Report for ECE368 Project 2

This project requires a program to compress an ASCII file using Huffman Coding, and another program to decompress the compressed file. There are three goals of this project.

1. Write the program that can restore the compressed file without any loss (PRIMARY)
2. The compressed file should take as less space as possible (SECONDARY)
3. The compressing/decompressing process should be as fast as possible (OPTIONAL)

Explanation and Complexities:

My compressing program has two major parts: 1. Build a Huffman tree; 2. Based on the tree, compress the file.

In order to build the tree, the function "`CountASCII`" is used to count the frequency of every ASCII character in the input file. DC3, with ASCII value of 19, is the EOF indicator which normally should not be in regular files. The count of DC3 will be manually set to 1. The function will scan the input file once. The frequencies are stored in an array of structures. Each structure contains the ASCII value, the frequency, and the key (in decimal), the length of the key, and a property called "Connected". Then, "`BuildHeader`" function wrote those frequencies onto the compressed file as the header. The frequencies are written in order of the ASCII values. After that, "`SortArray`" function sorts the ASCII array based on the frequency, and returns the total number of ASCII values that have been used by the input file. Then, "`BuildTree`" function build the Huffman tree based on the project description. The function creates an array with **twice** the number of used ASCII characters, and the tree is actually in an array, with half of its entries being the leaf nodes, and the other half of its entries being the parents of the leaves. The reason for that is during the building process, I can easily find the smallest nodes—both leaves and parents of leaves. There is a property called "Connected" in every node. When two leaves have been connected by their parents, Connected is set to 1, and later comparison will ignore these nodes with Connected = 1. This function will return the head of the tree. After the tree has been built, "`SecondSortArray`" function sorts the array based on ASCII value instead of frequencies, and "`UpdateASCII`" traces through the tree, and writes paths for every leaf as keys to entries in the array, where keys are how to find the corresponding character in the tree. (Starting from the head, 0 represents go left, and 1 represents go right)

Finally, "`Compressing`" function reads the input file again, and writes every character's binary key in the ASCII array onto the output file. To do so, there is a ByteBuffer with maximum length of 8. If the key is less than eight, it will be shift to the front of the ByteBuffer, and the next character's key will be written after the first one. After the last byte has been written, the key of DC3 will follow. For example:

Input file content: Abc, where A's key is 011, b's key is 0110101, c's key is 00000, DC3's key: 101

Byte 1: 01101101 Byte 2: 01000001 Byte 3: 01000000

Then, for every byte, the binary value was converted to integer, and the ASCII character at that ASCII value will be written to the file. This is what I called a Base 256. The reason for this final conversion is that a byte in the program may contain the value of 255, which requires 3 bytes when written to the file.

Byte 1: 109 Byte 2: 65 Byte 3: 64

Byte 1: 'm' Byte 2: 'A' Byte 3: '@'

Format: The compressed file contains header: frequencies of 256 ASCII characters, and key of every characters from original file.

For the first part of the problem, or building the tree, the problem size is 256 at most (except counting the frequency part). The second part, which is to build the tree, the problem size will depend on the size of the input file. Therefore, one goal of the first part is to reduce the operating time of compressing every character, which will happen in the second part. That is why array is used, since access time is O(1). The time complexity of the first part is O(N), since the program has to scan the entire input file to get the frequency. The space complexity is fixed, or O(1). No matter the problem size, at most 512 structures will be allocated for that ASCII array, and that number minus 1 also represents maximum number of nodes on the Huffman Tree. (the last node has been manually written a very big frequency for finding the least and the second least frequency) To generate keys of every character, recursion is used, and has time complexity of O(Depth of the tree), which at most being 256. Comparing to the problem size, which may exceed 1 million, the time and space of the keys may still be ignored.

For the compressing part of the program, the operating time will highly depend on the problem size. The function easily reads the character's ASCII value, and uses the value as index to find the key of that character. This requires O(n) for both time and space complexity.

The decompressing process also has two parts: 1. Based on the head of the file, generate the Huffman Tree using the same function as compressing; 2. Based on the Huffman Tree and the keys in the compressed file, find the corresponding ASCII character.

For the second part, the file is read byte by byte. Then, a use the AND logic, I filter one bit by another within a byte. Base on the bit value, I find the ASCII character in the tree. If already reached the end of one byte, but has not reached the leaf node, another byte will be read. When the DC3 is reached, I do not write it to the file, and terminate the function.

The first part requires O(1) for both space and time complexity as usual. The decompressing part requires O(N) for time complexity. It does not allocate extra memory, but it does require extra space on the disk to write the output file, and the space complexity is O(N).

Testing and validating:

A powerful software—Beyond Compare is used during the testing, which highlights the difference between file. There are four test cases for the file:

1. A 10-character file, which is used for debugging;
2. A 5.22 MB file written in English to represent a huge ASCII file;
3. A 258 KB SAP IDoc file, which represents a special file with a lot of extended ASCII codes that are not for common use;

Results:

File	File Size	Compressed Size	Compressing Ratio	Compressing Time	Decompressing Time	Matching Rate
1	12 B	520 B	4333%	~0 sec	~0 sec	100%
2	5.22 MB	3.13 MB	59.96%	1.87 sec	0.61 sec	100%
3	258 KB	44.1 KB	17.09%	0.03 sec	0.02 sec	100%

Math library is used because I need the power function to convert binary to decimal. Nothing from ECE 264 is being used here. Tree is used in both parts.