CS 61B Project 1
Color Images, Edge Detection, and Run-Length Encodings
Due midnight Saturday, February 22, 2014

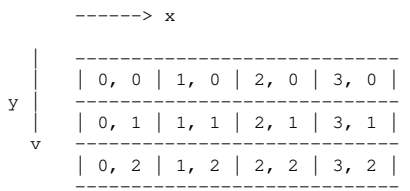
Warning: This project is time-consuming. Start early.

This is an individual assignment; you may not share code with other students.

Getting started: You will find the code for this assignment in ~cs61b/hw/pj1/. Start by copying it into your own pj1 directory.

In this project you will implement two simple image processing operations on color images: blurring and edge detection. You will use libraries to read and write files in the TIFF image format. One option in TIFF files is that they can be compressed if there are many adjacent pixels of the same color; the compressed form is called a run-length encoding. You will write code to convert an image into a run-length encoding and back.

Each image is a rectangular matrix of color pixels, which are indexed as follows (for a 4x3 image):



Note that the origin is in the upper left; the x-coordinate increases as you move right, and the y-coordinate increases as you go down. (This conforms to Java's graphics commands, though you won't need to use them directly in this project.) We use the notation (i, j) to denote the pixel whose x-coordinate is i and whose y-coordinate is j.

Each pixel has three numbers in the range 0...255 representing the red, green, and blue intensities of the pixel. These three bytes are known as the RGB values of the image. A pixel in which all three values are zero is pure black, and a pixel in which all three values are 255 is bright white. Although Java has a "byte" integer type, its range is -128...127, so we will usually use Java's "short" type for methods that take RGB parameters or return RGB values.

Part I: Image Blurring and Edge Detection

This part is worth 40% of your total score. (8 points out of 20).

Implement a class called PixImage that stores a color image. The PixImage class will include methods for reading or changing the image's pixels. It will also include a method for blurring an image and a method for detecting edges in an image. We have provided a skeleton file named PixImage.java that includes prototypes for the public methods the class offers. You are required to provide implementations of all these methods.

A PixImage is described by its size and the RGB values of each pixel, but it is up to you to decide _how_ a PixImage stores a color image. You should certainly use one or more arrays; otherwise, you have some freedom to choose the details.

The size of a PixImage is determined when it is constructed, and does not change afterwards. There is one PixImage constructor, which takes two integers as input, representing the width and height of the image, and returns an image of the specified size. For example, the statement

```
PixImage image = new PixImage(w, h);
```

should create a w x h Image object. In your implementation, you may define any fields, additional methods, additional classes, or other .java files you wish, but you cannot change the prototypes in PixImage.java. We will test your code by calling your public methods directly, so it is important that you follow this rule. Please read PixImage.java carefully for an explanation of what methods you must write. The most important of these are boxBlur(), a simple image blurring algorithm, and sobelEdges(), an edge detection algorithm.

The pixels of a PixImage can be changed with the method setPixel(). However, the methods boxBlur() and sobelEdges() should NEVER change "this" original PixImage; they should construct a new PixImage and update it to show the results. The pixels in the new, output PixImage should depend only on the pixels in "this" original PixImage. To obtain correct behavior, you will be working with two PixImages simultaneously, reading pixels from one and writing (modifying) pixels in the other.

In an image, a pixel not on the boundary has nine "neighbors": the pixel itself and the eight pixels immediately surrounding it (to the north, south, east, and west, and the four diagonal neighbors). For the sake of exposition, we consider each pixel to be its own "neighbor". A pixel on the boundary has six neighbors if it is not a corner pixel; only four neighbors if it is a corner pixel. In both boxBlur() and sobelEdges(), the contents of any particular output pixel depend only on the contents of its neighbors in the input image.

boxBlur() simply sets each output pixel to be the average of its neighbors (including itself) in "this" input PixImage. This means summing up the neighbors and dividing by the number of neighbors (4, 6, or 9). The sum might not be divisible by the number of neighbors, but the output pixel values must be integers, so we will allow Java to round non-integer quotients toward zero (as it always does when it divides one integer by another).

