HW 5: Image Processing

Dedicated to Efi Arazi (1937-2013), image processing pioneer,
and founding patron of IDC School of Computer Science.

Images are all around us, and so is image processing. Each time you move, rotate, shrink, or enlarge some image, you are using image processing software. You may be familiar with image editing programs like *Paint* or *Photoshop*. In this assignment you will learn and implement some of the key algorithms and techniques used in these programs.

Although there is no need to use any external image editing software in this assignment, we wish to note two programs. Gimp is a popular open-source alternative to PhotoShop, available freely but requires some installation effort. IrfanView is a free and light-weight image editor that can be easily downloaded and installed. Once again, you don't need any of these programs for this assignment, so this is just a general reference.

In the process of working on this assignment, you will learn a lot about working with multi-dimensional arrays, about using and writing functions, and about functions that process and return arrays.

You will also get acquainted with *image processing*, which is an exciting field of theory and practice. We hope that you will enjoy cutting your teeth into it. We begin with a brief digital imaging crash course.

Digital Imaging

Color

Color is a human perception of a natural phenomenon. Physically, color is a light wave that has a certain *wavelength* (analogous to *pitch* in audio). The part of the human brain known as *visual cortex* evolved to distinguish between about 10 million different wavelengths, and human languages have given some of these wavelengths names like “red”, “yellow”, “green”, “magenta”, and so on. When light waves hit the human eye, specialized sensor cells in the retina react to them, according to their wavelengths. Many of these sensor cells specialize in detecting different spectrums of wavelengths, which we call “red” (R), “green” (G), and “blue” (B).

All the colors that we are capable of seeing emerge from the way our brain mixes and combines different *intensities* (analogous to *volume* in audio) of those three basic colors. For example, "high red", combined with "medium green" and "a little blue", give light brown. You can experiment with RGB color mixing using this mixer tool.

The RGB system just described can be implemented digitally. We can view each color as a vector of three integer values, each ranging between 0 and 255, representing the *intensities* of the red, the green, and the blue basic colors. Here is an example of how the RGB system can be used to represent a few colors in Java:

```
int[] red =        {255,   0,   0};
int[] green =      {0  , 255,   0};
int[] blue =       {0  ,   0, 255};
int[] black =      {0  ,   0,   0};
int[] white =      {255, 255, 255};
int[] yellow =     {255, 255,   0};
int[] marineBlue = {0  , 102, 204};
// Etc.             most colors have no names; the next paragraph explains why.
```

Since each one of the three RGB codes ranges from 0 to 255, computers that use the RGB system can represent $256^3 = 16,777,216$ different colors. This is about 6 million more colors than the human brain can discern. Not bad.

Pixel

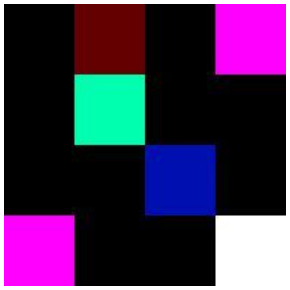
A *pixel* (shorthand for *picture element*) is the smallest controllable element of a digital image – "the atom" from which digital images are made. A pixel is characterized by (*i*,*j*) coordinates and a color.

Image

A digital image can be represented by a matrix of pixels: each (*i*,*j*) entry of the matrix represents a pixel. In an RGB color image, each pixel stores a 3-element vector: element 0 represents the *red* intensity, element 1 represents the *green* intensity and element 2 represents the *blue* intensity. We see that the entire image can be represented using a 3-dimensional, rectangular array, which we'll also refer to as the *image matrix.*

The image *resolution* is determined by the number of pixels. The more pixels, the sharper and more detailed the image. For example, a 720 x 480 RGB image is made up of 720 x 480 = 345,600 pixels, or 720 x 480 x 3 = 1,036,800 integer values. One obvious way to cut down the weight of this representation is to use the smallest possible integer data type that can accommodate 256 different values. In this project we will not worry about space efficiency though, and will use `int` values throughout.

To illustrate the approach, consider the following simple image:



Using the conventions discussed above, this image can be represented by the following matrix:

```
int[][][] tinypic = { {{  0,  0,  0}, {100,  0,  0}, {0,  0,  0}, {255,  0,255}},
                      {{  0,  0,  0}, {  0,255,175}, {0,  0,  0}, {  0,  0,  0}},
                      {{  0,  0,  0}, {  0,  0,  0}, {0, 15,175}, {  0,  0,  0}},
                      {{255,  0,  0}, {  0,  0,  0}, {0,  0,  0}, {255,255,255}} };
```

To illustrate, we see that the top-left pixel is black (0,0,0) and the bottom-right pixel is white (255,255,255).

