

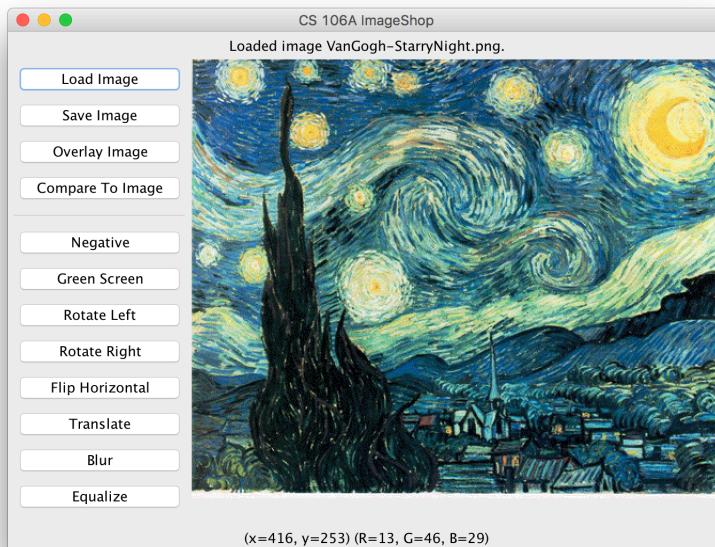
Assignment #5—ImageShop

Due: 11AM PST on Monday, August 7

This assignment may be done in pairs (which is optional, not required)

Based on handouts by Marty Stepp, Eric Roberts and Keith Schwarz

You have probably had occasion to use some sort of image-editing software, whether it is Adobe Photoshop™, Adobe Illustrator™, online image editing tools, or others. In this assignment, you will build a simple version of one of these image editors that implements several useful operations.



The purpose of this assignment is to practice using arrays (1D and 2D) in your programs to manipulate images. We provide you with a completed **GraphicsProgram** called **ImageShopProgram.java** that contains all of the user interaction behavior; you have to write the image manipulation algorithms in a file named **ImageShopAlgorithms.java**.

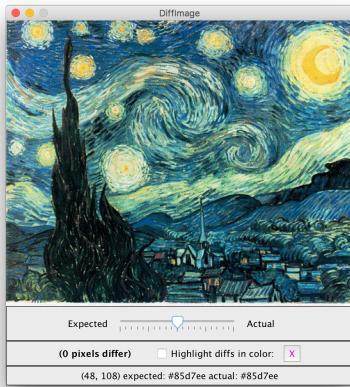
This assignment may be done in **pairs**, or may be done individually. **You may only pair up with someone in the same section time and location.** If you work as a pair, **comment both members' names** on top of every .java file. Make **only one** assignment submission; do not turn in two copies.

In general, you should limit yourself to using Java syntax taught in lecture, and the parts of the textbook we have read, up through the release of this assignment (July 31). If you would like to implement any extensions, please *implement them in separate files*. Clearly comment at the top of this file what extensions you have implemented. Instructions on how to add files to the starter project are listed in the FAQ of the Eclipse page on the course website.

Note that this assignment is not as long as this handout implies – there are simply many included screenshots for each image algorithm!

Output Comparison

Like with past assignments, there is a built-in output comparison tool to check your output. In ImageShop, to compare the displayed image to a particular image file, click “Compare To Image” and select an image to compare it to. A window will pop up comparing the pixels in those images:



In the starter project’s `res/` folder, we include sample images with which to test your algorithms (or you may add your own). In the `output/` folder, we include sample output images for each of the filters you will implement; in particular, all image screenshots on the pages below are included. Please see the Demo JAR posted on the course website for more sample output.

ImageShop Overview

As mentioned previously, the starter project contains a completed `ImageShopProgram.java` file that displays and handles all onscreen buttons, controls, saving and loading images, and more. Your work will be in `ImageShopAlgorithms.java`, where we have left blank methods for each of the image algorithms you must implement. Specifically, each one accepts a `GImage` parameter representing the current onscreen image, and should create and return a *new* `GImage` of that image after that algorithm has been applied. As a reminder, if you have a 2D array of pixels named `pixels`, you can create a new `GImage` with those pixels by saying

```
GImage image = new GImage(pixels);
```

