Lab 9: Huff(man)ing and Puffing

Lab: Due: Wed. 11/13 at 11PM (for Mon. lab), Thurs. 11/14 at 5PM (for Mon. evening), or Thurs. 11/14 at 11 (for Tues.)

Lab Report Due: Beginning of class on Monday, 11/18

You are once again encouraged to work in pairs.

The number of bits required to encode an image for digital storage or transmission can be quite large. Consumer quality digital cameras take pictures that are 4608 pixels wide and 3456 pixels tall or larger. Such an image contains a total of 15,925,248 pixels or just about 16 megapixels. Each pixel is represented by three 8-bit numbers encoding its redness, greenness and blueness. In raw form, therefore, it would take 382,205,952 bits to represent such an image. On a 100 megabit/second Ethernet it would take nearly 4 seconds to transmit such an image without even accounting for collisions or other overhead.

To make it possible to transmit images more quickly and to store large numbers of images on computer disks and camera memory cards, considerable effort has been devoted to devising techniques for compressing the information in digital images. File formats like GIF, JPEG, and PNG represent the implementations of some of the compression techniques that have been developed.

To help you appreciate both how one might compress an image and how difficult it is to achieve high levels of compression, we want you to implement components of several compression techniques and then evaluate their effectiveness. Note: The “evaluate” aspect is a new feature of this lab. In addition to writing a program this week, we actually want you to write a lab report summarizing the data you collect using the program you will write. These reports will be collected in class after the programs are completed.

As we did last week, we have not included an implementation plan in this lab handout. We want you to bring a full implementation plan with you to your lab period and then to follow that implementation plan as you complete the lab. Plan carefully: We will not provide an implementation plan for this lab.

Image Simplification

It is possible to use Huffman codes to compress image data. To do this, one treats the 256 values that can appear in a pixel array as the letters of an alphabet containing 256 symbols. Based on Huffman’s algorithm, short codewords are then assigned to the brightness values that appear frequently and longer codewords are assigned to the values that appear less frequently. Then, the table of pixel values are all translated from their original 8-bit binary codes to the Huffman codewords that had been assigned, and the entire list of codewords is saved or transmitted.¹

Unfortunately, using Huffman codes to compress image data in this way is not very effective. Huffman codes exploit the fact that some symbols in a message occur more frequently than others. The more extreme the differences between the frequencies with which symbols occur in the data, the greater the degree of compression a Huffman code will provide. The frequencies with which brightness values occur within an image tend to be too uniform for effective Huffman coding.

Consider the image shown on the right.

¹ In addition, one would have to encode the Huffman tree describing the code used and the width and height of the image. Since this information would require relatively few bits compared to those used to represent the pixel array values, we will not account for the cost of encoding it in this lab.
The graph shown on the left is a histogram of the frequencies with which various brightness values appear in this image. The histogram ranges from brightness 0 to 255. There is a significant peak around 200 and a smaller peak between 50 and 60. Within the range of 150 values between these peaks, the histogram is rather flat. Huffman coding cannot do significantly better than a fixed length code when applied to such data. In fact, a Huffman encoding of the brightness values in this image would require 7.94 bits per pixel, less than 1% better than the obvious fixed length code. Fortunately, there are techniques that enable Huffman coding to work far more effectively.

For example, suppose we process small blocks of pixels by leaving the value in the upper left corner unchanged and replacing each of the other brightness values with the difference between the original value and the value found in the upper left corner. That is, if we started with the 2x2 block of pixels:

\[
\begin{array}{cc}
A & B \\
C & D \\
\end{array}
\]

We would replace its contents with the values:

\[
\begin{array}{cc}
A & B - A \\
C - A & D - A \\
\end{array}
\]

In most images, the shades of pixels that are adjacent are very similar. Therefore the differences we compute while performing this transformation are likely to be small values. These pixels account for 75% of the pixels in the transformed image. Accordingly, this transformation will change the distribution of values in the pixel array significantly. A large number of the pixel values will be close to 0.