In the images that we normally see on screens or printed on paper, the displayed physical area that each pixel occupies is very small. An image like `tinypic`, which is made up of 16 pixels only, is actually very tiny. We have blown it up by about 5000% before pasting it into this document.

Image file

In order to store images persistently, and transfer them over the network, we have to decide how to represent images using text files. There are many different agreed-upon file formats for doing just that, including JPEG, GIF, PNG, BMP, and PPM, each having its pros and cons, each suitable for certain purposes. In order to cut down storage and communications costs, most of these formats store images in a compressed form.

In this assignment we use the PPM file format. PPM is an uncompressed format, so it's easy to work with. It is recognized by many image editors, including *Gimp* and *IrfanView*. The PPM file of the `tinypic` image is as follows:

```
P3
4 4
255
  0   0   0   100   0   0     0   0   0   255   0 255
  0   0   0     0 255 175     0   0   0     0   0   0
  0   0   0     0   0   0     0  15 175     0   0   0
255   0 255     0   0   0     0   0   0   255 255 255
```

The first three lines are called the *file header*. `P3` is an agreed-upon code that describes the type of the image being stored. Let us not worry about it, remembering to always set it to `P3`. Next comes the number of columns and the number of rows in the image (in this example, 4 x 4). Next, we list the maximum value appearing in the file, which is 255 in all the images used in this project. Then comes the body of the file, which is a list of triplets, each being an integer between 0 and the maximum value (which happens to be 255). White space is used liberally to make the data more readable to humans, and is skipped by the software. Following convention, we list each row of pixels in a separate line.

The Assignment

In this assignment you will gradually build a library of image editing functions. You will also write client code for testing and playing with these functions, for your pleasure and entertainment. Before doing anything though, read this entire document, from beginning to end. There is no need to understand everything you read; this understanding will grow on you as you start working on the code.

All the editing functions that you have to write must be added to the supplied `ImageOps.java` class. Presently, this class contains one function only, named `show`. This function takes a 3-dimensional integer array as a parameter, and renders the image that the array represents on the screen, using the services of the `StdDraw` class. For now, there is no need to understand the `show` function code, just use it as needed.

Testing: The assignment requires writing a family of image processing and helper functions. It is absolutely crucial to test each function as soon as you finish writing it. For example, here is a typical testing code segment:

```
public static void main(String[] args) {
    // *** Testing the reading of an image from a file
    // Reads image data from a file, into an array
    int[][][] pic = read("tinypic.ppm");

    // Displays the image matrix (the function is described below)
    showData(pic);

    // Displays the image
    show(pic);
}
```

Write and execute similar tests for each function that you write. If you want, you can factor these tests into several stand-alone helper functions, and call these test functions from your `main` function.

We now turn to specify all the functions that you have to write.

Part I

Reading an image file

The function `int[][][] read(String fileName)` receives the name of a PPM file, and returns a three-dimensional array containing the image data. The input file must be located in the project folder.

Implementation tips: use `StdIn` to read the image data from the given file. To get started, your code must make a call to the function `StdIn.setInput(String filename)`. This function informs all the `StdIn` functions that, until further notice, standard input will come from the file `fileName`. You can assume that this file contains valid PPM code. Note: `setInput` is a new function that we added to the `StdIn` class that was given to you. Therefore, it is not documented in the typical `StdIn` class API pages found in the Internet.

Write the `read` function, and test it before continuing to do anything else in this assignment. Note that as far as `StdIn` is concerned, the input file is simply a text file. `StdIn` knows nothing about PPM, so it is up to your `read` function to use `StdIn` functions to get and parse the data, and then build the image matrix from it.

Displaying the image data

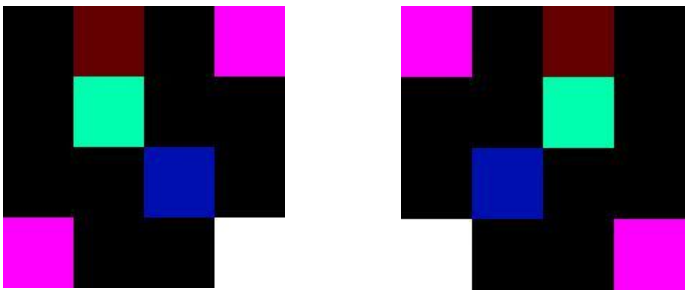
The function `void showData(int[][][] pic)` receives a 3-dimensional array representing an image, and writes the contents of this array. Use the `printf` statement to print the integer values in a way which is easy to read and understand. For example, if we call the `showData` function with the matrix that represents the `tinypic` image, we should get the following output:

```
  0   0   0   100   0   0     0   0   0   255   0 255
  0   0   0     0 255 175     0   0   0     0   0   0
  0   0   0     0   0   0     0  15 175     0   0   0
255   0 255     0   0   0     0   0   0   255 255 255
```

The `showData` function is designed to help you test your code. It provides a more direct view of the image than the `show` function. Thus, it is an essential debugging tool in this assignment.

