Mitch Bouma
Project 2 Report
ECE 368
Compile line: **`gcc -Werror -Wall -O3 huff.c/unhuff.c -o huff/unhuff`**

This project consists of 2 files, huff and unhuff. Huff, the compression portion, works by first allocating extra memory in order to concatenate the .huff extension to the future compressed file. Then, it creates a 256-value array, and loops through the input file while incrementing the index that corresponds to each character that is read. This generates a count of how many times each ASCII character appears in the file (higher frequencies mean less bits used when compressing). It also returns the total number of unique characters found. Next, it calls the buildList function which loops through this new array and creates linked list nodes for any character that has a frequency greater than 0.These nodes are also inserted in order so that it is easier to choose the two nodes to combine when building the tree. It then adds one final node with the EOF value which I chose to be 0. Next, the tree is built. The struct used for the tree is the same as what is used in the linked list. It contains pointers to left, right, and next nodes, the ASCII value, and the frequency count. Obviously, the left and right pointers are set to null when in the linked list. It goes through the list, combining the first two nodes, setting the parent of those 2 to point at head -> next -> next, and then reinserting the parent in order in the list. Two functions are called that find the largest height and total number of leaves which are used as the number of columns and rows for a codebook of new bit sequences. To build this codebook, it traverses the tree. If it comes to a leaf node, it writes codebook[row][0] as the ASCII value at that node. Otherwise, it writes a 0/1 in the current column and down every row. As leaf nodes are met, the row is already at the right place, and the sequence has been written. A map is then set up which tells the row number in the codebook that a specific ASCII character can be found. The next step is writing the header to the new file. It does so by traversing the tree and writing a 1 if at a leaf node, followed by the bit sequence for the ASCII character at that node. After ending each left/right recursion, it then prints a 0 to show a non leaf node. The final step is to write the actual compressed characters. It does so by reading a

character, finding the row for that character in the codebook, and writing the bits in the following columns until it gets to a -1 (which means that is the end of that ASCII bit sequence). The write bit function which is used for both the header and compressed portions uses a mask of 1000 0000 that moves right one by one and is and'd with the current byte to determine whether to pack a 0 or 1. Finally, the padZero function adds whatever bits may be necessary to complete a byte.

The unhuff portion also first makes space for the added .unhuff extension. Then, it reads the header by reversing the process in huff. If a 1 is read, it creates a node and inserts it onto the linked list. After the 1 is read, a loop goes through and takes in the next 8 bits which correspond to the ASCII character at the leaf. This becomes the value in the node. If a 0 is read, the two nodes at the end of the list are merged and made into a tree. Once head -> next is NULL, only one node is left which means the tree is complete. It then follows the tree as bits are read in (left for 0, right for 1) and prints each leaf node's value as it gets there. The results for the given test cases can be seen below. As expected, the size actually increased for small files since it takes more space to save the header.

File name	Huff time (s)	Unhuff time (s)	Original size	Compressed size	Ratio
Text0.txt	0.000	0.000	4 bytes	10 bytes	0.400
Text1.txt	0.000	0.002	8 bytes	21 bytes	0.381
Text2.txt	0.000	0.022	155 bytes	127 bytes	1.220
Text3.txt	0.240	0.244	3 mb	2.20 mb	1.364
Text4.txt	0.506	0.478	6 mb	4.41 mb	1.361
Text5.txt	0.692	0.720	9.01 mb	6.61 mb	1.363