COMP20290 COMPRESSION ASSIGNMENT

By Simonas Ramonas 18763829 and Shijie Liu 18439314

Compression Analysis

Q1: Compression ratio and time.

File – mobydick.txt -> original bits: 9,531,704

	Huffman Compression	RLE Compression
Compressed bits	5,341,208 bits	38,698,936 bits
Compression ratio	0.56	4.06
Compression time	173.39ms	100.87ms

File – medTale.txt -> original bits: 45,056

	Huffman Compression	RLE Compression
Compressed bits	23,912 bits	182,520 bits
Compression ratio	0.53	4.05
Compression time	26.48ms	9.8ms

File – genomeVirus.txt -> original bits: 50,008

	Huffman Compression	RLE Compression
Compressed bits	12,576 bits	223,632bits
Compression ratio	0.25	4.47
Compression time	24.22ms	11.46ms

File – q32x48.bin –> original bits: 1,536

	Huffman Compression	RLE Compression
Compressed bits	816 bits	1,144 bits
Compression ratio	0.53	0.74
Compression time	15.2ms	1.95ms

Extra file – huffmanCoding.txt -> original bits: 212,288

	Huffman Compression	RLE Compression
Compressed bits	127,672bits	867,984bits
Compression ratio	0.6	4.09
Compression time	41.35ms	22.21ms

From the result of the compression comparison, we see that Huffman algorithm results in a much better compression ratio than RLE compression. Saving 40 – 50% of memory, occasionally saving 70-80% based on the file(i.e. like genomeVirus.txt).

As for time analysis, Huffman algorithm has the time complexity of O(n log n), and Run Length algorithm has time complexity of O(n), going through the each element once.

During our analysis testing, we realised that even though Huffman compressed the file down to a smaller size than the file compressed by RLE Compression, the time taken for compression however was longer than RLE. (Explanation is also answer for Q4 below :))

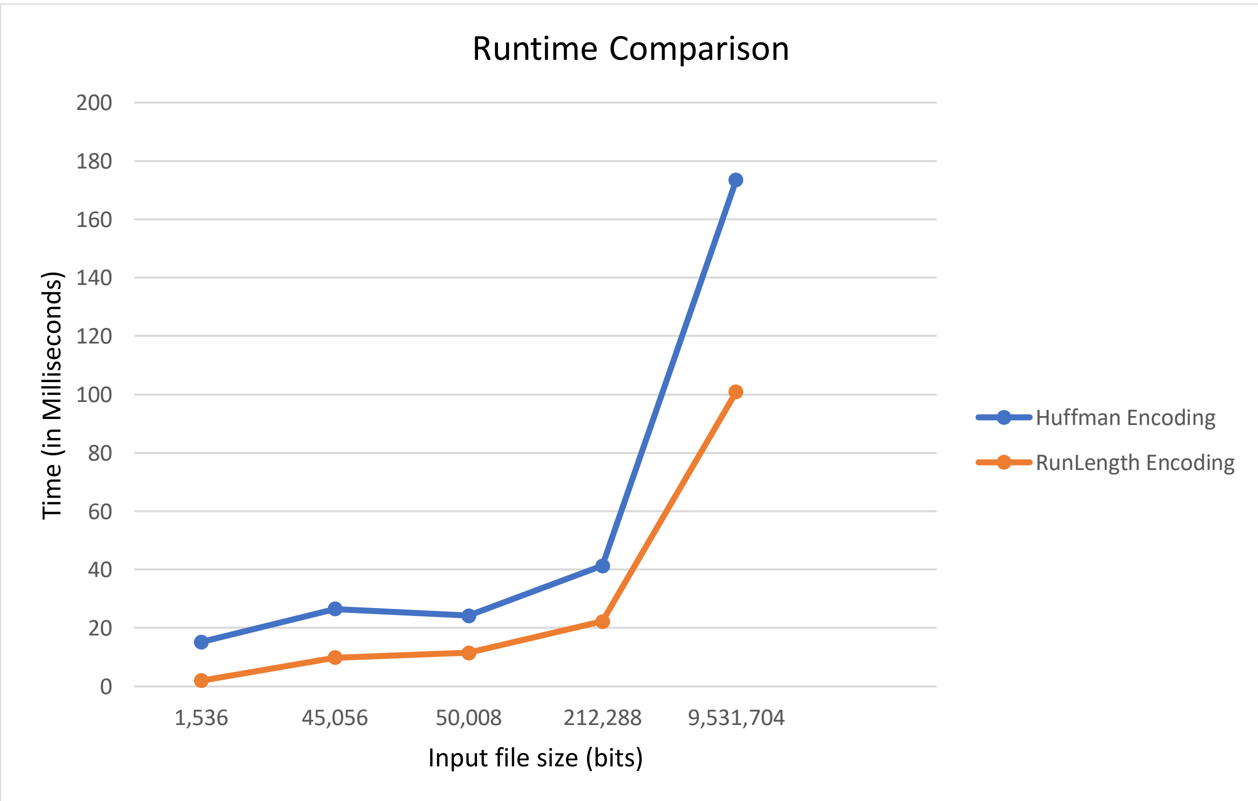
This was the case because of their compression techniques, with RLE compression, it runs through the file once and outputs the character followed by the number of consecutive occurrences of that character and goes onto next character.