Each color (red, green, blue) is blurred separately. The red input should have NO effect on the green or blue outputs, etc.

boxBlur() takes a parameter "numIterations" that specifies a number of repeated iterations of box blurring to perform. If numIterations is zero or negative, boxBlur() should return "this" PixImage (rather than construct a new PixImage). If numIterations is positive, the return value is a newly constructed PixImage showing what "this" PixImage would become after being blurred "numIterations" times. IMPORTANT: each iteration of blurring should be writing to a _different_ PixImage than the one produced by the previous iteration. You should NEVER be reading and writing pixels in the same image simultaneously, and you will get the wrong answer if you try.

sobelEdges() implements the Sobel edge detection algorithm. The output PixImage will be a grayscale image (i.e. red == green == blue for every pixel) with light pixels where edges appear in "this" input PixImage (i.e. where regions of contrasting colors meets) and dark pixels everywhere else. If you imagine that the image is a continuous field of color, the Sobel algorithm computes an approximate gradient of the color intensities at each pixel.

For each pixel (x, y), you will compute an approximate gradient (gx, gy) for each of the three colors. As with blurring, the intensity of pixel (x, y) in the output PixImage depends only on the neighbors of (x, y) in "this" input PixImage, and the red, green, and blue intensities are treated separately at first. The red gradient (gx, gy) is a 2D vector that locally approximates the direction of greatest increase of the red pixel intensities (and depends ONLY on the red intensities of the pixels.) If two regions of very different red intensities meet at the pixel (x, y), then (gx, gy) is a long vector that is roughly perpendicular to the boundary where the constrasting regions meet.

We compute the red gradient (gx, gy) with the following _convolutions_.

```
        -------------     -----------------------------------------------
        | 1 | 0 | -1 |     | x - 1, y - 1 |   x  , y - 1 | x + 1, y - 1 |
        -------------     -----------------------------------------------
   gx = | 2 | 0 | -2 |  *  | x - 1,   y   |   x  ,   y   | x + 1,   y   |
        -------------     -----------------------------------------------
        | 1 | 0 | -1 |     | x - 1, y + 1 |   x  , y + 1 | x + 1, y + 1 |
        -------------     -----------------------------------------------

        ---------------   -----------------------------------------------
        |  1 |  2 |  1 |   | x - 1, y - 1 |   x  , y - 1 | x + 1, y - 1 |
        ---------------   -----------------------------------------------
   gy = |  0 |  0 |  0 | * | x - 1,   y   |   x  ,   y   | x + 1,   y   |
        ---------------   -----------------------------------------------
        | -1 | -2 | -1 |   | x - 1, y + 1 |   x  , y + 1 | x + 1, y + 1 |
        ---------------   -----------------------------------------------
```

The boxes on the right store the red pixel intensities for the neighbors of (x, y). The convolution operation "*" simply means that we multiply each box on the left with the corresponding box on the right, then sum the nine products. (It's like an inner product, aka dot product, of two vectors of length 9. Note that this is NOT matrix multiplication!) The green and blue gradients are defined likewise.

(If you are interested in further details, see the Wikipedia page http://en.wikipedia.org/wiki/Sobel_operator .)

This gives us three gradient vectors for each pixel (red, green, and blue). Define the _energy_ of a gradient vector (gx, gy) to be the square of its length; by Pythagoras' Theorem, the energy is gx * gx + gy * gy. Define the _energy_ of a pixel to be the sum of its red, green, and blue energies. (If you think of a pixel's three gradients together as being a vector in a six-dimensional space, the pixel's energy is the square of the length of that six-dimensional vector.)

```
energy(x, y) = gx(red)^2 + gy(red)^2 + gx(green)^2 + gy(green)^2 +
               gx(blue)^2 + gy(blue)^2.
```

IMPORTANT: You must compute the energy EXACTLY. The pixel intensities are of type "short", but the energy is usually too large to fit in a "short". Thus you must cast all the gradient vectors to type "long" or "int" BEFORE you compute any squares, and you must keep the results as "long" or "int" for the rest of the computation.