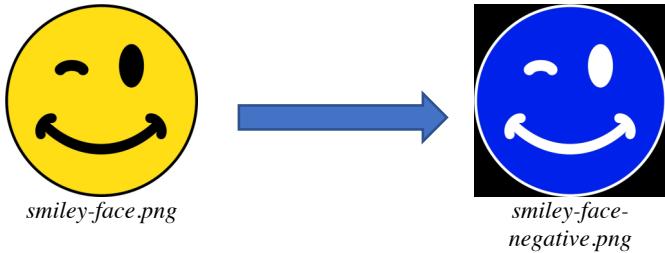
You will implement the following image manipulation algorithms:

negative	Inverts the colors of an image
greenScreen	Removes green pixels so the image can be overlaid on another image
rotateLeft	Rotates the pixels 90° counterclockwise from their current state
rotateRight	Rotates the pixels 90° clockwise from their current state
flipHorizontal	Flips the pixels horizontally (across an imaginary centered vertical line)
translate	Moves the positions of the pixels by a given <i>dx</i> , <i>dy</i> offset
blur	Averages pixel values with neighbors to produce a “softening” effect
equalize	Spreads out pixel intensities to increase contrast

Each algorithm is described in detail on the following pages. **Your algorithms should *not use any instance variables*; each algorithm can be solved on its own without these.** Each algorithm should work on any image of any size, including very large images or very small images such as 1x1 pixels, etc. When describing the algorithms we may refer to pixels in the format (r, g, b) such as (24, 191, 65) to indicate a pixel with a red component of 24, green of 191, and blue of 65.

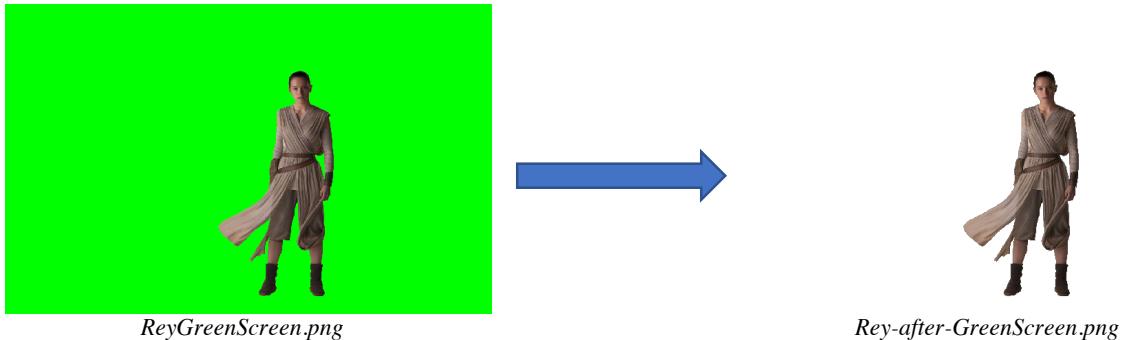
Algorithm #1: Negative

In this method you should create a new image whose pixels are the inverse of those in the source image. To convert an image to its inverse, for each pixel, set all three of its red, green, and blue values to be the inverse of their current color value. The inverse of a color value k is defined as $255 - k$. For example, the pixel (110, 52, 236) has an inverse of (145, 203, 19).



Algorithm #2: Green Screen

Green Screen implements an operation that is used frequently in movies to merge actors into a background scene. The technique uses a particular range of colors (such as green) to represent a background that can later be made transparent via software. The most common colors are green and blue (which give rise to the more specific names *green screen* and *blue screen*) because those colors are most easily differentiated from flesh tones.



Your task in this method is to create a new image whose pixels are the same as those of the source image, but with any green pixels converted to *transparent* ones (the image above at right has a transparent background). Since the green pixels in an image will rarely be precisely equal to Java's **Color.GREEN**, you should treat a pixel as green if its green component **is at least twice as large as the maximum of its red and blue component**. The **Math.max** method may come in handy here; it takes 2 numbers as parameters and returns the larger one:

```
int bigger = Math.max(23, 42);           // returns 42
```

To create a *transparent* pixel, you need to specify a *fourth* value stored inside a pixel (alongside its RGB values), between 0 and 1, called the *alpha* value. An alpha of **0** means the pixel is completely transparent and will show whatever is beneath it. An alpha of **1** (the default) means the pixel is completely opaque and will show just the RGB color it stores. You can specify any alpha value between 0 and 1. Luckily, **GImage.createRGBPixel** lets you optionally specify the alpha as a 4th parameter:

```
// completely transparent (1, 1, 1) pixel
int transparentPixel = GImage.createRGBPixel(1, 1, 1, 0);
```

```
// (1, 1, 1) pixel with alpha 0.35
int opaquePixel = GImage.createRGBPixel(1, 1, 1, 0.35);

// completely opaque (1, 1, 1) pixel (no alpha required)
int opaquePixel = GImage.createRGBPixel(1, 1, 1);
```