Horizontal Flipping

The following example illustrates what is meant by "flipping an image horizontally":



A note about side effects: In general, we don't want image processing functions to change the original image matrices that they are called to process. In some functions in this project we allow ourselves to do so, since it makes the programming easier.

The function `void flipHorizontally(int[][][] pic)` flips a given image matrix, horizontally. This is done by reversing the order of the pixels is each row of the matrix (within each pixel, of course, the colors remain fixed).

Here is an example of applying the `flipHorizontally` function to the `tinypic` image:

`tinypic`: (source)

```
  0   0   0   100   0   0     0   0   0   255   0 255
  0   0   0     0 255 175     0   0   0     0   0   0
  0   0   0     0   0   0     0  15 175     0   0   0
255   0 255     0   0   0     0   0   0   255 255 255
```

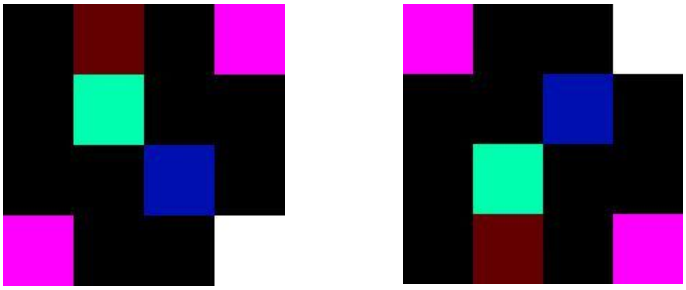
horizontally flipped:

```
255   0 255     0   0   0   100   0   0     0   0   0
  0   0   0     0   0   0     0 255 175     0   0   0
  0   0   0     0  15 175     0   0   0     0   0   0
255 255 255     0   0   0     0   0   0   255   0 255
```

Start by writing the helper function `void swap(int[][][] pic, int i1, int j1, int i2, int j2)`. This function takes two given pixels (i_1,j_1) and (i_2,j_2) in a given image matrix, and swaps them. After completing and testing this helper function, proceed to write and test the `horizontallyFlipped` function.

Vertical Flipping

The following example illustrates what is meant by "flipping an image vertically":



The function description is very similar to the previous function. Instead of reversing the order of elements in each row, we reverse the order of the rows:

`tinypic`: (source)

```
  0   0   0   100   0   0     0   0   0   255   0 255
  0   0   0     0 255 175     0   0   0     0   0   0
  0   0   0     0   0   0     0  15 175     0   0   0
255   0 255     0   0   0     0   0   0   255 255 255
```

vertically flipped:

```
255   0 255     0   0   0     0   0   0   255 255 255
  0   0   0     0   0   0     0  15 175     0   0   0
  0   0   0     0 255 175     0   0   0     0   0   0
  0   0   0   100   0   0     0   0   0   255   0 255
```

Write and test the `flipVertically` function.

Greyscale

The RGB system has a convenient property: when all the three color intensities are the same, the resulting color is a shade of grey, ranging from black (0,0,0) to white (255,255,255). The 256 possible values are called "greyscale codes". With that in mind, "greyscaling" is a technique for transforming a colored image into an image containing only shades of grey, while doing it in a way which is pleasant and sensible to the human eye. Below is an example of a colored image and its greyscaled version:

How to transform a 3-way RGB value representing some color into a single greyscale value representing a shade of grey that corresponds to that color? Suppose that the RGB values (each being a number from 0 to 255) are represented by the variables `r`, `g` and `b`. We define *luminance* to be the following linear combination:

```
luminance(r,g,b) = (int) (0.299 * r + 0.587 * g + 0.114 * b)
```

Since the luminance weights are positive, and sum up to 1, and since the intensities are all integers between 0 and 255, the computed luminance is also an integer between 0 and 255. Therefore, if we choose to represent a greyscale image using a matrix of 3-value vectors, we can set every pixel to `{lum,lum,lum}` (the value of `luminance`, repeated 3 times). This is quite a wasteful representation for greyscale images, but it's easy to work with it in the context of this project, so that's what we'll use.

