Steven Lee
Lab 3 Analysis
Huffman Encoding Trees

First, I will describe and justify each class use. Main reads in data from text files, constructs each necessary class, then writes the final output to output.txt. In main, I used a queue data structure with a linked implementation to read in the input from ClearText.txt and Encoded.txt. Queues were good here because, since I was required to write my own test cases, I was not sure what the number of test cases would be. A linked implementation of the queue made sure that I would not overflow the array, and since I did not need to perform fast sorting or searching on the input, arrays would not provide those types of benefits in this case. Finally, a linked implementation made sure that I did not waste space. The queue was written in the Queue class. For FreqTable.txt input, I used an array. In this case, I knew the length of the array was going to be the length of the alphabet, and I thought I needed to sort the array as well by frequency. For this reason, I needed the quicksort algorithm, which works with an array implementation.

The FreqTable class sorts the freqtable.txt alphabet using the quicksort algorithm. This was helpful so that I could see the actual frequencies in ascending order, and build more complicated trees later. Additionally, it handled the strings of the form '(char) - (frequency)' and split them up into char and int arrays so that they could be used by the HuffTree class, to be described later.

The Node class was necessary for implementing linked data structures, but it did not itself implement any data structures. In that class, I had both a 'next' variable and a 'left' and 'right' variable. In this way, I could use the same node class for both lists and trees, depending on my need in the particular structure. Java method overriding was useful here, as different types of nodes could be created depending on the parameters passed to the node.

The PriorityQueue class was used for constructing the Huffman tree. In this case, I again used a linked data structure. A linked structure was good for a few reasons. First, the queue is constantly changing size, as elements are popped, then re-inserted into the queue. Additionally, items were pushed onto the queue by priority, which meant that they needed to be inserted at any arbitrary location. A linked structure allows insertion as an O(1) operation, unlike the slower insertion operation for the middle of an array. PriorityQueue also stored several operations important for making the huffman tree, such as tie breaking for making sure that I had the correct insertions throughout the building process. The main drawback is that searching for the correct location in a singly linked implementation is O(n/2). I think searching may have been possible to improve in an array implementation because I would have been able to jump ahead more than one element at a time while searching for the correct place to do the insertion. With a linked implementation, I had to visit every node until I found the right one.

The HuffTree class constructs the Huffman encoding tree by building a binary tree using a priority queue. This works by passing the frequency table into the HuffTree class, and the Huffman tree is built, bottom up, by repeatedly combining and reinserting nodes into the priority queue until there is only one item in the queue, which is the root of the Huffman encoding tree. This works by using the PriorityQueue class earlier described. While there is more than item in the queue, the top two items are popped, combined with a parent node, and reinserted into the queue until the Huffman Encoding tree is made.

The EncodeDecode class encodes clear text, and decodes encoded text. Using the HuffTree class, EncodeDecode uses queues to handle the variable amount of input coming from the encoded and decoded test cases. After that, arrays are used for all other variable handling. This class implements the encoding by swapping each character for its code according to the Huffman tree, then saving that string to an array. The decoding works by traversing the binary tree left or right according to whether it reads a 0 or a 1, until it reaches a node with a single character for its data, which is then appended to the final string.

The Metrics class measures the time taken for each operation, encoding and decoding, and provides individual as well as summary statistics. It is helpful to have a separate class for these measurement operations because they do not contribute to the computation of the final answer, so any functions called from them should be in a class separate from what they are measuring.

In terms of efficiency with respect to time and space, there are a few considerations. I performed each read operation in O(N) time, with respect to the input. This is the fastest possible read, as any faster performance would mean that some characters were not read. I needed to use quicksort on the frequency table data, as well as each node which was greater than a single character in order to exactly match the final table according to the requirements. This data was never entirely ordered, nor entirely disordered, meaning that average case performance is the best predictor, which is O(nlog(n)). The slowest parts of my program run in O(n log(n)) time, which happens in the encoding and decoding of the string as well. This is because I need to iterate through an entire string, which is O(n), then I need to search a binary tree for each character, which is O(log(n)) on average. For space efficiency, I use multiple arrays in order to perform sorts, and store output, but I do not use any algorithms which intentionally use extra space.

With respect to the handout questions, I do think that I have achieved useful data compression in this method. Each of the strings which are input can be broken down into arrays of characters. Each character is a base 10 integer, which can then be converted into a binary string. To note, the binary string version of each letter is shorter than the ASCII character converted to binary, with the exception of X and Q. However, these are the most uncommon letters, while the most common letters are half the size of their ASCII equivalents. If I used a different scheme to break ties, I would expect the amount of compression to go down. This is what happened when I coded the Huffman tree incorrectly, and this resulted in all characters

being longer. For example, my second attempt at ‘E’ ended with a code ‘0010’, which was less compressed than the correct character ‘010’. In my first attempt, I did not use a priority queue, but used a queue instead. While this achieved greater compression with ‘E’, at ‘0’, all other letters were significantly longer, with Q being 26 binary ‘characters’ long.

In this lab, I learned a lot about how important thinking through the project is before coding. I thought I was pretty good at doing that, since lab 2 was faster, but I should have thought even more about implementation for lab 3 prior to actually coding. I took a few wrong directions in actually solving the problem while coding, which was a waste of time and effort. If I had solved the problem on paper prior to solving the problem in a program, I think I would have realized that my first few attempts would not work. Additionally, I only figured out the solution once I made a Huffman encoding tree for 5 letters, rather than 26. That helped me see where I was going wrong better than the 26 letter variety. Finally, I learned that being able do dump the contents of any data structure in its entirety is very important for debugging. The priority queue did not make very much sense until I printed it every time I popped the contents, then I could see where certain tie breakers were failing, causing me to not print ‘HELLO WORLD’ correctly. Next time, I will make sure to solve the problem on paper first, then solve a smaller version in code, then solve the actual problem. I think that will help with not wasting time, even though it creates higher overhead.

As enhancements, I provide a few measurements to output.txt. First, I print out the encodings for each letter, with E as ‘010’ as an example, so that you can easily see what the Huffman codes are. I found this helpful when I was comparing my output to the sample encoded string for hello world in the assignment. I also provide individual time metrics for each input, along with summary statistics in a formatted string. I think that this is helpful for seeing the differences in encoding compared to decoding. If you disregard the first input, which always seems to take significantly longer, then encoding and decoding tend to have the same time on average. This makes sense, since they both have O(N logN) costs to converting them.