Unfortunately, this also means that a significant number of the values we computed will be negative and therefore fall outside the original range of pixels. We can minimize the number of such out-of-range values by adding 128 to each difference, centering the new values in the range of existing values.

The result of applying this transformation to all 2x2 blocks in the image presented earlier is shown on the right. Since one quarter of the pixel values are unchanged, the original image remains visible. The changed pixels give the new image a dull gray look.

Note that we can recover the pixel values of the original image from the transformed image. If we process each block within the transformed image by adding the value in the upper left corner to all of the other values and subtracting 128, the original image will reappear.

The histogram of the transformed image is shown at the top of the next page. The distribution of brightness values is no longer uniform. Instead, there is a significant peak at 128, reflecting the fact that most of the differences computed while making the transformation were near 0. Huffman coding this collection of pixel values is therefore much more effective.

The average number of bits per pixel is roughly
7, reflecting a 12% savings overall. As a result, this transformation provides a way to compress an image for transmission. We first apply the transformation. Then we Huffman encode the resulting pixel values and transmit them. When this transmission is received, the receiving computer can first decode the Huffman codewords to restore the brightness values of the transformed image. Finally, the corner values can be added to the other values in each block to restore the original image.

The savings obtained using this technique will vary from image to image depending on how the brightness values in each image are distributed. Changing from 2 x 2 blocks to larger blocks would also affect the savings. As a result, the only way to really evaluate such a technique is to try several variants on a selection of typical (and atypical) images and analyze the results. For this lab, we want you to conduct such an experiment comparing several possible transformations.

The process of replacing most of the pixels in a block with their differences from the pixel in the corner of the block is just one way one might transform an image to increase the effectiveness of Huffman coding. Any transformation that will a) lead to a less uniformly distributed set of values in the image’s encoding and b) provide the means to restore the original brightness values (or something that closely approximates them) can be used. We will call such a transformation an image simplification.

For this lab, we want you to implement portions of four image simplification algorithms described below (Note: We do not want you to implement the approach we just described!). In addition, we want you to implement an algorithm that computes the bits per pixel required to encode a set of brightness values using a Huffman code. Then, you will use these tools together to evaluate the effectiveness of five image simplification algorithms including the four you have implemented.

The Algorithms

**Range Reduction**

The first algorithm we want you to evaluate is the simple technique of reducing the number of brightness values used to encode the image as you did in the AdjustLevels program you implemented during lab 7. This does not actually change the shape of the histogram associated with an image, but by reducing the number of distinct values used, it makes it possible to encode the values with fewer bits. On the other hand, given the reduced number of brightness values, it is not possible to restore the original image exactly. As long as the number of brightness values used in the transformed image is not too small, however, the transformed image will be a close approximation of the original image. Such a transformation is said to be lossy. By contrast, the block differencing simplification described in our introduction is said to be lossless.

We will provide you with a completed implementation of this transformation in the starter project for the lab. We include this simplification for two reasons. First, when you are collecting data on the simplification schemes you implement, it can serve as a baseline. Second, just as we had you implement the AdjustLevels class by extending your ImageViewer class in lab 7, in this week’s lab, you will implement simplification schemes by extending an ImageSimplifier class provided in the starter project. The implementation of RangeSimplifier included in the starter project will serve as an example of how to define simplifiers by extending ImageSimplifier.
Study this class carefully so that you understand it well enough to emulate it in your own code!

**Waterfall**
The first simplification scheme we want you to implement yourself involves computing the difference between pixel values much like the scheme described in the introduction. The idea is very simple. Leave the brightness values along the topmost row of each of the three pixel arrays that describe the image unchanged. Replace each of the other brightness value with the difference between its original value and the original value of the pixel directly above it.

The name of this algorithm comes from the process used to restore the original image given this collection of transformed values. Starting at the top of each column you will add the first pixel value (which will be unchanged) to the value below it (which will be a difference). The sum of these two values will be the original value of the lower pixel. You then repeat this process “falling” down from the top of the column of pixels to the bottom. When you are done, all of the pixel values will be restored to those of the original image.

