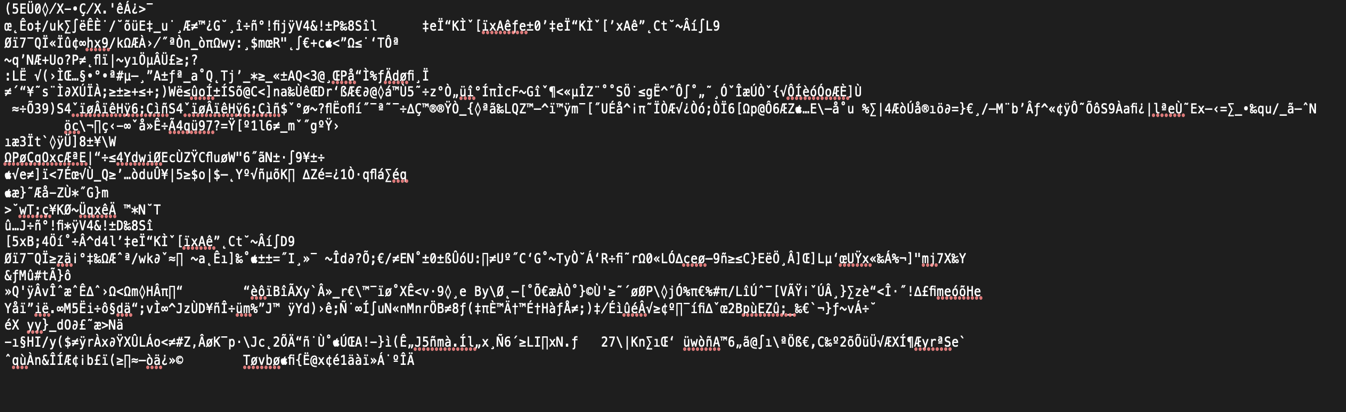
**Zahara Vazir 18417696**

**Gavin Ka Cheung 18303701**

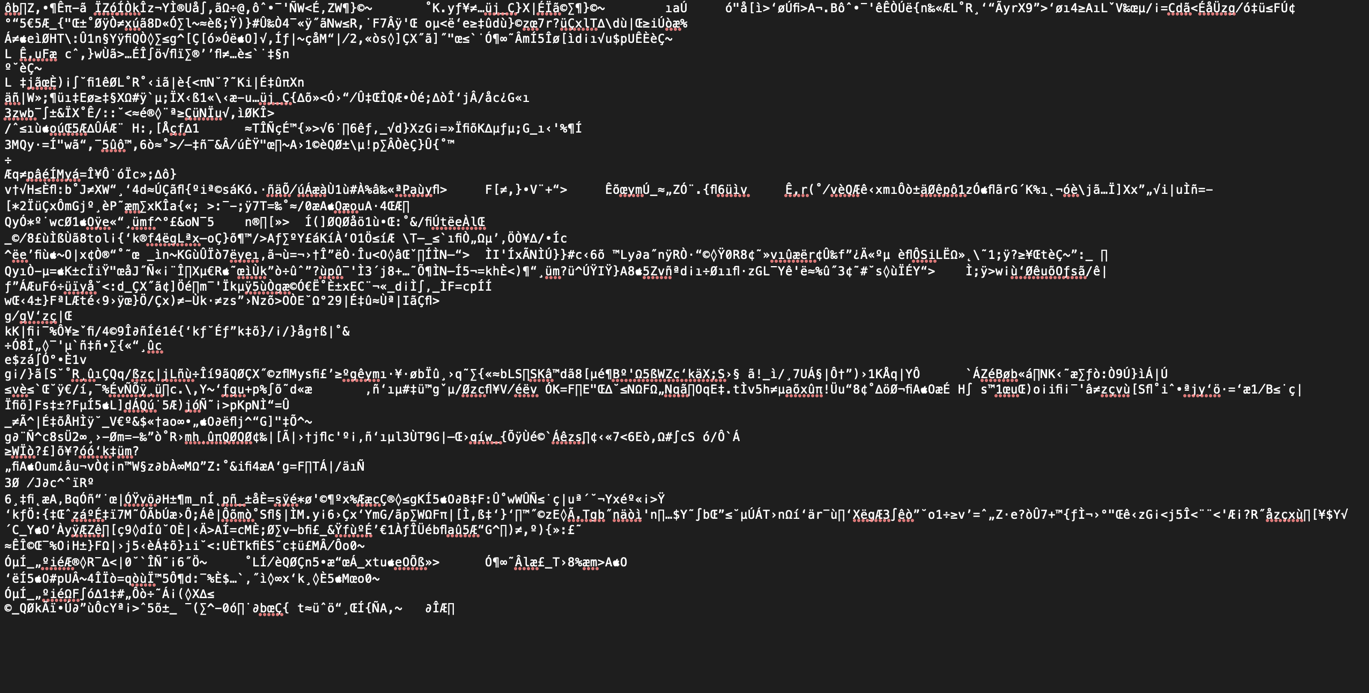
This assignment was completed in a pair. Gavin implemented the coding algorithms. I completed the hand drawn Huffman Tree and did all the analysis for the different compression algorithms. I provided extensive testing and results.