The maximum possible value of a Sobel gradient (for one color) is (1020, 1020), so each pixel's energy (combining all three colors) is a number in the range 0...6,242,400. We have provided a method mag2gray() in the PixImage class that takes a pixel energy of type "long" and flattens it down to a grayscale intensity in the range 0...255. The map is logarithmic, so that images over a wide range of intensities will reveal their edges. However, energies of roughly 5,080 and below map to intensity zero, so very-low contrast edges do not appear in the output PixImage. Don't worry if you don't understand mag2gray(); JUST DON'T CHANGE mag2gray().

In your output PixImage, set the red, green, and blue intensities of the pixel (x, y) to be the value mag2gray(energy(x, y)).

Pixels on the boundary of the output image require special treatment, because they do not have nine neighbors. We treat them by _reflecting_ the image across each image boundary. (Imagine the image is sitting right on the shore of a lake, so an upside-down copy of the image is reflected below it.) Thus, we treat the pixel (-1, 2) as if it had the same RGB intensities as (0, 2), and the pixel (1, height) as if it had the same RGB intensities as (1, height - 1).

Then we compute the Sobel convolutions as usual. The reflections prevent spurious "edges" from appearing on the boundaries of the image.

(Hint: programming will be a lot easier if you write helper functions that do the reflection for you, and use them for every pixel access in your edge detector, so you don't have to think about it again.)

We have provided Java classes to help you see your output images and debug your implementation of Part I, in these files:

```
Blur.java
Sobel.java
```

The main() methods in these classes read an image in TIFF format, use your code to perform blurring and/or edge detection, write the modified image in TIFF format, and display the input and output images. You will need to compile them against the JAI image libraries in the .jar files we have included, which may require you to add the .jar files to your "classpath". In Unix:

```
javac -cp "jai_core.jar:jai_codec.jar" *.java
```

Both programs take one or two command-line arguments. The first argument specifies the name of an input image file in TIFF format. (If you specify no arguments, the programs will remind you how to use them.) The optional second argument specifies the number of iterations of your box blurring filter to perform. For example, if you run

```
java -cp ".:jai_core.jar:jai_codec.jar" Blur image.tiff 3
```

then Blur will load the image from image.tiff, perform three iterations of blurring, write the blurred image to a file named blur_image.tiff, and display the input and output images. If you omit the second command-line argument, the default number of iterations is 1.

The Sobel program does Sobel edge detection. Optionally, it will also perform iterations of blurring prior to edge detection. A small amount of blurring tends to make edge detection more robust in images whose lines of contrast are not very sharp. If you run

```
java Sobel image.tiff 5
```

then Sobel will load the image from image.tiff, perform five iterations of blurring, perform Sobel edge detection on the blurred image, write the blurred image to a file named blur_image.tiff, write the grayscale-edge image to a file named edge_image.tiff, and display all three images (1 input, 2 output). If you omit the second command-line argument, no blurring is performed, and no blurred image is written nor displayed. (Sobel also has an optional third command-line argument that you shouldn't try until you complete Part III.)

We have included some .tiff files for you to play with. There is also a bit of test code for the boxBlur() and sobelEdges() methods in the main() method of PixImage.java.

You might find it useful to check out the methods in ImageUtils.java that read and write the TIFF files. We have also included a copy of the TIFF standard (the file TIFF6.pdf) for those who are curious. Neither of these things are necessary to complete the project, but if you want more control over writing images to files for your own entertainment, it is easy to modify Blur.java or Sobel.java for that purpose.

Part II: Converting a Run-Length Encoding to an Image
==========
This part is worth 25% of your total score. (5 points out of 20).