The waterfall algorithm should be implemented by defining a `WaterfallSimplifier` class that extends the `ImageSimplifier` class included in the project starter folder. Its implementation should mimic the `RangeSimplifier` class that we have also included in the starter.

**Wavelet**
While the waterfall algorithm processes an image’s pixels in pairs from top to bottom, the wavelet simplification process works with pairs from left to right. The leftmost pixel in each pair is replaced by the average of the two values in the pair and the rightmost pixel is replaced by half of the difference. For example, given the pair:

```
A  B
```

We would replace its values with:

```
(A + B)/2  (A - B)/2
```

Rather than leave the transformed values next to one another, the wavelet transformation moves all of the averages toward the left side of the image and all of the differences to the right. Thus, if a single row of an image that was eight pixels wide contained the values:

```
A  B  C  D  E  F  G  H
```

then after the transformation was complete the values stored in the row would be:

```
(A+B)/2  (C+D)/2  (E+F)/2  (G+H)/2  (A-B)/2  (C-D)/2  (E-F)/2  (G-H)/2
```

Of course, if an image’s width is odd, there will be one pixel that has no partner to form a pair with. We will handle this by simply placing the value of the last pixel in such a row between the averages and the differences. That is, given a row like:

```
A  B  C  D  E  F  G  H  I
```

the wavelet simplifier will produce the transformed row:

```
(A+B)/2  (C+D)/2  (E+F)/2  (G+H)/2  I  (A-B)/2  (C-D)/2  (E-F)/2  (G-H)/2
```

---

2 We do this intentionally so that later you can combine this with the waterfall method.

4
The resulting image will look like a horizontally contracted copy of the original with a similarly sized dark region to its right. The result of applying this transformation to the image we have been using as our example is shown below on the right. If you look carefully, you can see that the dark rectangle actually contains bright regions that correspond to the edges of objects in the original image. The edges are where the differences between adjacent pixels tend to be largest.

The procedure for restoring the original image given the values produced by the wavelet transformation is simple. If the initial values of two adjacent pixels were A and B, then the values stored for these pixels in the transformed image will be \( S = (A+B)/2 \) and \( T = (A-B)/2 \). If you evaluate the expression \( S + T \), the result will be A. If you evaluate \( S - T \), the result will be B.

Actually, while this is true in normal mathematics, in Java, things won’t quite work out. Since we will be working with integer values, when we compute \( (A+B)/2 \), Java will throw away any remainder from the division. As a result, \( S + T \) and \( S - T \) will only produce close approximations of A and B. Wavelet simplification is therefore another example of a lossy approach.

We will provide you with partial code for the `WaveletSimplifier` class in the starter folder. This `WaveletSimplifier` class includes code to transform a normal image into a simplified version. Your job will be to write code to restore an approximation of the original image given the simplified version.

**Recursive Wavelet**

If you look at the sample image shown above to illustrate the result of applying the wavelet transformation, you should notice that while it is a bit squished, the left half of that image looks a lot like a normal picture. Wavelet is a transformation designed to process pictures. Suppose we applied it again to just the left half of the image shown above. The result would look like the image on the right. The left quarter of the image is a very compressed version of the original. The right half of the image is the difference values from the original image. The dark quarter between these two is a collection of difference values from the first compressed version of the image. As such, it is very close to a compressed version of the right half of the image.

One thing is obvious. More of the pixels in this image are nearly black. Therefore, it will have an even bigger peak in its histogram and should compress better. Of course, if it is good to apply wavelet twice, it must be better to do it three times, or four, or...

The third technique we would like you to evaluate is a recursive version of wavelet. It will begin by applying the simple version of wavelet described above to an image. Then, if the original image is wider than two pixels, it should extract the left half of the result as a new image and recursively apply itself to
the result. Finally, it should paste the result of this recursive call back into the left half of the image from which it extracted the left half. The result should look like the image to the right.

