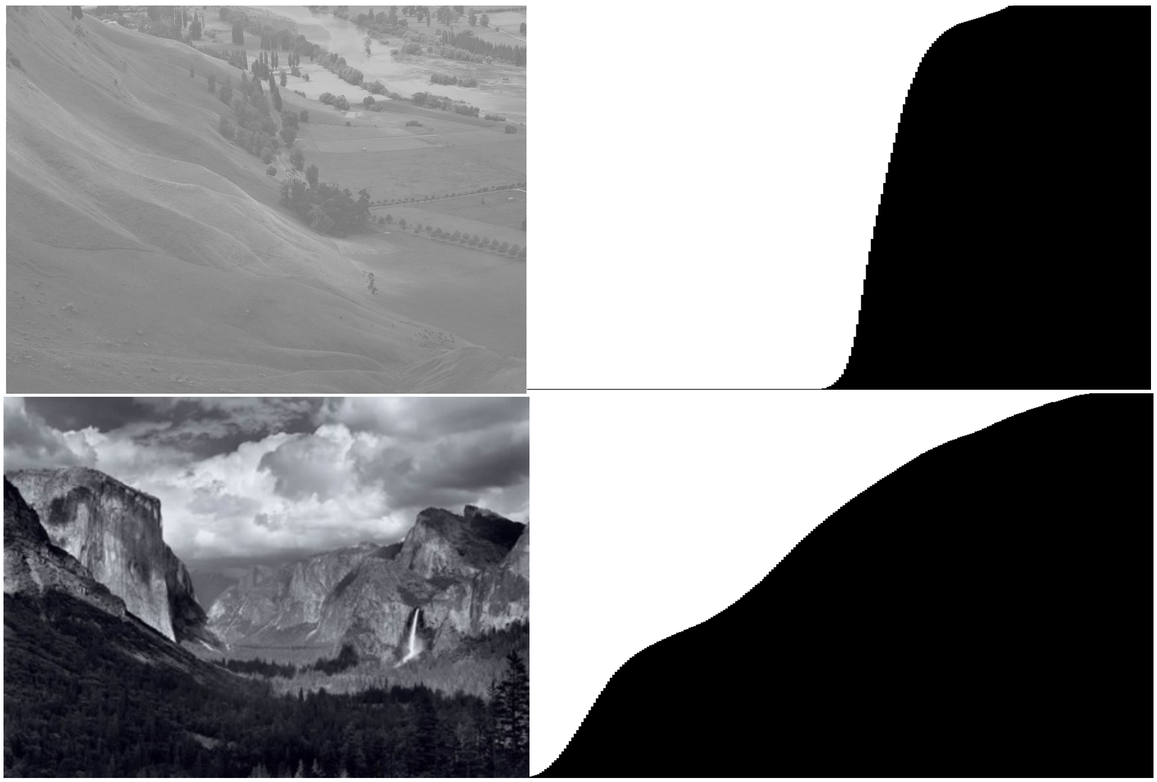
then the first few terms of the corresponding cumulative histogram would be

2, 5, 10, 17, 28, 41, …

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

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.



In this part of the assignment, you will implement an algorithm that increases an image's contrast by spreading out its cumulative histogram through a wider range of luminances.

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 changing the intensities of the pixels in the original image so 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 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 convert that fraction into a luminance value by multiplying it by 255, the maximum possible luminance. The overall calculation is the following:

$$\begin{aligned} newLuminance &= \mathrm{MAX\_LUMINANCE} \cdot fractionSmaller \\ &= \frac{\mathrm{MAX\_LUMINANCE} \cdot cumulativeHistogram[L]}{totalPixels} \end{aligned}$$

The histogram equalization algorithm is therefore given by the following. First, compute the image histogram for the original image. Next, compute the cumulative histogram from the image histogram. Finally, replace each luminance value in the original image using the above formula.

