Lab 9: Huff(man)ing and Puffing

Due November 18/19 (Implementation plans due 11/16-17, reports due 11/23)

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 3664 pixels wide and 2742 pixels tall or larger. Such an image contains a total of 10,046,688 pixels or just about 10 megapixels. Each pixel is represented by three 8-bit numbers encoding it redness, greenness and blueness. In raw form, therefore, it would take 241,120,512 bits to represent such an image. On a 10 megabit/second Ethernet it would take over 20 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 implementation of some of the compression techniques that have been developed.

To help you appreciate both how one might go about compressing an image and how difficult it is to achieve high levels of compression, we would like 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 Java program this week, we actually want you to write a lab report summarizing the data you collect using the program you have written. These reports will be collected in class after the programs are completed.

Finally, we want you to bring a completed implementation plan with you to your lab period.

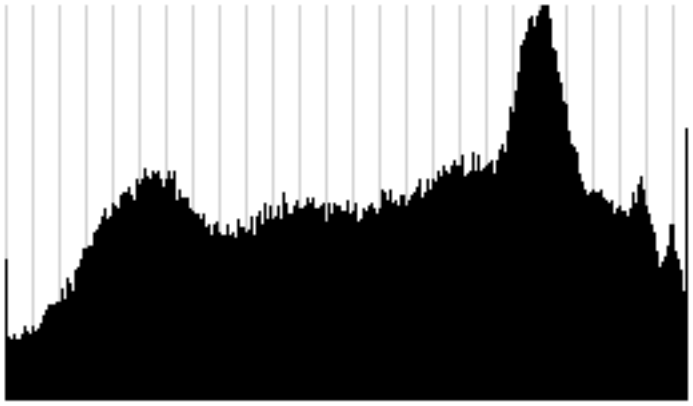
Image Simplification

Earlier in the semester, we discussed the use of Huffman codes to reduce the number of bits required to encode a text message. It is possible to use Huffman codes to compress image data. To do this, one would treat 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 would be assigned to the brightness values that appeared frequently in a pixel array and longer codewords would be assigned to the values that appeared less frequently. Then, the table of pixel values would all be translated from their original 8-bit binary codes to the Huffman codewords that had been assigned and the entire list of codewords would be 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. The distribution of these brightness values is uniform. 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% less than the obvious fixed length code. Fortunately, there are techniques we can use to encode the brightness values of an image that enable Huffman coding to work far more effectively. As an example, consider the following transformation.

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 in the block with the difference between the original value and the value found in the upper left corner. For example, if we started with the 2x2 block of pixels:

A	B
C	D

We would replace its contents with the values:

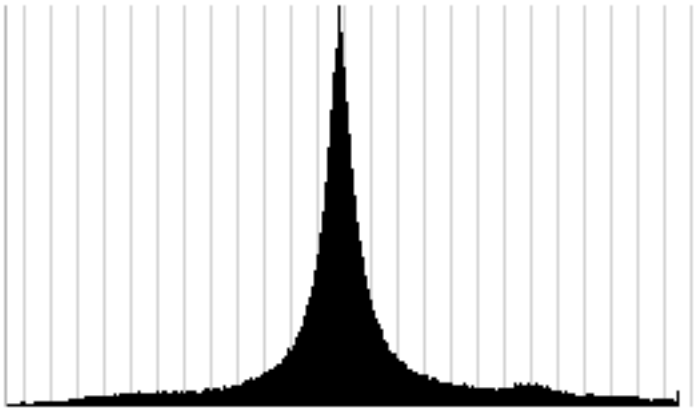
A	B - A
C - A	D - A

The pixels that are changed by this transformation are likely to have fairly small values. 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 significant 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 all of the pixel values from the original image given just the transformed image. If we process each block within the transformed image by adding the value in the corner of the block to all of the other values, the original image will reappear.



The histogram of the transformed image is shown on the right. 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 the 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 effect the saving. 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.

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 uniform set of values in the image's encoding and b) provide the means to restore the original brightness value 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. 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 "Quantizer" you implemented during lab 7. Pixels that had different but similar values in the original image will be represented by a single value after this transformation is applied. 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 used, 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 not actually make you implement this transformation again. Instead we will provide you with a completed implementation in a 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 Quantizer class by extending your `ImageFilter` class in

labs 7 and 8, 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 examples of how you should define your simplifiers by extending `ImageSimplifier`.