Again, we will provide you with partial code for this simplification process in our starter folder. The class named `RecWaveletSimplifier` includes complete code to simplify an image in this way, including working code to perform the cut in half and paste operations. All you have to add is a recursive method to restore an approximation of an original image given a simplified version. We emphasize this method should use recursion rather than loops of any kind.

While you have written recursive code before, the implementation of this transformation will illustrate a slightly different form of recursion than you encountered in Lab 6. There will be no recursive class involved. That is, you won’t define a class that has an instance variable that refers to another instance of the same class. Instead, you will simply define an image processing method within a class that invokes itself on a smaller image. As a result, there will also be no empty boolean to tell you when to stop recursing. Instead, as suggested above, this recursive process will terminate when the image has been reduced to a single column.

The Kitchen Sink
Well, if waterfall is good and wavelet is good, what if we did both? We deliberately described wavelet working left to right and waterfall working top to bottom so that this would be appropriate. For your last simplifier, implement a class that first applies the recursive wavelet transformation to an image and then applies waterfall to the result. When you are all done, all but one pixel in the result will be a difference value. To reverse the process, simply apply the decoding transformations in the opposite order. That is, first apply the waterfall unsimplifier and then the recursive wavelet unsimplifier.

Huffman Code Size Computation
In class, we discussed a procedure for computing the cost of a Huffman code without actually building the Huffman tree. In particular, we described an algorithm that returns the total size (in bits) of the encoded document. We would like you to implement this algorithm as part of this lab.

You will implement this algorithm within a new method named `getHuffmanSize` in the `Histogram` class. This is because the array used in the `Histogram` class to store counts of the number of pixels for each of the brightness values is exactly the input needed by Huffman’s algorithm --- a set of weights or counts.

Computing the Huffman cost requires repeatedly finding the two smallest remaining elements in a `weights` array, merging them by adding their values together, and adding their combined value to a total that will eventually equal the number of bits required to encode the message described by the `weights`. To implement the algorithm, you’ll want to do the following:

- Copy all of the non-zero entries in the histogram array into contiguous positions in a new `weights` array. The `weights` array should be created to have the same size as the histogram array, but if there are zero entries in the histogram array, the `weights` array will have some unused entries at the end.

- Maintain a variable named `remaining` whose value will equal the number of entries in the `weights` array that are still in use. Initially, the value of `remaining` should equal the number of non-zero elements found in the histogram array. Its value will be reduced by one with each iteration of the main loop of your Huffman algorithm implementation.
• Define an auxiliary method named `findMin`. `findMin` should return the index of a minimum value in `weights` stored between index 0 and index `remaining-1`.
• Initialize a variable that will keep track of the total number of bits required by setting it to 0.

The main loop in your implementation of the Huffman cost algorithm will then repeatedly:
• Use `findMin` to locate a minimal value among the elements remaining in the `weights` array,
• Save the value of this minimal element,
• Effectively remove the minimal element from the `weights` array by moving the value stored at position `remaining-1` into the position that held the minimal value and then decrementing `remaining`.
• Use `findMin` again to locate a minimal value among the remaining elements.
• Add the saved minimum value to the minimum of the remaining elements, replace the minimum of the remaining elements with this sum and add this sum to the total number of bits required.

In the AllImages folder we provided for Lab 7 you will find an image file designed to help you test this algorithm. The file is named “HuffmanTest.png”. The image uses only 5 distinct shades of gray. The ratios of the colors used in this image are 4/13, 4/13, 3/13, 1/13, and 1/13. Given these numbers you should be able to perform Huffman’s algorithm by hand to verify that the number you produce by hand is the same as that produced by your code.