Once an image has been “green-screened”, you can overlay it on top of another image. For example, we can take our **Rey-after-GreenScreen.png** image and put it on top of **MilleniumFalcon.png** to look like this:

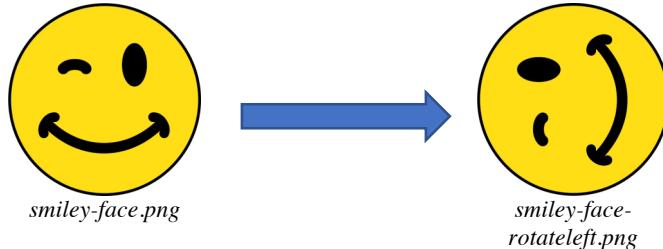


ReyMilleniumFalcon.png

To do this, first run the Green Screen algorithm on an image, and save it to a file. Then, load in the background image you would like to use, and click “**Overlay Image**” to overlay another image on top. Select a previously-saved green-screened image, and it will be added on top.

Algorithm #3: Rotate Left

In this method you should create a new image whose pixel positions are rotated 90-degrees **counter-clockwise** relative to those in the source image.

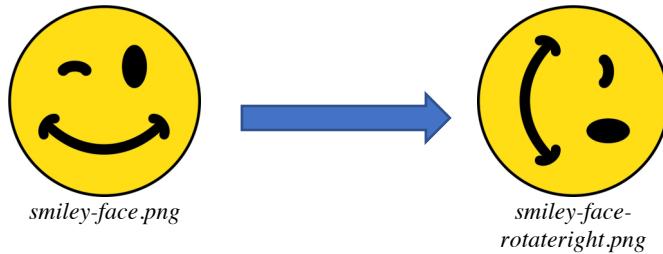


For example, the diagram below shows a source image of size 6x4 rotated left to create a result image of size 4x6; the pixels are indicated by letters just for illustration. Note that the source image's width may be different from its height, and that the dimensions of the result image reflect the rotation you have performed; the result image is as wide as the source was tall, and as tall as the source was wide.

before rotateLeft							after			
0	1	2	3	4	5		0	1	2	3
0	A	B	C	D	E	F	0	F	L	R
1	G	H	I	J	K	L	1	E	K	Q
2	M	N	O	P	Q	R	2	D	J	P
3	S	T	U	V	W	X	3	C	I	O
						-->	4	B	H	N
							5	A	G	S

Algorithm #4: Rotate Right

In this method you should create a new image whose pixel positions are rotated 90-degrees **clockwise** relative to those in the source image.

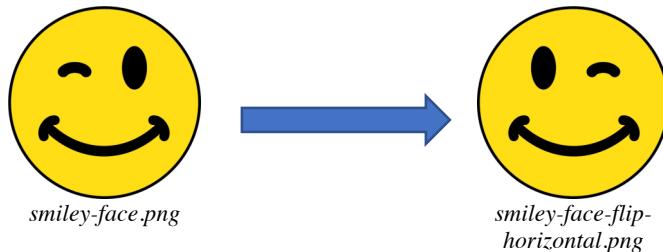


For example, the diagram below shows a source image of size 6x4 rotated right to create a result image of size 4x6; the pixels are indicated by letters just for illustration. Note that the source image's width may be different from its height, and that the dimensions of the result image reflect the rotation you have performed; the result image is as wide as the source was tall, and as tall as the source was wide.

before						rotateRight					after					
0	1	2	3	4	5						0	1	2	3		
0	A	B	C	D	E	F	-->				0	S	M	G	A	
1	G	H	I	J	K	L					1	T	N	H	B	
2	M	N	O	P	Q	R					2	U	O	I	C	
3	S	T	U	V	W	X					3	V	P	J	D	

Algorithm #5: Flip Horizontal

In this method you should create a new image whose pixel positions are flipped across an imaginary *vertical* centered line through the image.



For example, the diagram below shows a source image of size 6x4 flipped horizontally to create a result image of size 6x4; the pixels are indicated by letters just for illustration. Note that the source image's width may be different from its height, and that the dimensions of the result image should always be the same as the source image. Additionally, flipping an image horizontally twice should return it to its original form.

