ECE368 Project 2 – Huffman Coding Report
Name: Yi En Gan
Method of compilation:
gcc -o huff huff.c

The project utilizes Huffman coding for the purpose of compressing and decompressing the input files. The solution to the project can be categorized into two components.

The encoding process utilizes three struct types.

```
typedef struct BTNode {
    int value;
    struct BTNode* left;
    struct BTNode* right;
} BTNode;

typedef struct {
    unsigned char* charArr;
    int* binArr;
} HuffTable;

typedef struct {
    int data[TREE_HEIGHT];
    int index;
} Stack;
```

The encoding process can be further categorized into four steps as shown in huff.c.

The first step is the opening and counting of the input text file. This part utilizes built-in C functions such as fopen() and fgetc() to iterate through the input file and compute the ASCII frequency array.

The next step is the construction of the Huffman tree. Using the frequency array, the function generates a binary tree by locating the two least frequent ASCII characters in the text file and forms a leaf node pair with a common parent. This continues until all ASCII characters in the input text file are represented. A pseudo EOF is inserted into the tree with a frequency of one.

The third step is the construction of a Huffman table. The data structure of the table consists of an array of type char* and an array of type int*. Given that the program only registers the value of leaf nodes, the type of traversal used is inconsequential. The Huffman coding of a character is the path taken from the root node to the leaf node. A stack is used to achieve this.

Finally, the program writes the header information, which is the header size followed by the pre-order traversal of the binary tree. It then reads the input file in chunks and encodes each character in the chunk buffer. The program alternates between calling the read and encode/write function until the bytes read is not equal to the chunk size, which indicates that the EOF has been reached.

The encoding process requires the program to locate the Huffman coding of the character in the Huffman table. Once it is located, the bits are written into a char type value and shifted to the left by multiplication of two. The program consistently checks if eight bits have been written into the char value. If so, the char value is written into the output file and the char value is reset.

The other component is the decoding process, detailed in unhuff.c.

The unhuff.c only uses HuffTable and Stack data structures.

Header Format in Encoded File:
[header_size] [most_freq_char] […] [least_freq_char]

ASCII Character Position in Header	Huffman code
0	00
1	01
2	100
3	101
4	1100
5	1101
6	11100

Unlike the encoding process, the decoding process does not make use of the binary tree. Instead, it reads the header information. Due to the structure of the binary tree being standardized for the purpose of Huffman coding, the Huffman code of the ASCII characters can be determined by their position in the header information. It is found that the Huffman code of the nth character is (n/2) 1s followed by either 00 or 01. The edge cases for when there are less than 2 leaf nodes and for the two least frequent leaf nodes are accounted for in the program.

The next step is the reading and decoding of the text file. Like huff.c, the decoding process reads the input file in chunks and decodes them in chunks. The program alternates between calling the read and decoding/write function until the bytes read from the input file is unequal to the chunk size. The decoding program takes special care of the bytes to be written into the output file.

Given the Huffman code for a character may be split between two bytes, the decoding function will have to store the remaining unprocessed bytes in a stack and return them to the next decoding call. After every decoding call, the read function reads a new chunk into an unsigned char* buffer to be processed by the decoding function. Once a Huffman sequence has been identified from the chunk buffer, the corresponding ASCII character in the Huffman table is printed into the output file. The program terminates once the pseudo EOF character has been identified.

After running the program, it is found that the encoded file is approximately the same size as input files that are small. For larger input files, the size of the encoded file is usually 80~90% of the input. The compression ratio of the file corresponds with the size of the input file with respect to the header size, and the number of unique ASCII characters that appear in the input text file.

The weakness of the Huffman coding is revealed when the input file contains numerous unique characters that do not appear in the file frequent enough. The input file causes the Huffman coding of some characters to exceed 8 bits or 1 byte, which fails to achieve the goal of compression.