1 Assignment Goals

The apparent goal of this assignment is to create an efficient set of modifications to image files using image filters. Additionally the assignment can be used to review basic functions and debugging within the IDE that the student is using for eclipse. My personal goals were to familiarize myself with JAVA as a language again and then to create efficient functions with efficient runtime complexity. I did this by focusing on three proper aspects of my implementation: loop structure, data types, and the provided API. These themes underly the attempt specified and written for this assignment.

2 Solution Specifications

The solutions used within this code came in two different approaches. The first six functions used a quick modification structure to allow bitwise operators to do the heavy lifting instead of expressed functions, as in this this was a faster approach. The next approach was the half loops used for the Horizontal and Vertical Reflection Functions. The rest of the functions used minimal modifications to these two approaches.

2.1 Bitwise Functions

These functions include NoRed, NoGreen, NoBlue, OnlyRed, OnlyGreen, OnlyBlue. Essentially on an integer loaded with an ARGB value a simple operation to edit the bit String can be done. For example using bitwise & and the inverse of 0xffff<<16 (Note that the first ff is to conserve the transparency value alpha). This can be modified for all other functions in the above list. Thus all that remains is to traverse every pixel in the array. This leads to a runtime complexity of $O(n)$ where $n$ is the number of pixels. It should also be noted for irregular arrays of pictures that do not take a rectangular structure the implementation as it stands for these functions should work.

2.2 Half Loops

The functions that used this method were HorizontalReflection and VerticalReflection (Karma adds both Peices4 and Rotate180). These functions only Traverse half of the loop in favor of performing a computation (in this case swap) twice in an efficient manner. Normally there would exist and edge case for odd numbers where odd numbers would have to be traversed, however in such Reflection functions the odd number would be left intact and thus it would be unnecessary to read the central point. While the implementation still leaves the runtime complexity at $O(n)$ it means that the whole array need not be read and can save some time. The assumptions for bot these functions are a rectangular structure but both VerticalReflection and HorizontalReflection
can be used on non rectangular structures but will result in odd transformations, that are not reflected upon a central line but instead the center of each row and column, respectively.

2.3 Other Assigned Functions

The other functions each were their own individual variation on the basic double for loop structure for iterating 2-D arrays:

- **BlackAndWhite**: Simply averages the RGB values and then uses that average for each of the RGB values, conserving the Alpha value.

- **Grow**: This function assumes that the image is rectangular. Before the loop structure initializes an empty array twice the width and height of the original. Then it maps each pixel in the original to its 4 locations in the larger array. Runtime of $O(n)$ where $n$ is the number of pixels.

- **Shrink**: The function initializes an empty array half the width and the height of the original. Then in the loop structure it average 4 pixels RGB values and maps them to one pixel in the smaller array. Runtime $O(n)$ where $n$ is the number of pixels (bigger or smaller array).

- **Threshold**: The function checks the threshold value to measure if the each of the pixel’s RGB values is above the threshold. If so it is set to 0xff the max value for a pixel otherwise it is set to 0.

3 Completion and Karma

3.1 Encountered Problems

Initially the project seemed very straight forward. The first six functions were implemented with no problem although I had an odd conjecture about for loops in Java that I later investigated for Karma. When I reached **BlackAndWhite** I forgot for about half an hour how colors are constructed by computers. I remembered a lecture from high school and then implemented a function that simply ensured each color was just as present as the next.

I promptly tested this solution for several images and found that it worked. The edge cases for **Shrink** provided another challenge it was easy to resolve but required a significant number of if then statements to check for the edge case of one of the array lengths being one. Thus **Shrink** can only work on rectangular shapes.

Other edge cases that initially startled me during tests were easily the odd numbered edge case for half-loops. These required a special statement at the end of the loop accounting for them. However I still would like to note that the relative speed was marginally reduced to account for accuracy. Although the traversal may have been slower, than the original it still significantly beats the complexity of introducing.

For Karma another method called **InsertOrder** which was used in the implementation of **DeNoise** was especially difficult to write as I attempted to overcomplicate the method. I forgot to place within the for loop the break condition, and it seemed unnecessary to put it within the loop rather than have it as a condition. After reviewing one of the book Chapters I realized that the middle of a for loop is simply a condition and this issue was resolved (Further details in 3.2).
3.2 Convolutional Karma

This Section refers directly to Karma that I implemented, which was given on the assignment sheet. This includes two methods explicitly stated: Smooth and DeNoise. And finally the Transparency method, which cause every single bitwise transformation to be rewritten to include a force on for the first 8 bits, in order to preserve the alpha value.