**Using the Starter Project**
We want you to incorporate your implementations of the simplification algorithms described above into a program that will allow you to compare their behavior by systematically applying the algorithms to a variety of images. The interface for this program will resemble the interface of the program you wrote last week in many ways. Rather than having you adapt the code you wrote over the last two weeks to this new purpose, we will provide a starter file containing versions of the `ImageViewer`, `DisplayDifference`, `DisplayHistograms`, and `Histogram` classes that should be quite similar to the ones you created. In addition, our starter project will contain complete implementations of the `RangeSimplifier` described above and `SimplifierDriver`, a class that extends `ImageViewer` and provides the GUI components needed to select and apply a simplifier to an image. Finally, as explained above, we have included partial implementations of the `WaveletSimplifier` and `RecWaveletSimplifier` classes. During the lab, you will have to add two additional classes to implement the waterfall simplifier and the combined waterfall and recursive wavelet transformation.

A picture of the interface the `SimplifierDriver` class provides is shown on the right. Since the class extends `ImageViewer`, the functions of the buttons on the top of the window should be familiar. Two additional controls are provided at the bottom of the window. The menu on the left displays entries for all of the types of simplifiers available. The range reduction simplifier is the only one for which a
complete implementation is provided in our starter project. As you complete the other simplifiers described above, you will add entries for those simplifiers. Once a simplifier has been selected and an image loaded, pressing the “Simplify” button causes two new windows to be created displaying a simplified version of the original image and an image obtained by reversing the simplification process. The picture on the next page shows how these windows might look together.

The partially obscured window on the left is the original program window. The two others are the windows created when “Simplify” is pressed. (The windows do not usually appear so nicely spaced out on the screen. You may have to rearrange them on your computer’s display.) The window in the middle is labeled “Simplified” and displays a simplified version of the original (in this case darkened by dividing all brightness values by a constant). The window on the right is labeled “Restored” and was produced by brightening the image (multiplying all pixel values by the same constant).

After these windows have been displayed, you can use them to evaluate the effectiveness of the simplification scheme. Once you have implemented the algorithm for computing the cost of a Huffman code, pressing the “Show Histograms” button on the “Simplified” window will show how many bits per pixel are required to compress the image in its simplified form. For lossy compression schemes, the “Restored” window lets you evaluate the impact of the distortion introduced in two ways. First, you can visually inspect the restored image. Second, the “Restored” window is initialized so that it displays the restored image but also holds the original image. As a result, by pressing the “Show Difference” button you can examine the differences between the two images very precisely.

More About the Other Classes Provided

ImageViewer:
We provide an implementation of the ImageViewer class missing two of the buttons you implemented last week, “Zoom in” and “Zoom out”.

Histogram Classes:
Histogram and DisplayHistograms are similar to what you used last week except that:
• **DisplayHistograms** now tries to display the average bits per pixel when the data that generates such a histogram is compressed using Huffman’s algorithm (but it will not succeed until you complete the implementation of the **getHuffmanSize** method).

• **Histogram** is now designed to correctly count brightness values even if some of them are negative or greater than 256. This is important since the waterfall and wavelet algorithms both need to store negative differences in some cases.

• The **Histogram** class includes an incomplete method to calculate the Huffman cost. You will have to complete this method during the lab.

**DisplayDifference:**
This class should be equivalent to what you wrote last week.

**RangeSimplifier:**
This class simplifies an image by dividing all its pixel values by a constant and approximately restores the simplified image by multiplying by the same constant. As explained above, it is included mainly to provide an example of how the **ImageSimplifier** class can be extended.

**ImageSimplifier:**
This class is designed to make it simple to define image simplifiers by extending its definition. All classes that extend **ImageSimplifier** should define a **toString** method that returns a string describing the simplification technique implemented. This string will be used to identify the method in the menu displayed at the bottom of a **SimplifierDriver** window.

The waterfall and wavelet simplification algorithms can be implemented by defining classes that extend **ImageSimplifier** and override the **encodePixels** and **decodePixels** methods. Each of these methods is passed an array of pixel values describing one color layer of an image to be processed. The **encodePixels** method should apply the desired simplification transformation. The **decodePixels** method should attempt to restore a good approximation or an exact copy of the original pixel array.

