Task 3: Compression Analysis

Q1. Calculate the compression ratio achieved for each file. Also, report the time to compress and decompress each file.

File	**Original Size**	**Compressed Size**	**Compression (Compressed size / Original size)**	**Nano time taken**
mobydick.txt	1,164 KB	651 KB	217/388 = 55.93%	1007033441
q32x48.bin	192 B			
medTale.txt	5.59 KB	3.00 KB	300/559 = 53.67%	214229743
genomeVirus.txt	6.10 KB	1.53 KB	153/610 = 25.08%	299173114
lana.txt	291 KB	170 KB	170/291 = 58.42%	814423893

Q2. Take the files you have just compressed and decompress them. Report the final bits of the decompressed files and the time taken to decompress each file.

File	**Decompressed Size**	**Nano time taken**
mobydick.txt	1,164 KB	482816703
q32x48.bin	192 B	
medTale.txt	5.59 KB	92737720
genomeVirus.txt	6.10 KB	118584502
lana.txt	291 KB	351867676

Q3. What happens if you try to compress one of the already compressed files? Why do you think this occurs?
The file grows slightly larger. We believe this occurs as the data cannot be losslessly compressed any further using huffman encoding, and because of this the result of compression remains the same size but has an encoded trie added to the start, making the file larger. The file cannot be further compressed as the bitstring shows no pattern in the distribution of the bytes, meaning that huffman encoding, a probability-based encoding which capitalises on the distribution of letters in most natural language texts, will not yield positive results when applied to its own output.

Q4. Use the provided RunLength function to compress the bitmap file q32x48.bin. Do the same with your Huffman algorithm. Compare your results. What reason can you give for the difference in compression rates?
File Size: 192 B
RunLength Compression file size: 143 B
Huffman Compression file size:

Unfortunately, we were not able to get the huffman compression algorithm to compress/decompress the binary file properly. We did change the compression model from dealing with chars to dealing with bytes, thinking this might remedy the issue, but unfortunately this did not work.

After some further investigation, the issue with binary files has not been resolved, however there was a line of investigation followed which led to the discovery of at least an indication of the root of the problem.

The line of investigation boiled down to examining the bit patterns of the bytes in the file, and the bit patterns of the bytes which Java read from the file.

When examining q32x48.bin, it is found that the 10th byte is 0xFC (or 11111100 in binary). However, when examining the bytes which Java had ingested from the file, it was discovered that the 10th byte was 0x3F (or 00111111 binary). What I noticed about both of these numbers is that they are the reverse of each other. How exactly this kind of translation error had occurred I do not know, however it became apparent to me that the root cause of the problem was somewhere in the file i/o. Unfortunately, however much I tried to remedy the problem by refactoring the BinaryStdIn class, I couldn't get the program to interpret the byte correctly. While not at all a justification for the program not handling binary files properly, I did feel it necessary to at least explain where the shortcoming was originating from.