A large number of large image files can consume a lot of disk space. Some PixImages can be stored more compactly if we represent them as "run-length encodings." Imagine taking all the rows of pixels in the image, and connecting them into one long strip. Think of the pixels as being numbered thusly:

0	1	2	3
4	5	6	7
8	9	10	11

Some images have long strips of pixels of the same color (RGB intensities). In particular, the grayscale images produced by sobelEdges() can have large uniform regions, especially where no edges are detected. Run-length encoding is a technique in which a strip of identical consecutive pixels (possibly spanning several rows of the image) are represented as a single record or object. For instance, the following strip of intensities:

7	7	7	88	88	88	88	88	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11

could be represented with just three records, each representing one "run":

7,3	88,5	0,4

"7,3" means that there are three consecutive 7's, followed by "88,5" to signify five consecutive 88's, and then "0,4" for four jet black pixels. With this encoding, a huge image whose pixels are mostly one color (like daily comic strips, which are mostly white) can be stored in a small amount of memory. The TIFF image format has run-length encoding built in, so in these cases it can produce shorter image files. (Note that the TIFF encoder we'll be using only gives us a savings when there are runs of pixels for which all three colors are identical. However, it is also possible to write TIFF files in which the three colors are separated, so one can exploit runs within a single color.)

Your task is to implement a RunLengthEncoding class, which represents a run-length encoding as a linked list of "run" objects. It is up to you whether to use a singly- or doubly-linked list, but a doubly-linked list might make Part IV easier. (You are not permitted to use large arrays for this purpose.)

Because this is a data structures course, please use your own list class(es) or ones you have learned in class. In future courses, it will sometimes make more sense for you to use a linked list class written by somebody else, such as java.util.LinkedList. However, in CS 61B this is forbidden, because I want you to always understand every detail of how your data structures work. Likewise, you may not use java.util.Vector or other data structures libraries.

Part II(a): Implement two simple constructors for RunLengthEncodings. One constructs a run-length encoding of a jet black image. The other constructs a run-length encoding based on four arrays provided as parameters to the constructor. These arrays tell you exactly what runs your run-length encoding should contain, so you are simply converting arrays to a linked list. (Read the prototype in RunLengthEncoding.java.)

Part II(b): Your run-length encodings will be useful only if other classes are able to read your encodings after you create them. Therefore, implement the iterator() method in the RunLengthEncoding class and the RunIterator() constructor, the hasNext() method, and the next() method in the RunIterator class.

These methods work together to provide an interface by which other classes can read the runs in your run-length encoding, one by one. A calling application begins by using RunLengthEncoding.iterator() to create a new RunIterator i that points to the first run in the run-length encoding--the run that contains pixel (0, 0). (Outside classes should never call the RunIterator() constructor directly; only RunLengthEncoding.iterator() should do that.) Each time i.next() is invoked, it returns the next run--represented as an array of four ints--until every run has been returned. The returned array encodes a run by storing the length of the run (the number of pixels) at index zero, the red pixel intensity at index one, the green pixel intensity at index two, and the blue pixel intensity at index three. (This four-int array is constructed in next(), and can be discarded by the calling method after use. This array should not be part of your RunLengthEncoding data structure! It must be freshly constructed for the sole purpose of returning four ints.)

Given an iterator i, i.hasNext() returns true if it has more runs to return. It returns false if i has already been used to return every run, letting the calling program know that there are no more runs in the encoding. Calling programs should always check i.hasNext() before they call i.next(); if they call i.next() at the end of the encoding, i.next() may throw an exception and crash your program. (If a calling application wants to reset i so it points back to the first run again, it has to construct a brand new RunIterator.)

Please read the file RunIterator.java carefully for more information. You might also find it helpful to look up the Iterator class in the Java API.

Part II(c): Implement a toPixImage() method in the RunLengthEncoding class, which converts a run-length encoding to a PixImage object.

Read RunLengthEncoding.java carefully for an explanation of what methods you must write. The fields of the PixImage class MUST be private, and the RunLengthEncoding class cannot manipulate these fields directly. Hence, the toPixImage() method will rely upon the PixImage() constructor and the setPixel() method.

You cannot change any of the prototypes in RunLengthEncoding.java or RunIterator.java--except the RunIterator() constructor, which you can give any signature you like, as it is called only from RunLengthEncoding.iterator(). We will test your code by calling your public methods directly.

There is a bit of test code for Parts II, III, and IV in the main() method of RunLengthEncoding.java.