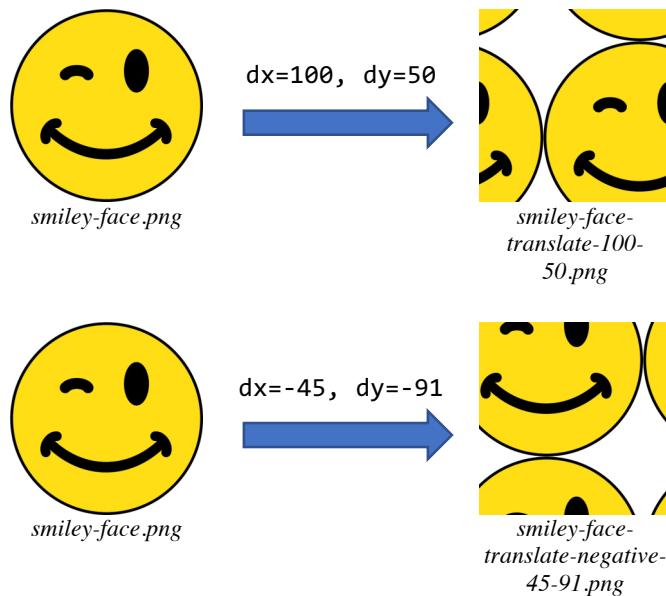
before						flipHorizontal					after					
0	1	2	3	4	5						0	1	2	3	4	5
0	A	B	C	D	E	F	-->				0	F	E	D	C	B
1	G	H	I	J	K	L					1	L	K	J	I	H
2	M	N	O	P	Q	R					2	R	Q	P	O	N
3	S	T	U	V	W	X					3	X	W	V	U	T

Algorithm #6: Translate

In this method you should create a new image whose pixel coordinates have been shifted in the x-dimension by an offset of dx pixels, and shifted in the y dimension by an offset of dy pixels, relative to those in the source image (dx and dy are provided as parameters). For example, if dx is 2 and dy is -1, the pixels of the source are moved by 2px to the right (positive x) and 1px upward (negative y). The resulting image is the same size as the source image. For example, if the 6x4 image below were moved by $dx=2$ and $dy=1$, the pixel at $(x=2, y=3)$ labeled as "U" would move to $(x=4, y=2)$.

before translate by ($dx=2$, $dy=-1$)						after								
0	1	2	3	4	5	0	1	2	3	4	5			
0	A	B	C	D	E	F	0	K	L	G	H	I	J	
1	G	H	I	J	K	L	-->	1	Q	R	M	N	O	P
2	M	N	O	P	Q	R		2	W	X	S	T	U	V
3	S	T	U	V	W	X		3	E	F	A	B	C	D

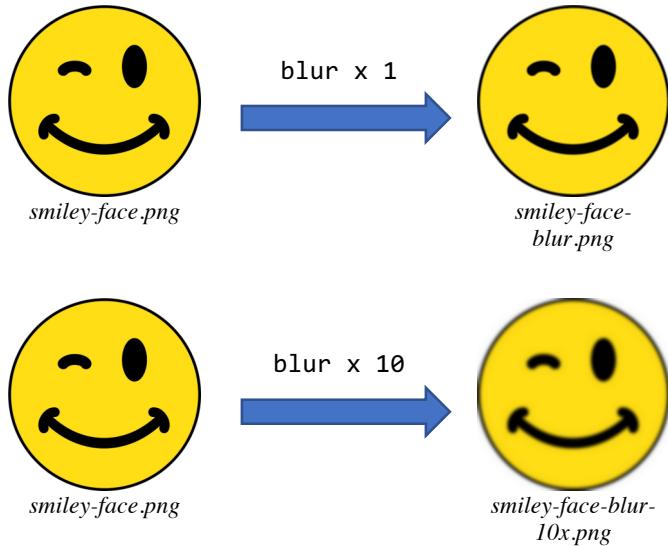
If shifting a pixel causes it to pass the edge of the image, "**wrap**" that pixel around to the opposite side of the image (think of the image as an infinite canvas that "wraps around" on itself in all directions). For example, if dx is 2 and dy is -1, the pixel at $(x=5, y=2)$ labeled as "R" above would normally have shifted to $(x=7, y=1)$, but $x=7$ is off the edge, so we wrap it around to $(x=1, y=1)$ in the result. Similarly, "D" at $(3, 0)$ moves to $(5, 3)$. The dx and/or dy might be **negative** or larger than the overall size of the image. In the example above, translating by a dx of 8, or 14, or 320, or -4, or -6004, etc., would yield the same result as translating by a dx of 2.



Be careful not to confuse the order and meaning of the two indexes in your 2-D array. The first of the two indexes is the row (y), and the second index is the column (x). `pixels[i][j]` refers to a pixel with $(x=j, y=i)$.