The Assignment

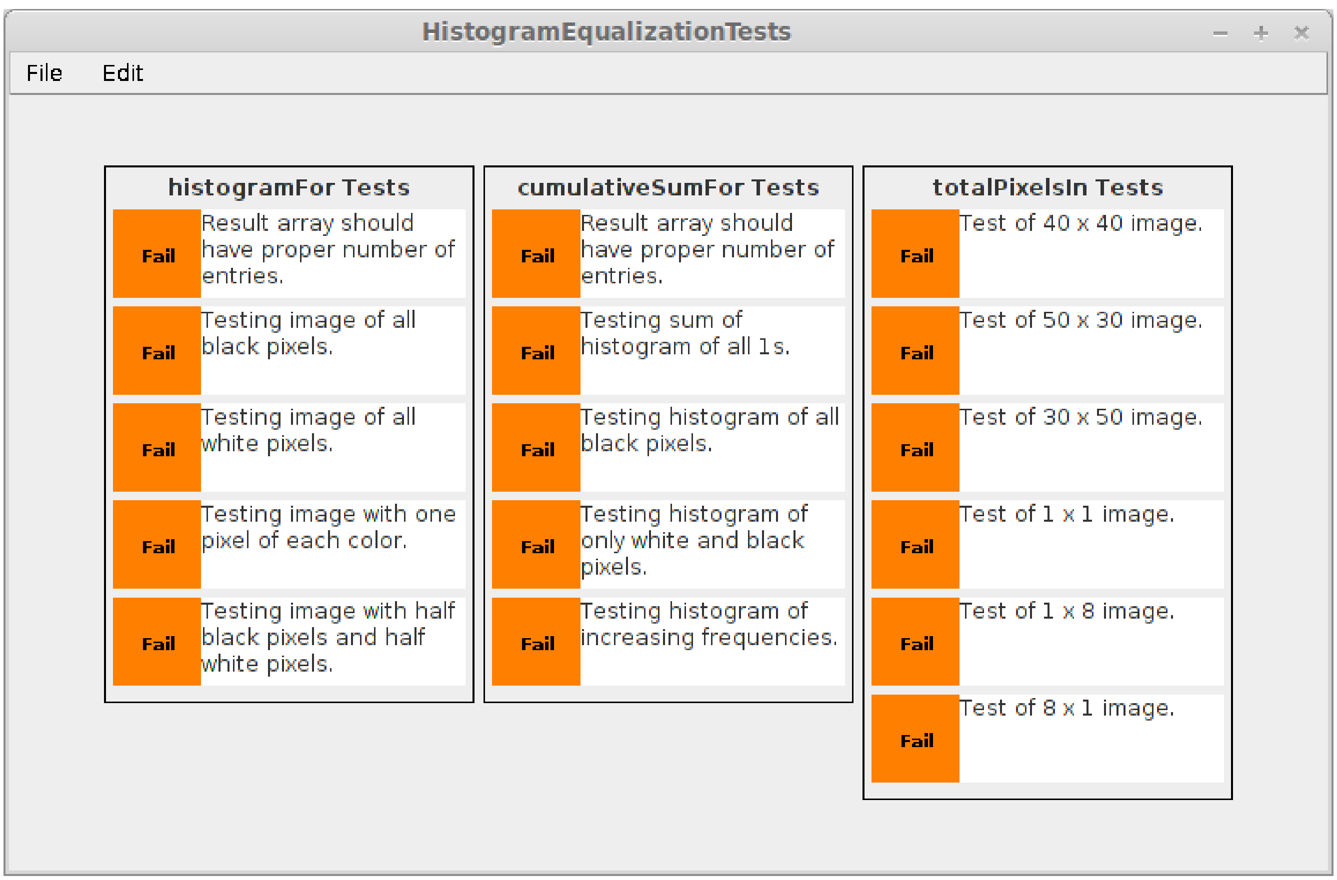
Your job is to implement these methods in the `HistogramEqualizationLogic` class:

```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! */
      }
                            /* … continued on the next page … */
```

```java
	/**
	 * 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:



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.

Advice, Tips, and Tricks

We *strongly* recommend using our testing infrastructure when writing this program and testing as you go. 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.

Be careful with integer division and casting in the last step. You will be dividing `int`s by one another, so be sure not to inadvertently round all the values down.

The histogram equalization algorithm convert color images to grayscale in order to show off the high contrast. Don't worry if your results are always black-and-white; that's the expected result.

Although we haven't talked too much about efficiency in this course, when you're working on the images in this assignment, efficiency will sometimes become a concern. For the purposes of this assignment, you should only worry about efficiency in the following way: in your final code for `equalize`, make sure to call the other helper methods you've written once and exactly once. That avoids repeating the same computation unnecessarily, which would really slow down the program.

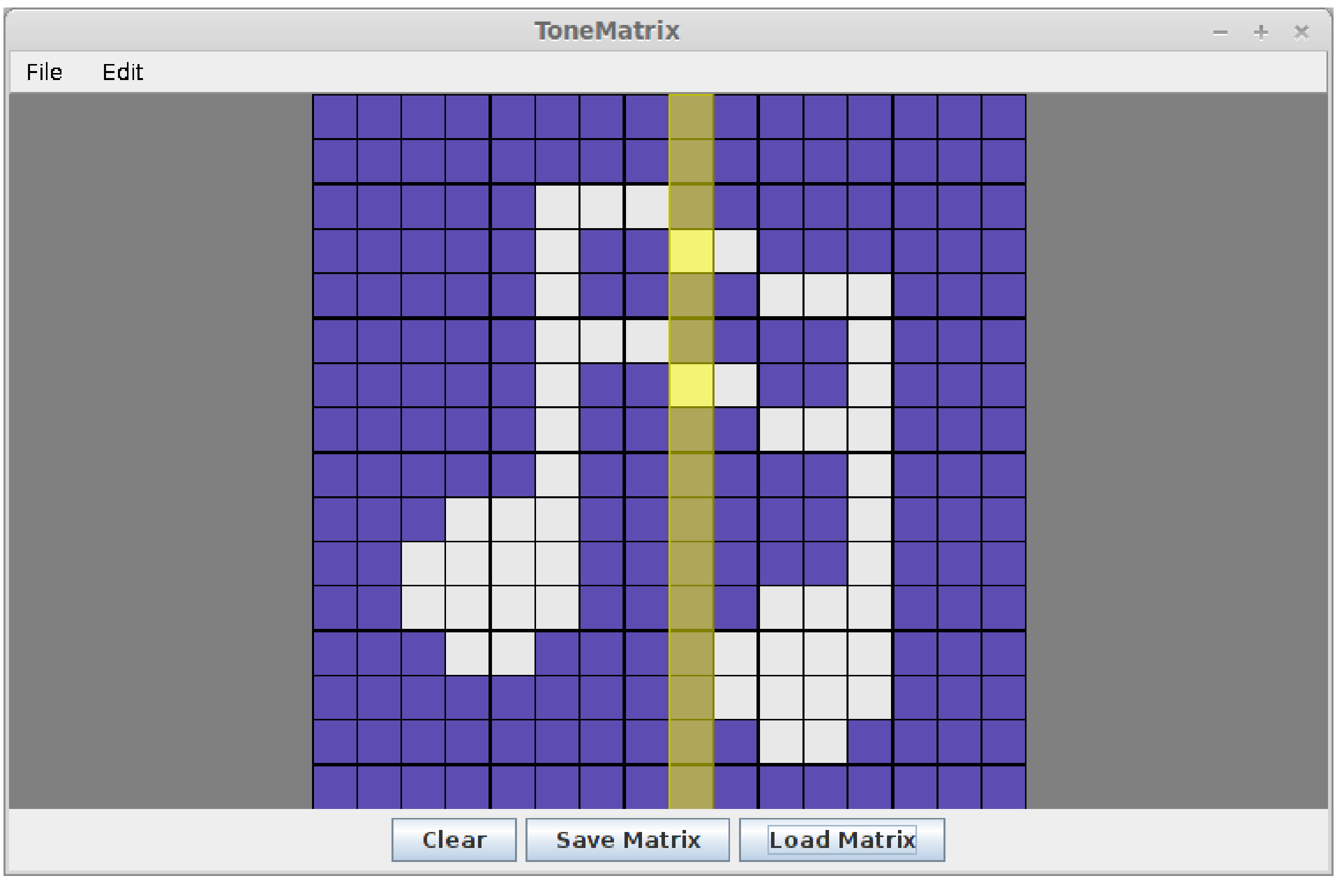
Part Three: Tone Matrix*

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

Before you read the rest of the description of this problem, go visit http://tonematrix.audiotool.com/ and play around with the Tone Matrix. You'll be implementing your own version of that site, and it's probably going to be a lot easier to understand what you'll be doing if you've tried it out a bit. ☺

The Tone Matrix is a 16 × 16 grid of lights, each of which is 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 lights further to the right. The row 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. 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.

Here's a picture of our version of the Tone Matrix in action:

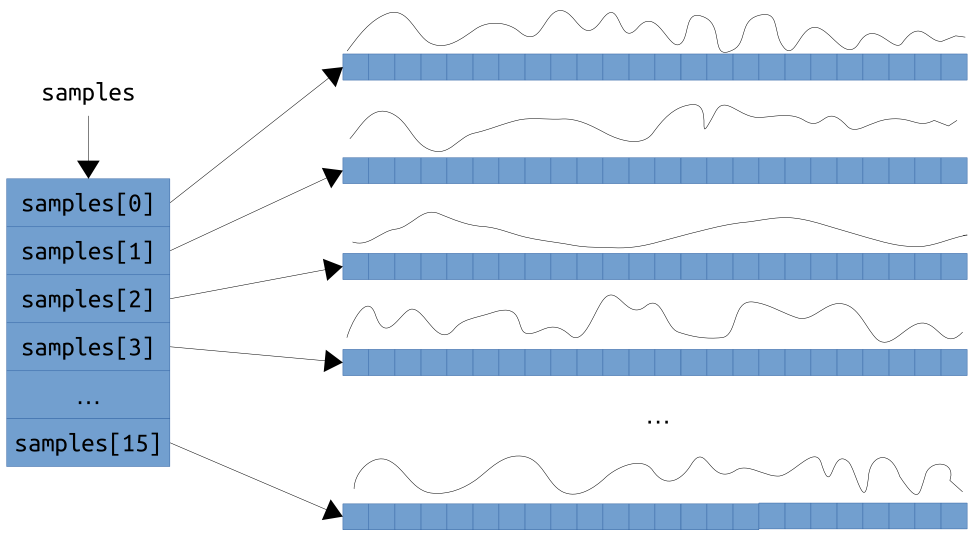


