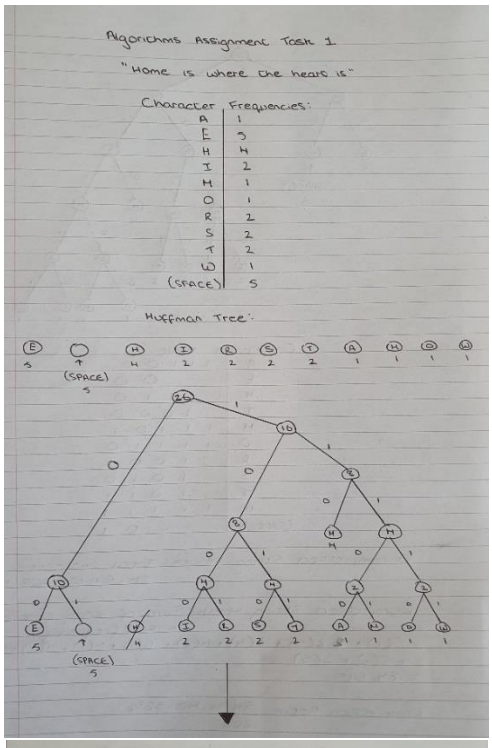
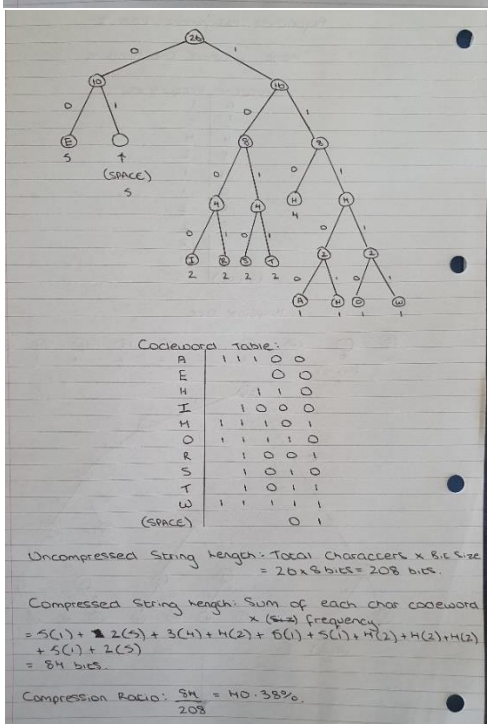
COMP20290 – Assignment – 19369726 – Patrikas Seskauskas

Task 1:





Task 2:

Link to Huffman Algorithm class in my repository (Usage instructions are in comments in the class): https://github.com/CompAlgorithms/algorithms20290-2021-repository-PSeskauskas/blob/main/src/Assignment/HuffmanAlgorithm.java

Task 3:

(Step 1) Compression:

File Name	Compressed File	Original Bits	Compressed Bits	Time (s)	Compression Ratio = Compressed/Original Bits
genomeVirus.txt	**genomeVirusComp.txt**	**50008**	**12576**	**0.011**	**25.15%**
medTale.txt	**medTaleComp.txt**	**45056**	**23912**	**0.006**	**53.07%**
mobydick.txt	**mobydickComp.txt**	**9706488**	**5504496**	**0.220**	**56.71%**
q32x48.bin	**q32x48Comp.bin**	**1536**	**816**	**0.003**	**53.13%**
loremIpsum.txt	**loremIpsumComp.txt**	**104024**	**56224**	**0.014**	**54.05%**

(Step 2) Decompression:

Compressed File	Decompressed File	Compressed Bits	Decompressed Bits	Time (s)
genomeVirusComp.txt	**genomeVirusDecomp.txt**	**12576**	**50008**	**0.005**
medTaleComp.txt	**medTaleDecomp.txt**	**23912**	**45056**	**0.006**
mobydickComp.txt	**mobydickDecomp.txt**	**5504496**	**9706488**	**0.159**
q32x48Comp.bin	**q32x48Decomp.bin**	**816**	**1536**	**0.002**
loremIpsumComp.txt	**loremIpsumDecomp.txt**	**56224**	**104024**	**0.010**

(Step 3) Analysis:

The time complexity of the Huffman compression algorithm is O(n log n).
The compression/decompression time depends on the number of bits in the file.

