So for the phrase “I like donuts”, the words themselves have 11 characters. This, stored in character form, would take up 8x11 bits, or 88 bits. The decode has 49 binary bits. This is a small savings. Of course, for a compression format like this, a phrase that consists of more common letters will be more greatly compressed than a phrase that contains less common letters like Q or Z. Overall, this will lead to a small improvement in file size, as the only letters that are equal to the size of their translated codes is Q and X. All the others have less bits in them than a char would need.

However, if we were to expand the frequency table to include more than just uppercase letters, our savings would soon disappear. This is because the tree gets orders of magnitude larger every time something is added, and eventually the cost savings would become negligible, or even negative. However, there could still be some savings obtained from compressing just the files with the most common letters/words in them.

Huffman encoding, compared to ciphers, offers a more interesting way of decoding. Ciphers are not very safe, since pattern analysis could easily lead to insights that allow the cipher to be cracked. This could be pattern recognition, or just randomly trying different cipher combinations and lucking out on one that works. For Huffman encoding, there is no pattern recognition. It would be hard to decode if you did not have the key, as you do not know how the frequency table was built, or how far the nodes extend.

If I had ordered the values differently, there were many times where the result I would have gotten was different. Even just playing with the string file in the Readme on a Huffman generator website, I could see where my program and theirs differed. Mine program put two one-letter combinations together that had the same frequency. On that step, however, the website put a one-letter combination with a 3 letter combination that had the same frequency. Had I followed the website, it would have resulted in gibberish coming out of the decodes and encodes.

For structures, I had to use a combination of a queue and a binary tree to create the Huffman tree. The queue was made so that I could easily do an insertion sort every time I had to make a new node. It was also so that I could easily take the top two values off—since it was sorted, the top two values are the ones that I would want to combine to make a new node. The combination with a tree just made it easier for me to build, since there were already left and right pointers in addition to the previous and next pointers that the queue needed. I also had to use an array to store the decoded codes for the letters, so that I could use them when I wanted to encode a string.

I learned that sometimes you can take two data structures and combine them for your needs. Now that I think about it, the sparse matrix examples that the professor talked about in class was doing such combinations. Maybe a QueueTree isn’t the most elegant solution, but it served its purpose for me well, and will be something that I will remember to use in the future. I also learned that for some reason, it seems that there’s a problem with writing to StringBuilders when you have a nested loop. I’m not sure if it was just on my computer, or my version of java, but it took me about 3 hours to figure out that I could just write it to output (though I don’t know why writing it directly worked, but not appending to string/StringBuilder). Also, I learned that there are characters like no break spaces, which look like white spaces until you put it into Java. I have never encountered them before this lab. They made importing the frequency table a little harder than I expected.

Lastly, I learned just how useful exception catching was. I could catch an error that I expected, and use that to decide the next moves of the program. This was necessary to read in the frequency file correctly, as I couldn’t deal with the non-break spaces otherwise. I will definitely be adding this to my programming repertoire.

If I had to do something differently, I would try to figure out why StringBuilder is broken for the loops and use that. I like using Stringbuilder instead of writing directly to file, because you can manipulate the Stringbuilder later, but not written files. Also, I would like to be able to use Scanner and Priority Queue. Scanner would allow me to more easily ask for the file locations and whether a user wants to encode or decode. Priority Queue would have been a perfect solution for this, had I been allowed to import it. I would also like to learn how to use Comparator implementations.

For cost, I would say this is at least O(n). This is because I had one inner loop, when I was comparing the letters with the characters to output a translated code. This would scale with the number of frequencies in the frequency table. This is also true for the length of the cleartext string, as each additional letter involves adding another layer to the inner loop, causing it to run for m/2 times longer (m being the length of the frequency file). So for an encoding problem, you would have the cost of enqueuing. This would be the cost of insertion sort, about O(n^2). Then, you would have the task of building the tree. This is also O(n^2), since it needs insertion sort to handle the new nodes. Then there is the cost of reading the input file, which is O(n), since each line adds linearly to the cost. Then there is the cost of encoding, which is O(mxn/2), where m is the length of the frequency file and n is the length of the string. It is divided by half because the program stops looking once it finds a match. Thus, there is a 50% chance it’s in the first half of the file. For the cost of decoding, it is about O(nlogn), since it will have to traverse the nodes, and there are nlogn nodes once everything is completed.

For space complexity, it will be O(mlogm + n). It will increase with the size of the frequency file as well as the size of the string. However, it will increase faster with the size of the frequency file, because the tree will get larger, and the nodes will have larger values (aka more nodes, which is a O(nlogn) increase). It is a linear increase with string size, because each new character is expected to contribute a linear increase to file size.