When you start up the Tone Matrix application, you will see a blank grid with a vertical line sweeping from the left to the right. You can click on the squares to turn the lights on and off. Initially, not much will happen, but once you've completed the assignment, the Tone Matrix will start to play music whenever the sweeping line passes over those lights.

* 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 play sounds.

The Assignment

When the program starts up, our starter code automatically generates an array of sound clips representing the sounds each row in the Tone Matrix make. We've represented this as a **double**[][]. While this might look like a grid of **double**s, this is actually an "array of arrays." Each entry in the array is a sound clip, and since sound clips are represented as **double**[]s, the resulting array is a **double**[][], an array of arrays of **double**s. This is shown here schematically:



This samples array has one sound clip for each row in the matrix, since all lights in the same row of the matrix make the same sound.

Your job will be to implement the logic that will determine what sound to play at each point in time. Our provided starter code will continuously sweeps a vertical line across the grid of lights. Whenever the line sweeps over a row, you'll look at which lights are turned on in that row and, from that, determines which sound clips to play. You'll then combines all of those sound clips into a single clip, which our starter code will then automatically send to the speakers. (We'll talk about how to combine the clips together in a second).

Specifically, we'd like you to implement the following method in the `ToneMatrixLogic` class:

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

This method accepts as input a grid of **boolean**s describing which lights are on in the Tone Matrix, a number representing which column in the matrix is selected, and a **double**[][] representing the sound clips associated with each row in the matrix. Your method will then combine together all the sounds corresponding to lit rows in the current column and return the resulting sound clip.

Here's a more detailed breakdown of the parameters to this method:

- **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 the array of sound clips described above. 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 (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 value ToneMatrixConstants.sampleSize().