Part III: Converting an Image to a Run-Length Encoding

This part is worth 25% of your total score. (5 points out of 20).

Write a RunLengthEncoding constructor that takes a PixImage object as its sole parameter and converts it into a run-length encoding of the PixImage.

The fields of the PixImage class MUST be private, so the RunLengthEncoding constructor will rely upon the getWidth(), getHeight(), getRed(), getGreen(), and getBlue() methods.

Testing

The following is worth 1 point out of the 5, but should probably be done as soon as you can during Part III.

Your RunLengthEncoding implementation is required to have a check() method, which walks through your run-length encoding and checks its validity. Specifically, it should print a warning message if any of the following problems are found:

- If two consecutive runs have exactly the same type of contents. For instance, a "99,12" run followed by an "99,8" run is illegal, because they should have been consolidated into a single run of twenty 99's.
- If the sum of all the run lengths doesn't equal the size (in pixels) of the PixImage; i.e. its width times its height.
- If a run has a length less than 1.

You may find that the check() method is very useful in helping to debug your RunLengthEncoding() constructors and setPixel() in Part IV. I also recommend implementing a toString() method for your RunLengthEncoding so you can print and examine it to help debugging.

If the Sobel program is given three (or more) command-line arguments, regardless of what the third argument is, it will write the grayscale-edge image twice, once as an uncompressed TIFF file and once as a run-length encoded TIFF file. The two TIFF files will be different and have different lengths, but the images should look identical. If they don't, there is probably a bug in your run-length encoding method or your RunIterator.

You might find it interesting to compare the sizes of the two TIFF files. (Not surprisingly, the disparity is greatest for the input file black.tiff, which is all black pixels.)

Part IV: Changing a Pixel in a Run-Length Encoding

The last part is the hardest, but it is only worth 10% of the total score (2 points out of 20), so don't panic if you can't finish it.

Implement the setPixel() method of the RunLengthEncoding class, which is similar to the setPixel() method of the PixImage class. However, this code is much trickier to write. Observe that setPixel() can lengthen, or even shorten, an existing run-length encoding. To change a pixel in a run-length encoded image, you will need to find the right run in the linked list, and sometimes break it apart into two or three runs. If the changed pixel is adjacent to other pixels of identical color, you should consolidate runs to keep memory use down. (Your check() method ensures that your encoding is as compact as possible.)

IMPORTANT: For full points, your setPixel() method must run in time proportional to the number of runs in the encoding. You MAY NOT convert the run-length encoding to a PixImage object, change the pixel in the PixImage, and then convert back to a run-length encoding; not only is that much too slow, it will be considered CHEATING and punished accordingly.

Test Code

We are still working on an autograder for the project, and will provide it when it is ready. Until then, there is some test code in the main() methods of both PixImage and RunLengthEncoding, and the programs Blur and Sobel can also help.

The autograder will assign some, but not all, of the points for the project. Additional points will be assigned by a human reader for partly finished code and for your check() method, which is not autogradeable. Points may be subtracted if you break some of the rules stated above, especially the rules on where you must use arrays and where you must use linked lists.

Submitting your Solution

Make sure that your program compiles and runs on the _lab_ machines with the autograding program before you submit it. Change (cd) to your pj1 directory, which should contain PixImage.java, RunLengthEncoding.java, RunIterator.java, and any other .java files you wish to submit. If your implementation uses .java files in addition to those we have specified, have no fear: the "submit" program will ask you which .java files in your pj1 directory you want to submit. From your pj1 directory, type "submit pj1".

After submitting, if you realize your solution is flawed, you may fix it and submit again. You may submit as often as you like. Only the last version you submit will be graded, unless you inform your reader promptly that you would prefer to have an earlier submission graded instead.

If your submission is late, you will lose 1% of your earned score for every two hours (rounded up) your project is late.

Afterthought

Our own John Canny, a professor right here in the Berkeley CS department, invented an "optimal" edge detector, which is a more sophisticated version of what we've done in this project. If you're interested, see the Wikipedia page http://en.wikipedia.org/wiki/Canny_edge_detector .