One advantage of implementing all four algorithms by extending **ImageSimplifier** is that this makes it possible for the **SimplifierDriver** class to treat all four of these image transforming objects as interchangeable. In particular, once you have implemented a new simplifier, all you will need to do to incorporate it into the rest of the program is add an object of the class you have defined to the **JComboBox** created in the **SimplifierDriver** constructor. This is the only change you should have to make to **SimplifierDriver** during the lab.

**Getting Started**
Checkout the starter code from your account on evolene.cs.williams.edu. If you are working with a partner, share your repository with your partner as a maintainer using the web interface to GitLab on evolene.

**Submitting your Code**
Make sure to take a final look through your code, checking its correctness and style. Check over the style guide accessible through the course web page and make sure you have followed its guidelines. Submit by simply committing and pushing your final version to evolene.

You can find instructions describing how to submit your program on the “Labs” page of our web site at http://www.cs.williams.edu/~cs134/Labs.html

**Report and Experiments**
In addition to the source code for a working program, you will also submit a typed lab report. This lab report need not be long (Limit yourself to 2 pages!), however it needs to be clear and address the questions listed in this section. Use good formatting, grammar and spelling.
Because this is a technical document, it is important to be precise, using mathematics and data tables to support your findings.

The centerpiece of the report will be your data table. It should list the compression ratios that you observed for each compression method on a small sample of the images found in the Compression folder of the AllImages folder and the subjective quality of the compressed image. The images you should use are butterfly.png, contrast.png, gradient.png, mandrill.png, noise.png, and sunset.png. Note any artifacts (distortions) that you observe in the restored images. Recall that pressing the Show Difference button in the restored image window will display an image that reflects the exact difference between the original and restored images.

Specifically (but briefly) address the following questions and topics in the report:

- What kinds of images compress well? What kinds of images compress poorly?
- Which of the algorithms you implements appear to provide the best compression?

Even if you are unable to complete the code for the lab (or have a few errors left), you should still submit the report. In this case, use the sample solution application we will provide through the course web page.

**Extra! Extra!**

A few of you have shown unending enthusiasm for adding extra features to your programs. This week’s lab and its associated report may be enough to keep you busy. In particular we strongly suggest that you complete your lab report before undertaking any extensions. However, just in case some of you somehow find time to do more, here are a few ideas for extensions. We have provided a variety of ideas in the expectation that you will just pick one or two.

**Composing Simplifiers**

For our Kitchen Sink simplifier you joined together two existing simplifiers, the Waterfall and recursive Wavelet simplifiers. You could certainly imagine other combinations. For example, you could combine the Range Reducer with Waterfall. One way to make this easy is to write a class whose constructor takes two existing simplifiers as parameters and constructs a new simplifier that combines these two existing simplifiers. Like your other simplifiers, this class should extend the ImageSimplifier class. It should be possible to implement this class by overriding just the filter and toString methods.

**Video Compression?**

No! We don’t think you should try to implement video compression. You could, however, fairly easily conduct a little experiment that might help you appreciate one important approach used in video compression.

Think about the differences between adjacent frames of a video. What would they look like if displayed as images?

You should by now be able to predict that these differences would consist mainly of pixel values with very small absolute values. You could verify this by using the code in the DisplayDifference class to constantly display the difference images computed between consecutive frames as you use the computer’s camera to capture video frames. In fact, you don’t even need to display the video (you know what you look like!). Just display the differences in real time. Do not try to pop up a new DisplayDifference window for each pair of images! Modify DisplayDifference so that the class that creates a DisplayDifference window can later invoke a method to update the images whose difference is displayed.

Since we know that it is really the shape of the histograms of the color layers that determine how effective techniques like Huffman Coding will be, another interesting option would be to display the histograms of the differences between adjacent frames of video in real time rather than displaying the difference images.
Leaky Waterfall
There is an interesting variation on the Waterfall simplifier.