We note in passing that the three weights 0.299, 0.587, and 0.114, which are commonly used in greyscaling, are based on the human eye's relative sensitivity to red, green, and blue. The specific weights were determined after running many experiments with human subjects.

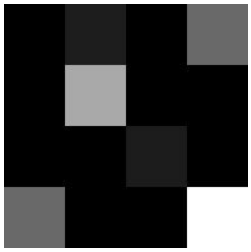
We divide the greyscaling implementation into two stages. First, write a function `int[] luminance(int[] color)` which takes an RGB color as input, and returns the corresponding greyscale value, using the formula presented above. As usual, test the function before going on. For example, the function should transform the color `{255,0,0}` (which happens to be red) into the greyscale value `{76,76,76}`.

Next, write the function `int[][][] greyScaled(int[][][] pic)`. This function takes an RGB image as input and returns a greyscaled version of this image. Needless to say, this function makes use of the luminance function described above.

For example, here is the greyscaled image of `tinypic`:

```
{{{  0,   0,   0}, { 29,  29,  29}, {  0,   0,   0}, {105, 105, 105}},
 {{  0,   0,   0}, {169, 169, 169}, {  0,   0,   0}, {  0,   0,   0}},
 {{  0,   0,   0}, {  0,   0,   0}, { 28,  28,  28}, {  0,   0,   0}},
 {{105, 105, 105}, {  0,   0,   0}, {  0,   0,   0}, {255, 255, 255}}}
```

And here is the image itself:



Client Editor

We now describe a client program that makes use of the three image editing functions described above. The `BasicEditor.java` program takes two command-line arguments: the name of a PPM file, followed by an optional second command-line argument which is either `fh`, `fv`, or `gr`. The program reads the image from the file. If there is no second command-line argument, the program opens a graphical window and displays the image as is. Otherwise, the program displays the image horizontally flipped, vertically flipped, or greyscaled, according to the second command-line argument.

Write and test the `BasicEditor.java` program.

Part II

Blurring

The term "blurring" is often used to describe the act of softening the contrast of an image. This image processing operation has many applications, especially when applied to selective areas of an image. For example, portraits of human faces often appear more pleasing when the skin and the background are blurred, which makes other features of the image stand out. When applied to an entire image, the blurring operation makes it look out of focus. For example:

We will implement an algorithm called *box blurring*, which operates as follows. For each pixel *p* of the image and for each one of the three pixel colors, calculate the average color of all the adjacent pixels (immediate neighbours) of *p*, including *p* itself, and then assign that color to *p*. You can think about the operation as a 3x3 box traveling around the image; in each step, the average color of the pixels inside the box is calculated, and then assigned to the middle pixel. The pixels in the boundaries of the matrix are treated differently, since they are surrounded by less than 9 neighbouring pixels.

For example (we use color coding to illustrate the averaging operation for three sample pixels):

`tinypic`:
(the source)

```
  0   0   0   100   0   0     0   0   0   255   0 255
  0   0   0     0 255 175     0   0   0     0   0   0
  0   0   0     0   0   0     0  15 175     0   0   0
255   0 255     0   0   0     0   0   0   255 255 255
```

blurred:

```
 25, 63, 43    16  42  29    59  42  71    63   0   63
 16  42  29    11  30  38    39  30  67    42   2   71
 42  42  71    28  30  67    28  58  67    42  45   71
 63,  0, 63    42   2  71    42  45  71    63  67  107
```

Explanation of the colored examples: (0+100+0+0)/4=25, (0+255+0+0+0+15+0+0+0)/9=30, (0+0+0+0+0+255)/6=42. The averages are cast as `int` values, resulting in the integer part of each average.

The function `int[][][] blurred (int[][][] pic)` takes as input an image matrix, and returns a blurred version of the corresponding digital image.

Implementation tip: To deal with the pixels in the matrix boundaries, you may want to work with an enlarged matrix, padded with zeros. Start by reading the documentation of, and implementing, the two helper functions `getColorIntensity` and `blurColor`. Then write the `blurred` function.

To test your implementation, write a `BlurringEditor.java` program that takes two command-line arguments: a PPM file name, followed by a number `N` which represents the number of successive blur operations that the program will execute. The program reads the source PPM file using the `ImageOps.read` function, and enters a loop of `N` steps. In each iteration, the function calls the `ImageOps.blurred` function on the previously computed image, and displays the result using the `ImageOps.show` function.