Algorithm #7: Blur

In this method you should implement a filter to blur an image. One way to do this is to create a new image whose pixel values are averaged with the values of their immediate neighbors from the source image; this simulates a “blurring” effect between pixels.



The general idea is that for a given pixel (r, c) located at row r and column c in the source image, you will change its red, green, and blue components to be the average (rounded **down** to the nearest integer) of the nine red, green, and blue components in the pixels at locations $(r-1, c-1)$ through $(r+1, c+1)$. For example, in the diagram below, the pixel (row 1, column 2) should be modified to store the average of the nine pixels $(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3), (2, 1), (2, 2)$, and $(2, 3)$. These are the eight neighbors of $(1, 2)$ as well as $(1, 2)$ itself. So the red part of $(1, 2)$ would be changed from 32 to $(84+74+16+66+32+95+28+47+31)/9 = 52$. The green component would be changed from 67 to $(22+38+17+53+67+65+49+21+41)/9 = 41$. The blue component would be changed from 12 to $(99+69+18+88+12+35+31+94+51)/9 = 55$. Therefore the overall pixel value at $(1, 2)$ in the result image would be $(52, 41, 55)$.

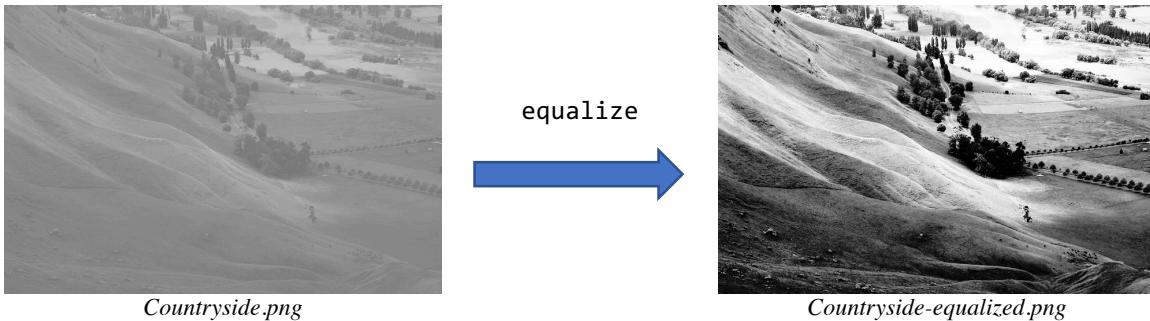
	0	1	2	3	4
0	(14, 97, 63)	(84, 22, 99)	(74, 38, 69)	(16, 17, 18)	(85, 75, 75)
1	(21, 18, 45)	(66, 53, 88)	(32, 67, 12)	(95, 65, 35)	(6, 0, 2)
2	(37, 29, 61)	(28, 49, 31)	(47, 21, 94)	(31, 41, 51)	(246, 84, 13)
3	(82, 33, 90)	(42, 43, 44)	(15, 80, 50)	(60, 40, 12)	(188, 45, 1)

A special case is the set of pixels along the edges of the image. When blurring those pixels, they do not have eight neighbors like other pixels do, so the average includes fewer data points. For example, in the diagram above, the pixel at $(0, 0)$ has no neighbors above or left of it, so it should become the average of the four pixels $(0, 0), (0, 1), (1, 0)$, and $(1, 1)$. So the red component of $(0, 0)$ would become $(14+84+21+66)/4 = 46$, and so on. The pixel at $(3, 3)$ has no neighbors below it, so it should become the average of the six pixels $(2, 2), (2, 3), (2, 4), (3, 2), (3, 3)$, and $(3, 4)$. The red component of $(3, 3)$ would become $(47+31+246+15+60+188)/6 = 97$, and so on. Take care that your algorithm does not crash by trying to access outside the bounds of the array.

A common bug in this algorithm is to try to modify the pixel array in-place. You should not do this; you should create a new second pixel array to store the result image's pixels. The reason is because you don't want modifications made to one pixel to impact another pixel in the same pass over the array. In our previous example, we already stated that pixel $(1, 2)$ should be changed from $(32, 67, 12)$ to $(52, 41, 55)$. But if you store $(52, 41, 55)$ into this pixel and then use that value for further calculations on pixels in the same pass over the array, their averages will be incorrect. For example, when computing the average for pixel $(1, 3)$, the pixel $(1, 2)$ is one of its neighbors. But you should use that pixel's original value of $(32, 67, 12)$ when computing that average.