In the version of Waterfall we asked you to implement, the difference between a pixel and the pixel above it was stored exactly. Suppose that instead, you saved small differences exactly but approximated larger differences. For example, you might store any difference whose absolute value was less than 5 exactly and round down any larger difference to the nearest multiple of 5 so that the difference values reported were members of the set \{-255, -250, \ldots -15, -10, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 15, \ldots 250, 255\}.

If this is the only change you make, this simplification process will be very lossy. Suppose, for example, that the values stored in one column in one pixel array of some very short image’s brightness values are 45, 52, 59, 66, 73, 80, 87, 94. (Notice that to keep things simple, we have made all of the brightness differences 7.) The normal Waterfall algorithm would replace all but the top pixel with the difference between itself and the pixel above it leading to a column containing the values 45, 7, 7, 7, 7, 7, 7. The approximation scheme we just suggested would have to round each of these 7s to 5 yielding a column of values 45, 5, 5, 5, 5, 5, 5. Each difference is only off by two. However, when we restore the image, the errors would accumulate. The calculated value for the first pixel below the top would be 50 instead of 52, the next pixel would be set to 55 instead of 59, etc. The restored column would look like 45, 50, 55, 60, 65, 70, 75, 80. The final pixel would be off by 14, even though no difference was ever off by more than 2.

The problem is that each difference we stored in our column was based on the difference between the actual value of the pixel above rather than on the approximated value that will be computed for that pixel when we un-simplify the image using the approximate differences. We can improve things greatly if the differences we send are calculated relative to the slightly inaccurate value that we can predict the receiver will compute using our approximate differences. That is, as before, we would start by subtracting 45 from 51 to obtain a difference of 7 and round that difference to 5. Knowing that the approximate difference recorded is 5, we can predict that when this pixel is restored it will incorrectly be set to 50 rather than 52. So, for the next pixel down, we will compute the difference between the next pixel, 59, and the value the pixel above will have in the restored image, 50. This leads to a larger difference, 9, but since we are still rounding down to the nearest multiple of 5, we will still record a difference of 5. As a result, we know the restored value of the next pixel will be 55 rather than 59. Now, however, things get better. We will subtract the value that will be the value we will restore for the last pixel, 55, from the value of the fourth pixel down, 66. This will lead to a difference of 11 which we will round to 10. As a result, the error for this pixel in the restored image will shrink to 1 rather than growing. Following this procedure for the entire column, the differences we report will be 45, 5, 5, 10, 5, 10, 5, 5, and the restored pixel values will be 45, 50, 55, 65, 70, 80, 85, 90. The pixels are still inaccurate, but the error will never be greater than the difference we introduce in approximating a single difference, 4, which will be imperceptible to the eye.

Lossless Wavelet
Just as it is possible to make Waterfall lossy, we can make Wavelet lossless. The loss in Wavelet comes from the divisions in the computations performed to store the average brightness of adjacent pixels, \((A-B)/2\) and half their difference, \((A-B)/2\). Since in Java, the division operation with integers returns an integer, the calculation of these values may be off by 1/2.

The simple way to make Wavelet lossless is therefore not to divide by two. Just store \(A+B\) and \(A-B\). Unfortunately, if this done, the range of values produced in each of these computations may become twice as large as the range from which the original values of A and B came. For non-recursive Wavelet, this is not an issue. When Wavelet is applied recursively, however, the range of the sums will double so frequently that the numbers computed will span a very large range.

The trick to avoiding the increasing range of values is to note that \((A+B)/2\) will only be an approximation when \(A+B\) is odd and this can only happen when exactly one of \(A\) and \(B\) are odd and the other is even. As a result, whenever \(A+B\) is odd, \(A-B\) will also be odd. So, revise your program to store \((A+B)/2\) and \((A-B)\). That is, approximate the sum of \(A\) and \(B\) by storing their average, but keep the exact difference. Then, when restoring an image, check to see if the difference stored for a pair of pixels is odd. If it is, you know that that value of \((A+B)/2\) is off by 0.5 and you can fix it!