Part of your challenge in implementing this method will be figuring out how to determine which sound clips need to be combined together. Think about how you can use the information in toneMatrix and the current column index to determine this.

The other challenge in this part of the assignment is actually combining the sound clips together. Recall that we represent each sound clip as an array of **double**s, each of which represents the intensity of the sound wave at some point in time. We'd like you to combine multiple sounds together as follows. First, create a new sound wave whose intensity at each point in time is the sum of the intensities of all the input sound waves at that point in time. For example, if you wanted to combine these three sound waves:

1.0	0.5	0.0	-0.5	-1.0	-0.5	0.0
0.5	0.5	-0.5	-0.5	0.5	0.5	-0.5
-1.0	0.0	1.0	-1.0	0.0	1.0	-1.0

Your method should start by producing this new sound wave:

0.5	1.0	0.5	-2.0	-0.5	1.0	-1.5

This sound wave, if played, will sound like all of the original sound clips are playing at the same time.

Unfortunately, there's a slight problem you'll need to account for. Our sound libraries assume that all sound intensities are in the range [-1.0, +1.0], and if you add up several different sound waves you might end up with a sound wave like the above one where the values are out of range. To fix this, after you've summed all of the sounds together, we'd like you to *normalize* the sound wave. To do so, find the entry in the resulting sound wave with the highest absolute value (in the above case, that would be -2.0), then divide each entry in the sound wave by that amount. This will force all of the values to be between -1.0 and +1.0, resolving the problem.

Applying this process to the resulting sound wave above produces this result, which is what we should actually go and send to the speakers:

-0.25	-0.5	0.25	1.0	0.25	-0.5	0.75

In summary, your method should

- determine which lights are on in the current column,
- add up all the sound clips corresponding to the lit rows,
- divide all of the entries in the resulting sound clip by the maximum-intensity entry, then
- return the result.

Advice, Tips, and Tricks