Compression Analysis:

- **genomeVirus.txt:** This .txt file contains Genomic code, containing the characters A, C, T, G only. Hence, the character frequency table would only have 4 entries of similar frequencies. Yielding a balanced Trie with equal length compressed codewords and a 25.15% compression ratio.
- **medTale.txt:** This .txt file contains medieval English text, giving a compression ratio of 53.07%, the lowest of the English/Latin texts tested.
- **mobydick.txt:** This .txt file contains the full Moby Dick text. This text is 22158 lines long and contains approximately 9.7 million bits causing the Huffman Algorithm to take longer than the other files used at 0.220 seconds. Although this text yields the highest compression ratio of all the files, 56.71% is still very reasonable for the size of the file.
- **q32x48.bin:** This file is of type bitmap, containing binary data. Therefore, the character frequency table would only contain 2 characters with high frequencies, yielding an expected 53.13% compression ratio.
- **loremIpsum.txt:** This .txt file contains several paragraphs of Lorem Ipsum text in Latin. Compressing this file gives a compression ratio of 54.05%, similar to both mobydick.txt and medTale.txt which are both English language texts. Seemingly, the language of a text file does not affect the compression ratio due to the alphabets for certain languages being very similar.

Decompression Analysis:

- Generally, the decompression time for each file was quicker, due to the Huffman tries already being built during compression, therefore, during decompression, to decompress the file each codeword would just need to be looked up on the Trie and expand each codeword to its original.
- In every case, the decompression algorithm restores the original bit size for every file along with all the original text. Therefore, the algorithm works as intended as a lossless compression algorithm.

Q3:

File Name	Original Bits	First Compression Bits	First Compression Ratio	Second Compression Bits	Second Compression Ratio
genomeVirus.txt	**50008**	**12576**	**25.15%**	**14896**	**118.45%**

I attempted to compress the genomeVirus.txt file a second time, the number of bits along with the compression ratio increases rather than decreases. First compression of the genomeVirus.txt file gives a compression ratio of 25.15%, whereas a second compression gives a compression ratio of 118.45%.

I believe that this occurs because the Huffman compression algorithm compresses ASCII characters that are 8 bits in length down to codewords that are lower than 8 bits. In the case of this file, each codeword would be of length 2 bits due to it containing Genomic code, making the character frequency table only have 4 entries. Therefore, the compressed character lengths of 2 bits cannot be compressed further since they're below 8 bits in length, making any attempt at compressing a second time useless.

Q4:

File Name	Original Bits	Huffman Compression Bits	Huffman Compression Ratio	Run Length Encoding Bits	Run Length Encoding Ratio
q32x48.bin	**1536**	**816**	**53.13%**	**1144**	**74.48%**

Running q32x48.bin through Huffman compression yields a more compressed file than with Run Length Encoding which can be seen with the lower compression ratio of 53.13% for Huffman Compression as opposed to 74.48% for Run Length Encoding.

In this case, Huffman compression had a 20% better compression ratio than Run Length for the file. This is due to the file in question being a bitmap file, containing binary 0s and 1s as its data.

With Huffman compression, there would only be 2 symbols with high frequencies in the character frequency table. Giving a very small Trie with very short codewords because Huffman compression relies solely on character frequency for its compression. The order of characters doesn't influence the compression algorithm.

Whereas for Run Length Encoding, the frequency, and the order in which each character occurs does influence the compression algorithm and thus the efficiency of the algorithm. Meaning that generally, the Huffman algorithm will work better than the Run Length Encoding algorithm.