Algorithm #8: Equalize

Digital processing can do an amazing job of enhancing a photograph. Consider, for example, the countryside image below at left. Particularly when you compare it to the enhanced version on the right, the picture on the left seems hazy. The enhanced version is the result of applying an algorithm called ***histogram equalization***, which grayscales the image and spreads out the intensities to increase its effective contrast and make it easier to identify individual features.



(http://en.wikipedia.org/wiki/File:Unequalized_Hawkes_Bay_NZ.jpg)

Histogram equalization takes advantage of the fact that the human eye perceives some colors as brighter than others, similar to how it perceives tones of certain sound frequencies as louder than others. Green, for example, appears brighter than red or blue, which tend to make images appear darker. Your job is to implement the histogram equalization algorithm, which can be broken down into the following steps, each of which is well-suited to be decomposed into its own method:

- 1) Compute the luminosity histogram for the source image
- 2) Compute the cumulative luminosity histogram from the luminosity histogram
- 3) Use the cumulative luminosity histogram to modify each pixel to increase contrast

Computing the Luminosity Histogram

To compute the luminosity histogram of the source image, we need to first define *luminosity*. Luminosity is a standardized calculation of the “brightness” of a pixel based on its RGB values. It is described on page 439 of The Art and Science of Java, and was also mentioned briefly in class with our grayscale example. The luminosity is an integer between 0 and 255, just as the intensity values for red, green, and blue are. A luminosity of 0 indicates black, a luminosity of 255 indicates white, and any other color falls somewhere in between. There is a provided method called **computeLuminosity** that you can use in **ImageShopAlgorithms.java** that takes RGB values for a pixel and returns its luminosity for those values.

```
int luminosity = computeLuminosity(red, green, blue);
```

Now, we want to compute the *luminosity histogram* of the source image, which represents the distribution of brightness in the source image. Specifically, it's an array of 256 integers – one for each possible luminosity value – where each entry in the array represents the **number of pixels in the image** with that luminosity. For example, the entry at index 0 of the array represents the number of pixels in the image with luminosity 0, the entry at index 1 represents the number of pixels in the image with luminosity 1, and so on.

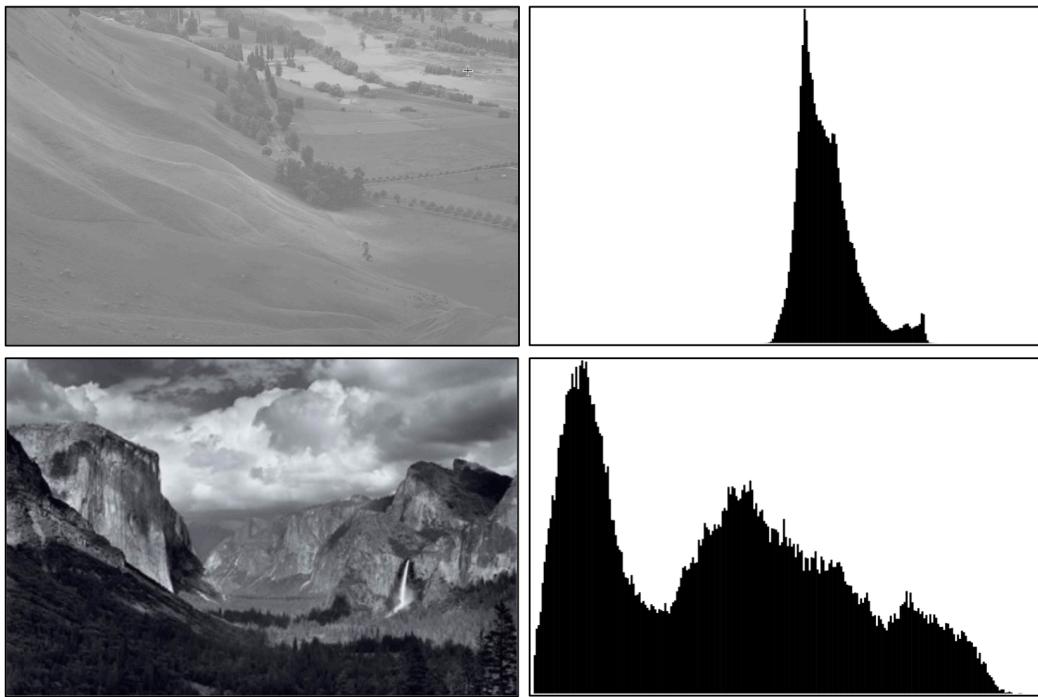


Image source: <http://anseladams.com/wp-content/uploads/2012/03/1901006-2-412x300.jpg>

Visualization of two images' luminosity histograms.