For the Directly assigned Karma the implementation is rather simple first a nested for loop structure allows the programmer to traverse each pixel. Then a for loop is prepared to traverse the neighborhood (specified by the user). An if statement is used to make sure the row being examined for the neighborhood exists. This can save a great deal of time as an inner for loop can be skipped. The last inner for loop is used to help preform the transformation on the pixel using each element of the neighborhood. Before method specific implementations are discussed it is important to note that although the user is given the ability to choose the neighborhood a 5x5 grid is suggested as such a neighborhood viewing takes \( O(n \times k^2) \) where \( n \) is the number of pixels and \( k \) is the length (or width of the neighborhood). One pixel already takes 25 times as many checks/computations to run on this methodology (assuming 5x5. Thus in order to keep this value very small it is important to suggest a small value. However, 3 was not chosen because it lacks does not perform the convolution over a large enough area to be instantly recognizable.

Smooth was implemented quite simply using a summation on the value of each part of the pixels in the neighborhood (aside from \( \alpha \)). Then the value used to divide the whole figure was incremented. At the end of the neighborhood loops the sum was divided by the counter \( \text{div} \) and then the results were put back into an int and then placed in the location of the unmodified pixel. DeNoise required a more involved approach. Although it stems from the same loop structure as Smooth, it has an extra operation as one must find the median of the neighborhood. Thus there is a real need for a sorted array of values. Thus a sorting method can be implemented but it bumps the complexity by an extra \( k^2 \). Thus the implementation is rather bulky but the most effective way I could conceive of to sort the array simultaneously. The method called to fulfill this feature is insertOrder the advantage of the separate method is that it can be called three times for each array of pixels. Note the conservation of the \( \alpha \) term.

3.3 Extra Karma

Extra Karma is divided into two sections: Extra Implementations and Experiments. Extra Implementations include methods that were added, experiments include classes run repeatedly to determine which set of code was faster at what times after the rest of the coding had been finished.

3.3.1 Extra Implementations

The Extra functions implemented were quirks upon what had already been used before, however they were interesting to make. The Pieces4 is a horizontal and vertical reflect combined. It was interesting to implement because it was supposed to become a Rotate180, but it was interesting what it did to certain images like 11.jpg. It’s basic implementation is two half loops to iterate over the array, at each iteration a single swap is made. Rotate180 is the same except that ensures there are two swaps and catches the odd number edge cases as well.

The final implemented function is Transparency this function adds \( \alpha \) to the RGB, enabling the transparency of the image to be controlled by the user. This implementation required a series of edits to the ImageEffect class. The first of which was to change the pixelsToImage function to load ARGB integers when saving to the file. The next step was to create a method for grabbing the
\(\alpha\) value from the pixel, this was done using bitshifting (\texttt{getAlpha}). This prompted a new \texttt{makePixel} method including \(\alpha\) as a component. After looking on Piazza for information on modifying the given files the old interface for \texttt{makePixel} was also required and implemented with the \(\alpha\) value being set automatically to 255. This means if the deprecated interface were to be called it would reset the transparency of the image. Additionally every method that used the original interfaced was given a modification to include the \(\alpha\) value, whether it be changing \texttt{OnlyGreen} by changing the \texttt{modification from 0xff<<8 to 0xff00ff<<8}. This preserves the \(\alpha\) value while ensuring that only the green part of the pixel is shown. For more complicated functions the new \texttt{makePixel} interface replaces the old and usually the \(\alpha\) value is conserved by keeping it to match the original.

3.3.2 Experiments

Because I wanted to focus on runtime at the start of this assignment, as I went through it I found more and more that I was interested in the quirks of the byte code I was working with. Included in the sent package are two classes that run for loops and while loops in weird combinations and find the amount of time in nanoseconds of difference. It seems that in Java for example runs faster than a while loop of the same length. This implies that the Bytecode or VM has a way of implementing for loops faster. While no rigorous experiment was carried out it is an entertaining notion that I examined. This code is stored in the \texttt{proj1Experi} folder within the zip file.

4 Software Test Methodology

To be perfectly honest the only consistent software testing was done on \texttt{11.jpg} as it happened to be the image that was the most recognizable. However \texttt{7.jpg} was used extensively to test edge cases for the Shrink method in order to ensure no errors were thrown. While most code got tested on almost all the images it was only a test checked by the naked eye and thus lacks a certain level of regularity. Some way to implement that testing is to use JUnit in order to fully create and regulate test cases that would feed in 2-D arrays that were not rectangular or that contained a very different set of information and then run all possible images within a specified grid. This is however tedious and rather time consuming and leads to a lack of appreciation for what the code actually does.

However two more interesting images were used one is that of a TARDIS, because I love Doctor Who but more importantly the picture was significantly wider so \texttt{Smooth} and \texttt{DeNoise} could be seen with a smaller effect on image quality, plus it looks nice to see it as it seems like the TARDIS is phasing away. The other image I used was the \texttt{Optical.png}. This image just looked really nice and also made \texttt{Invert}, and the No and Only functions seem to have more of a purpose.