Edge Detection

The last part of the assignment introduces an important technique known as *feature detection*. In particular, you will implement an algorithm designed to detect edges in an image. Edge detection is a fundamental capability without which we would not be able to detect objects in scenes, and companies like MobilEye would not be able to identify obstacles in the car's path. Intuitively, an edge is the boundary of an object inside an image. For example, here is an example of some exotic Indian picture, and a greyscale image representing its edges:

We will implement a *Gradient Edge Detection* algorithm, designed to detect vertical edges. The *gradient* measures the rate of change in an image (similar to a derivative in calculus). When an edge appears in a greyscaled image, we should expect to find a sharp change in the greyscale values around the edge, and so the gradient will be high. When no edge exists, the greyscale values should be more or less the same, so the gradient will be low.

Formally, let $p_{i,j}$ be the greyscale value of pixel i,j. The horizontal gradient at that location is defined as $g_{i,j} = p_{i,j+1} - p_{i,j-1}$, the difference between the greyscale values of the right and left pixel neighbours.

We will implement the edge detection operation using several functions, which we now turn to describe.

Gradient calculation: The function `int[][] horizontalGradient (int[][][] pic)` returns a two-dimensional array with the horizontal gradient calculated at each pixel. The source image (`pic`) is a 3-dimensional array, representing greyscale values. Since the horizontal gradient is undefined at the left and right boundaries of the image (nor can we have any edges there), we'll set these gradient values to 0.

To illustrate, below is horizontal gradient of the greyscaled version of `tinypic`:

`tinypic`: (greyscaled)

```
  0   0   0    29  29  29     0   0   0   105 105 105
  0   0   0   169 169 169     0   0   0     0   0   0
  0   0   0     0   0   0    28  28  28     0   0   0
105 105 105     0   0   0     0   0   0   255 255 255
```

horizontal gradient:

```
  0           0          76           0
  0           0        -169           0
  0          28           0           0
  0        -105         255           0
```

Write and test the `horizontalGradient` function.

Normalizing: The gradient is a measure of an edge strength at a specific location. These measures can be synthesized "back" into an image matrix, using a *normalization* operation. The term "normalization" is used to describe an operation that takes values in one range and maps them on values in another range. In our case, we wish to map all the values in the horizontal gradient matrix to greyscale values between 0 and 255. The normalization is done as follows. Let v_{min} and v_{max} be the minimum and maximum values in a given matrix. The normalized value of $g_{i,j}$, the matrix entry in the (i,j) coordinate, is the value of $(g_{i,j} - v_{min}) \cdot \frac{v_{max}}{v_{max} - v_{min}}$. For example, in the horizontal gradient matrix of `tinypic`, the normalization formula will be $(g_{i,j} - (-169) \cdot \frac{255}{255 - (-169)} = (g_{i,j} + 169) \cdot \frac{255}{424}$, yielding the following normalized matrix:

horizontal gradient (normalized)

```
  0   101   147   0
  0   101     0   0
  0   118   101   0
  0    38   255   0
```

The function `void normalize(int[][] arr)` receives a two-dimensional array, and normalizes all its values to integer values between 0 to 255. Write and test the `normalize` function.

Edge Detection: We are now ready to put everything together. The function `int[][][] edges (int[][][] pic)` receives an image matrix. First, it transforms the image into greyscale. Then, it calculates its normalized gradient. Finally, it returns a 3-dimensional greyscale image matrix, whose values are the normalized gradients of the source image. Write and test the `edges` function.

Blurring to decay: To demonstrate your work, write the program `BlurToDecay.java`. The program takes two command line arguments: the name of an input PPM file representing an image, and a number `N` representing a number of steps. At each step, the program uses the `ImageOps.edges` function on the current image, and displays it to the screen using the `ImageOps.show` function. Next, the program blurs the image using the `ImageOps.blur` function. If everything works well, the program should give an animated view of how the edges of a given image gradually disappear, until they decay completely (if `N` is sufficiently large).

Submission: Before submitting the assignment, take some time to inspect your code, and check that your functions are short and precise. If you find some repeated code, consider factoring it into a separate helper function. Make sure your program is written according to our **Java Coding Style Guidelines**. Also, make sure that each program starts with the program header described in the **Homework Submission Guidelines**. Any deviations from these guidelines will result in points penalty.

Submit these files only which should include all functions:

- `ImageOps.java`
- `BasicEditor.java`
- `BlurringEditor.java`
- `BlurToDecay.java`

Deadline: Submit your assignment no later than Thursday, December 6, 23:55.