An image's luminosity histogram says a lot about the distribution of brightness throughout the image. The example above shows the original low-contrast picture of the countryside at the top, along with its image histogram. The bottom row shows an image and histogram for a high-contrast picture. 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. We will eventually use this histogram to modify images to spread their brightness distributions out to be more like the lower image.

Compute the Cumulative Luminosity Histogram

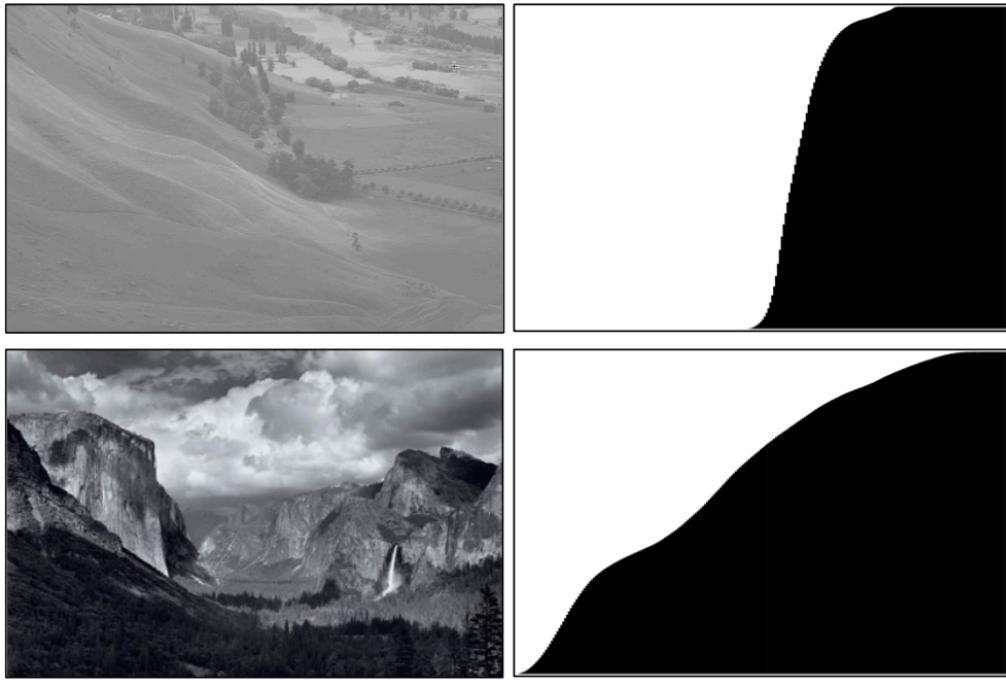
Now we need to take the luminosity histogram from the previous step and from it create the *cumulative luminosity histogram*, which is useful later in the algorithm. The cumulative luminosity histogram is the same size as the regular luminosity histogram; it's also an array of 256 integers, one for each possible luminosity value. However, instead of each entry in the array representing the number of pixels in the image **with that luminosity**, each entry represents the number of pixels in the image **with that luminosity or less**. For example, the entry at index 2 of the array represents the number of pixels in the image with luminosity 0, 1 or 2, the entry at index 3 represents the number of pixels in the image with luminosity 0, 1, 2 or 3, and so on. As an example, if the first six entries in the image histogram were

0	1	2	3	4	5
1	3	5	7	9	11

the corresponding cumulative histogram would be

0	1	2	3	4	5
1	4	9	16	25	36

As an example, the value at index 3 is 16 because $1+3+5+7$ pixels have a luminosity of 3 **or less**.



Visualization of two images' luminosity histograms.

An image's cumulative luminosity histogram says a lot about the distribution of brightness throughout the image. Notice how the low-contrast image at the top has a sharp transition in its cumulative luminosity histogram, representing the smaller distribution of luminosity values. Meanwhile, the normal-contrast image on the bottom has a smoother increase over time. We will eventually use this histogram to modify images to spread their brightness distributions out to be more like the lower image.

