**Programming Assignment 2 – Huffman’s Algorithm**

*Sam Bevans-Kerr*

This project took a long chunk of time for me to complete, even when I thought I understood how to make each function. For the Huffman Tree constructor in my code, the time complexity is O(n). In fact, each of my functions takes O(n) time to complete. Get\_character\_code is probably the most intensive function, requiring a full Huffman tree traversal using recursion. This also costs O(n). My encode function relies on get\_character\_code, and my decode function simply loops through a string. In most of the loops used in my code, the amount of loops is variable, and thus O(n).

My main data structure was a priority queue, modified to hold nodes, and pop them by their frequencies. It was helpful storing both nodes with ascii values, and pure frequency nodes (the sums) to sort. The queue was the main structure for creating the tree – each time two nodes are connected, their frequency sum is pushed back into the queue, ready for the next loop. This was a very good choice data structure wise. I can’t think of many other ways to complete a Huffman tree without some sort of queue or stack. It is very important not only to connect nodes, but to connect whole mini-trees together to create the final Huffman tree. I used simple nodes connected together by pointers to create the tree. Recursion was used to find leaf nodes to assign codes to, and simple loops were used to find the leaf nodes provided their code.

If I had to do this assignment again, I would definitely try to avoid overthinking the process. I feel I wasted many hours trying to solve a problem I had made far more difficult in my head. I kept an old un-working function commented in the project as a memento, as I spent many hours trying to work on it before rewriting. Many lines of code used to test the project are commented out but still left in, as they were immensely helpful in testing. If I had this project assigned again, I would look back over the tree traversal Stepik far sooner. The tree traversal Stepik assignment as well as the Solving a Maze with a Stack Stepik assignment were important in my get\_char() function. It was helpful in understanding the recursion needed to find a character in my tree. While I was writing my decode function, I tried to implement integer stacks instead of simply looping through the string of Huffman code. These mistakes ultimately cemented my learning and understanding of the project, though I do feel like the project itself was shrouded in too much initial mystery, and TA office hours were packed.

The hardest part of this project was either setting up the constructor to correctly build a Huffman tree, or encoding characters by traversing the tree. Traversing the tree, as stated above, took a long time to code and understand. I struggled with exiting recursion, as when my code “found” the key, it was often quickly forgotten in the memory due to the frequent recursion. The solution to this problem was setting the function to return a Boolean, as returning false quit the recursion. In terms of the constructor, my problem was again overthinking. Instead of just one special case (if there is only one character given), I had written in several. After a TA pointed out my mistakes, all it took was a few deletions of code to go back to where I was supposed to be. So while it was not as hard as encoding in the traditional way, creating the tree still threw me for a loop.

Overall, this project was one of the more interesting CISE projects I have completed in my opinion. The compression of text is a simple concept to understand (not so simple to implement) and has very obvious real-world applications. Knowing that Huffman coding is now one of the more basic text compression methods, I’m interested to learn about the more modern methods of text and data compression computer scientists use today.