One challenge in this part of the assignment is determining how to iterate over the arrays that you're given as input. It is probably not a good idea to do a blind double-`for` loop over the arrays you're given as input, since you only need to look at one column of `toneMatrix` and probably only need to look at a subset of the rows in `samples`. Think hard about how you want to iterate over the arrays – if you try iterating in the wrong way, you're likely to hit a roadblock early on.

When you're constructing the new sound wave, proceed in two separate steps. First, sum up all the sound waves corresponding to lights in the current column that are turned on. Then, and only then, find the maximum-intensity entry of that sound wave and divide the wave through by that amount.

As an edge case, if there are no sound clips selected in a given column, just return an array of all 0's. That will cause the Tone Matrix to stay silent for the appropriate length of time. Be careful not to normalize the wave in that case, since otherwise you'll be dividing each entry by zero (do you see why?)

Our provided starter code contains diagnostics that will make sure that the sound clips returned from your method don't contain values out of the range [-1.0, +1.0] and doesn't divide by zero. If the program reports any errors, it might mean that your code for normalizing sound waves is incorrect.

Note that you don't need to actually play the sound that you generate – our provided starter code will do that for you. You just need to create it.

Possible Extensions

There are a lot of possible extensions for this assignment. Here are a few suggestions:

- **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 inside of a color picture by using multiple pixels per 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?

You'll probably have noticed that the starter code we've provided to you just has source code for the files that you are supposed to implement. All the other Java files have been packed up into JAR files, which have the compiled implementation but not the source files.

Some of these extensions require you to change the provided JAR files. You can download the source for each program from the course website and modify them to your heart's content. We've tried to make these programs as easy to read as possible, though they do use some language features we haven't discussed. You may want to read Chapter 10 of *The Art and Science of Java* for more information.

Good Luck!