Modify Each Pixel To Increase Contrast

Now that we have the cumulative luminosity histogram, we can use it to modify each pixel to increase the image's overall brightness and contrast. The key is that we want to modify the image to **spread its luminosity values across as much of the range of possible luminosity values as we can**. We can do this via the following steps:

For each pixel in the source image:

Compute the pixel's luminosity

$$\text{new } R, G, \text{ and } B \text{ values} = 255 \times \frac{\# \text{ pixels with luminosity} \leq \text{this pixel's luminosity}}{\# \text{ total pixels}}$$

Modify the pixel at this location to be a grayscale pixel with the above RGB values

To understand how this works, suppose you had a low-contrast 10-pixel image with luminosities only between 125-130. The cumulative histogram for this example image could be as follows:

...	125	126	127	128	129	130	...
0	1	3	5	7	9	10	0

To make this image higher contrast, we want to spread these luminosities out so they occupy *more* of the range of luminosity values than just 125-130; this will result in more variation among pixel luminosities, and thus a better image.

For example, let's take the pixel with luminosity 125. In our cumulative histogram above, there is only 1 total pixel (this one) with luminosity ≤ 125 . Therefore, the percentage of pixels with that luminosity or less is $1/10 = 10\%$. Ideally, this pixel would be **10% bright**, to use as much of the luminosity spectrum as possible. The value that achieves this is **10% of 255** (the maximum luminosity), or 25.5. Thus, we want this pixel to have a luminosity of 25 (round down). One property of luminosity is that, if the R, G and B values of a pixel are the same, the luminosity is just this value. Therefore, we can change the pixel at this location to be a grayscale pixel with an R, G, and B value of 25. Thus, for each pixel we **calculate the percentage of pixels with that luminosity or less**, and multiply this by 255 to get a *new* luminosity for that pixel, which we use for its R, G and B values.

As another example, let's take a pixel with luminosity 129. In our cumulative histogram above, there are 9 pixels with luminosity ≤ 129 . Therefore, the percentage of pixels with that luminosity or less is $9/10 = 90\%$. Ideally, this pixel would be **90% bright**, to use as much of the luminosity spectrum as possible. The value that achieves this is **90% of 255**, or 229.5. Thus, we want this pixel to have a luminosity of 229 (round down). We therefore change the pixel at this location to be a grayscale pixel with an R, G, and B = 229.

Notice how a luminosity of 125 is mapped to a new luminosity of 25, and a luminosity of 129 is mapped to a new luminosity of 229; this dramatically expands the range of luminosity values in the image, resulting in higher contrast and better detail.

Possible Extensions

There are many possibilities for optional extra features that you can add if you like, potentially for a small amount of extra credit. If you are going to do extra features, **submit extra files** containing your extended version (**ImageShopAlgorithmsExtra.java**, **ImageShopProgramExtra.java**, etc.). Note that **ImageShopProgramExtra.java** itself is just a **GraphicsProgram**! At the top of your .java file in its comment header, you must **comment** what extra features you completed. Here are a few ideas:

- **Painting tools:** add drawing tools like a paint brush, line drawing, a touch-up tool, etc.
- **Other filters:** add other filters that interest you, such as red-eye reduction or “posterize” (Shepard Fairey’s iconic design of President Obama’s 2008 campaign poster, which converted all pixels to the closest equivalent chosen from a highly restricted set of colors).
- **Selection tool:** use mouse events to implement a selection tool for things like cropping an image, or for applying a filter to only a portion of an image.
- **Other:** use your imagination! What other features could you imagine in a program like this?

Grading

Functionality: Your code should compile without any errors or warnings. We use the Image Comparison Tool mentioned on page 1 to see that your image *exactly* matches the one expected.

Style: Follow style guidelines taught in class and listed in the course Style Guide. For example, use descriptive names for variables and methods. Format your code using indentation and whitespace. Avoid redundancy using methods, loops, and factoring. Use descriptive comments, including at the top of each .java file, atop each method, inline on complex sections of code, and a citation of all sources you used to help write your program. If you complete any extra features, list them in your comments to make sure the grader knows what you completed. In general, limit yourself to using Java syntax taught in lecture and textbook parts we have read so far.

As mentioned earlier, you should not use *any instance variables* to implement the required parts of this assignment.

Decomposition: Most of the methods you'll need to write on this assignment are already specified for you, but you should still work to avoid redundancy. For example, if two or more specified methods have similar behavior, make one call the other, or make a private method that captures the redundancy and is called by both. In particular, for **Equalize**, we highly recommend breaking down the logic into multiple private methods.

Honor Code: Follow the Honor Code when working on this assignment. Submit your own work and do not look at others' solutions (outside of your pair, if you are part of a pair). Do not give out your solution. Do not search online for solutions. Do not place a solution to this assignment on a public web site or forum. Solutions from this quarter, past quarters, and any solutions found online, will be electronically compared. If you need help on the assignment, please feel free to ask.