Waterfall

The first simplification scheme we want you to implement yourself involves computing the difference between pixel values much like the block corners 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 every 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 value 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 pixels 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 `RangeSimplier` 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 the values:

(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

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 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 E = (A+B)/2 and F = (A-B)/2. If you evaluate the expression E + F, the result will be the value of A. If you evaluate E - F, the result will be B.

Actually, while our claims about E, F, A, and B are 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, E + F and E - F 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 for this lab. This `WaveletSimplifier` class includes complete code to transform a normal image into a simplified version. Your job will be to write the 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 even a 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.

You should now recognize the purpose of two of the operations we had you implement last week. The code that implements the "Cut in Half" button is what you need to extract the left half of an image that has had the wavelet transformation applied to it once so that you can recursively process this half of the image. In addition, the code that implements the "Paste Image" button is just what you need to insert the result of the recursive application of the algorithm to restore an image that has been "waveletized" back in as the left half of the image to be returned.

Again, we will provide you with partial code for this simplification process in our starter folder. The class named 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.

While you have written recursive code before, the implementation of this transformation will illustrate a slightly different from of recursion. 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 top to bottom and waterfall working left to right so that this would be possible. For your last simplifier, implement a class that first applies the recursive wavelet transformation to an image and then applies wavelet 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 reversing transformations in the opposite order. That is, first apply the waterfall unsimplifier and then the wavelet unsimplifier.

Huffman Code Size Computation

This week's homework focuses on computing the cost of a Huffman code without actually building the Huffman tree. In particular, we describe an algorithm that returns the total size (in bits) of the encoded document. In the case of this lab, our document is an image and our encoded document is a compressed image. We would like you to implement a version of the algorithm in the homework as part of this lab.

You will implement this algorithm within a new method named `huffmanSize` 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. To implement the algorithm, you'll want to do the following:

- Copy all of the non-zero entries in the histogram array into a new `weights` array. The `weights` array should be created to have the same size as the histogram array.
- Because computing the Huffman cost involves working with decreasingly smaller lists of weights, we will maintain a variable called `remaining` that reflects the current length of our array of weights.
- Computing the Huffman cost involves both finding the position of and extracting a minimum value from an array of weights. It's easy to find the position of a minimum value, but to extract the minimum, while keeping the remaining elements in the array contiguous you'll need to use a little trick. Suppose your weight array contains 10 weights (i.e., the value of `remaining` is 10) and the minimum weight appears at index 3. The idea is to move the last weight (the one at index 9) to index 3 and then later dec-

rement `remaining`. This means the array still has its original length, but by decrementing `remaining` your program can keep track of the fact that only 9 remaining values are of interest.

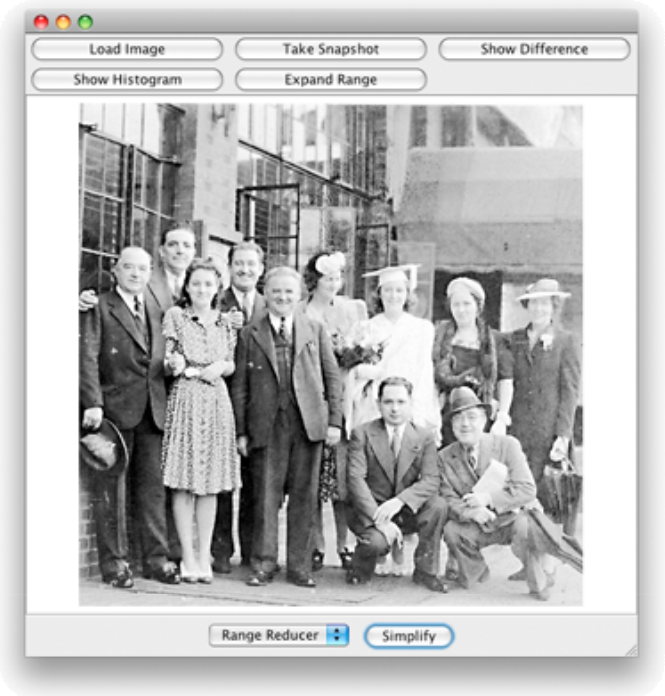
- You will want to define two auxiliary methods named `findMin` and `extractMin`. `findMin` takes an array of integers `weights` and the integer `remaining` and returns the *index of a minimum value* in `weights` stored between index 0 and index `remaining`-1. `extractMin` takes an array of integers `weights`, and the integer `remaining` and returns an integer representing the minimum weight stored between index 0 and index `remaining`-1. `extractMin` should call `findMin` as a helper method. In addition, `extractMin` performs the little trick above to place the minimum element in position `remaining`-1. Notice that `extractMin` does not itself decrement `remaining` --- you'll have to do that yourself after calling `extractMin`.
- `huffmanSize` should compute the Huffman cost by repeatedly calling `extractMin` until only a single weight remains in the array of weights. As should be clear by now, `huffmanSize` returns an `int`.