With Huffman algorithm, it runs through the file, counting the frequency of each character, and with MinPQ, it produces a Huffman binary tree structure storing the characters as nodes from least frequent (leaf nodes) to most frequent (parents/close to root). It then goes through the file again and encodes the file with the "code table" made from trie.

The reason that Huffman algorithm produces a smaller compressed file is that it gets the total frequency of characters in a file, whereas Run Length compression gets frequency of continuous characters. For example, with the String "ABABAB", Huffman would say A has 3 occurrences, B has 3 occurrences, make A = code 0, B = code 1.

With Run Length even though it is faster, but it gets bigger as it will say A1B1A1B1A1B1.

(Example here is also answer for Q4, but also included here for clarification).



Q2: file decompression analysis

Huffman Decompression:

File name	Decompressed size:	Time (in milliseconds
mobydick.txt	9,531,704	88.87
medTale.txt	45,056	6.67
genomeVirus.txt	50,008	8.19
q32x48.bin	1,536	2.7
huffmanCoding.txt	212,288	14.92

Run Length Decompression:

File name	Decompressed size:	Time (in milliseconds
mobydick.txt	9,531,704	239.12
medTale.txt	45,056	14.29
genomeVirus.txt	50,008	11.89
q32x48.bin	1,536	2.09
huffmanCoding.txt	212,288	27.38

Both algorithms were lossless when compressing and decompressing, bringing the file back to its original size. We can see that generally Huffman decompress is faster than Run Length decompression. Its decompression time is less than half of its compression time.

Whereas Run Length decompression time is almost always longer than its compression time.

Q3: Compressing file again.

File name	First compression size (bits)	Second compression size (bits)	Third compression size (bits)
mobydick.txt	5,341,208	5,263,336	5,265,928
medTale.txt	23,912	25,960	28,432
genomeVirus.txt	12,576	14,896	17,352
q32x48.bin	816	1,272	2,312
huffmanCoding.txt	127,672	128,968	131,528

When we try to compress a file that is already compressed, the resulting file is bigger than the compressed file. This is because Huffman compression uses a representation for each individual symbol. So, when we try to compress the compressed file, some new symbols might be created and therefore the representation might take up more space than it did previously.

Q4: bin file comparison:

q32x48.bin -> original size: 1,536 bits

Huffman compression ratio:	816/1536 == 0.53
Run Length ratio:	1144 / 1536 == 0.74

From our analysis we can conclude that Huffman compression has a better compression ratio than Run length compression.

The reason that Huffman algorithm produces a smaller compressed file is that it gets the total frequency of characters in a file, whereas Run Length compression gets frequency of continuous characters. For example, with the String “ABABAB”, Huffman would say A has 3 occurrences, B has 3 occurrences, make A = code 0, B = code 1.

With Run Length even though it is faster, but it gets bigger as it will say A1B1A1B1A1B1.