On the Labs page of the course website, we will provide an additional 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.

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 are complete (and 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 `RecWaveletSimplier` 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" was 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 display 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 Differences" button you can examine the differences between the two images very precisely.

Other Classes Provided

ImageViewer:
We provide an implementation of the ImageViewer class missing just two of the buttons you implemented last week, "Cut in Half" and "Paste".

Histogram Classes:
`Histogram` and `DisplayHistorgams` 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.
- 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 case.
- 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 that describes 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 (or perfect) approximation of the original pixel array.

The combined wavelet and waterfall algorithms will also be implemented by extending ImageSimplifier, but in this case, the new class should only override the encode and decode methods . This is a bit peculiar. The subclasses defined in this way will actually not use any code inherited from the ImageSimplifier superclass (the encodePixels, decodePixels, encodeLayer, and decodeLayer methods will be inherited unchanged, but they will not be used by the new encode and decode methods).

There is an advantage to implementing all four algorithms by extending ImageViewer. This approach 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
Download the starter file `Lab9Starter.zip` from the course website and unpack it in your `Documents` folder. Rename the folder using a name that contains “Lab9”, your name, and no blanks. You may also want to grab a copy of the HuffmanTest.png image while you are on the web page.

Report and Experiments
In addition to the source code for a working program, you will also submit a written lab report. This lab report need not be long (2-3 pages will be sufficient if you are concise), however it needs to be clear and address the questions listed in this section. As with any piece of writing, use clear formatting and follow normal English grammar and spelling.

Because this is a technical document, it is important to be precise, using mathematics, data tables, and diagrams to support your claims. It is also important to be objective. In arguing the merits of one compression method over another you must stick strictly to the facts. However, you are welcome to speculate as long as your speculation is clearly delineated and follows a clear line of reasoning.

The centerpiece of the report will be your data table. It should list the compression ratios that you observed for each 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. Select a subset of the images that includes examples of images with high contrast, images with areas of near uniform color, and images with

few such areas. Note any artifacts (distortions) that you observe. You are invited but not required to construct your own test images that may reveal weaknesses and strengths of the algorithms.

Specifically address the following questions and topics in the report:

- Which algorithm is best?
- Describe other test images that would be good for testing the properties and limitations of simplification algorithms.
- What kinds of images compress well? What kinds of images compress poorly?
- Sketch out a new image simplification algorithms (you don't have to write code; just describe an idea). How do you think this will perform compared to the ones you experimented on?

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, either discuss only the algorithms that you completed, or borrow a friend's completed implementation.

Submitting Your Work

As usual, make sure you include your name and lab section in a comment in each class definition. Find the folder for your project. Its names should be something like `FloydLab9`.

- Click on the Desktop, then go to the "Go" menu and "Connect to Server."
- Type "cortland" in for the Server Address and click "Connect."
- Select Guest, then click "Connect."
- Select the volume "Courses" to mount and then click "OK." (and then click "OK" again)
- A Finder window will appear where you should double-click on "cs134",
- Drag your project's folder into either "Dropoff-Monday" or "Dropoff-Tuesday".

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. The Mac will not let you submit again unless you change the name of your folder slightly. Just add something to the folder name (like the word "revised") and the re-submission will work fine.

Grading

This labs will be graded on the following scale:

++	An absolutely fantastic submission of the sort that will only come along a few times during the semester.
+	A submission that exceeds our standard expectation for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.
✓+	A submission that satisfies all the requirements for the assignment --- a job well done.
✓	A submission that meets the requirements for the assignment, possibly with a few small problems.
✓-	A submission that has problems serious enough to fall short of the requirements for the assignment.
-	A submission that is significantly incomplete, but nonetheless shows some effort and understanding.
--	A submission that shows little effort and does not represent passing work.

Completeness / Correctness

- The waterfall algorithm is correctly implemented
- The wavelet algorithm is correctly implemented
- The recursive wavelet algorithm is correctly implemented
- The combined waterfall and recursive wavelet algorithm is correctly implemented
- The `huffmanSize` method is correctly implemented

Style

- Commenting
- Good variable names
- Good, consistent formatting
- Correct use of instance variables and local variables
- Good use of blank lines
- Uses names for constants