**Algorithms Assignment - Part 3 : Testing**

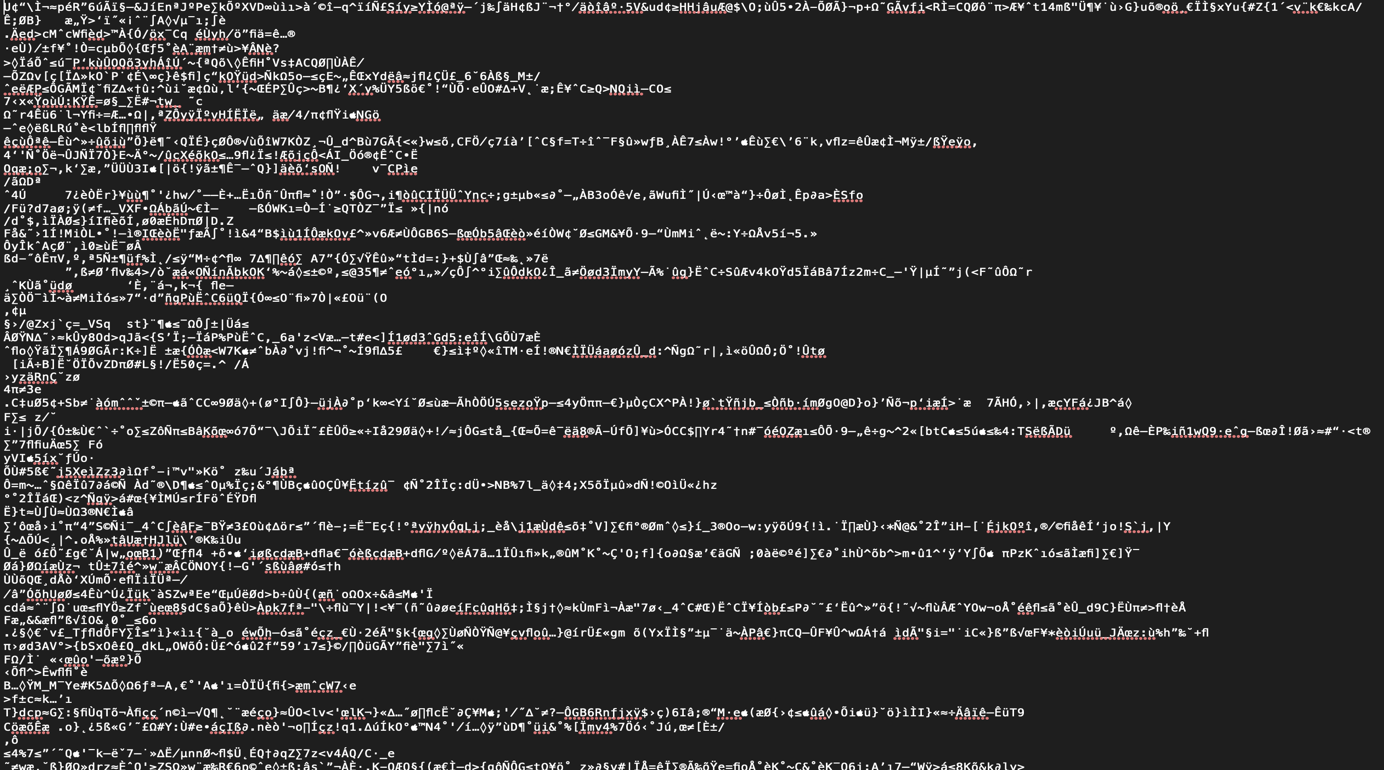
GenomeVirus.txt compressed file



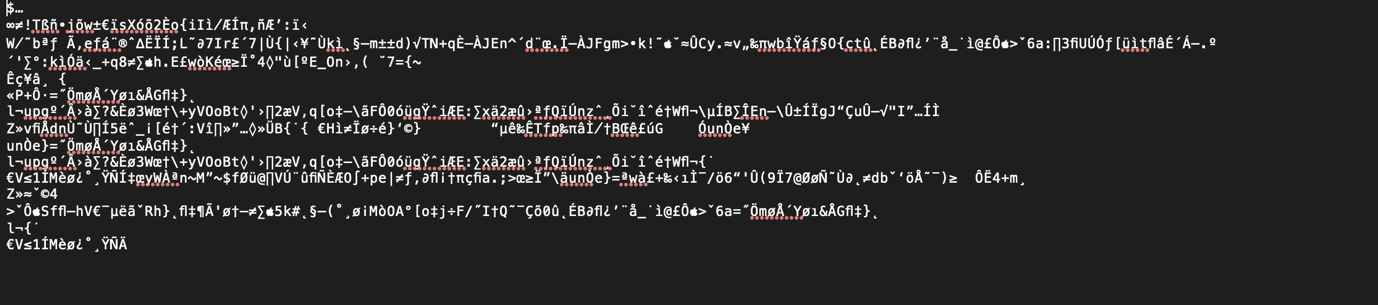
medTale.txt compressed file



Mobydick.txt compressed file



Rocketman.txt compressed file



Calculating Compression Ratios and Time Taken For Compression :

Huffman\_starter :

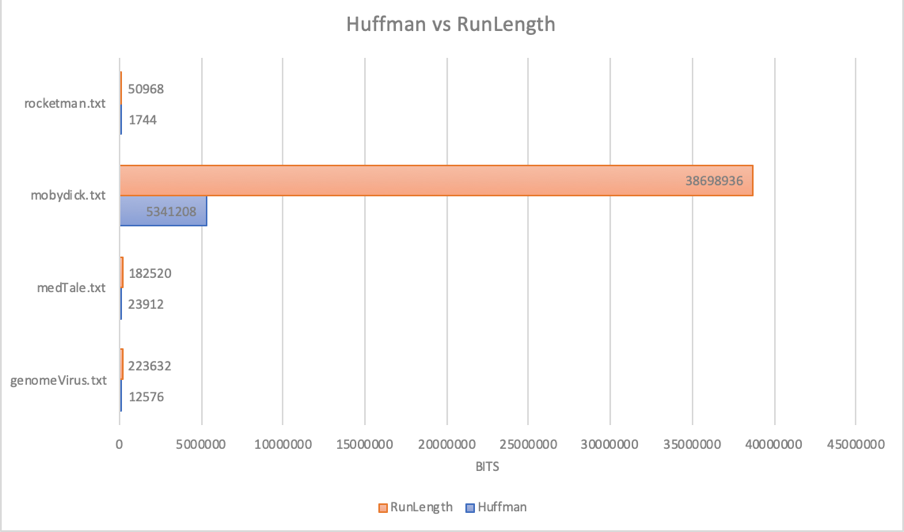
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Time taken for compression : (milliseconds) | Original Bits : | Compressed Bits : | Compression Ratio : |
| genomeVirus.txt | 11.5 | 50008 Bits | 12576 Bits | 25.15% |
| medTale.txt | 16.5 | 45056 Bits | 23912 Bits | 53.01% |
| mobydick.txt | 106.5 | 9531704 Bits | 5341208 Bits | 56.04% |
| rocketman.txt | 7 | 12632 Bits | 1744 Bits | 13.80% |

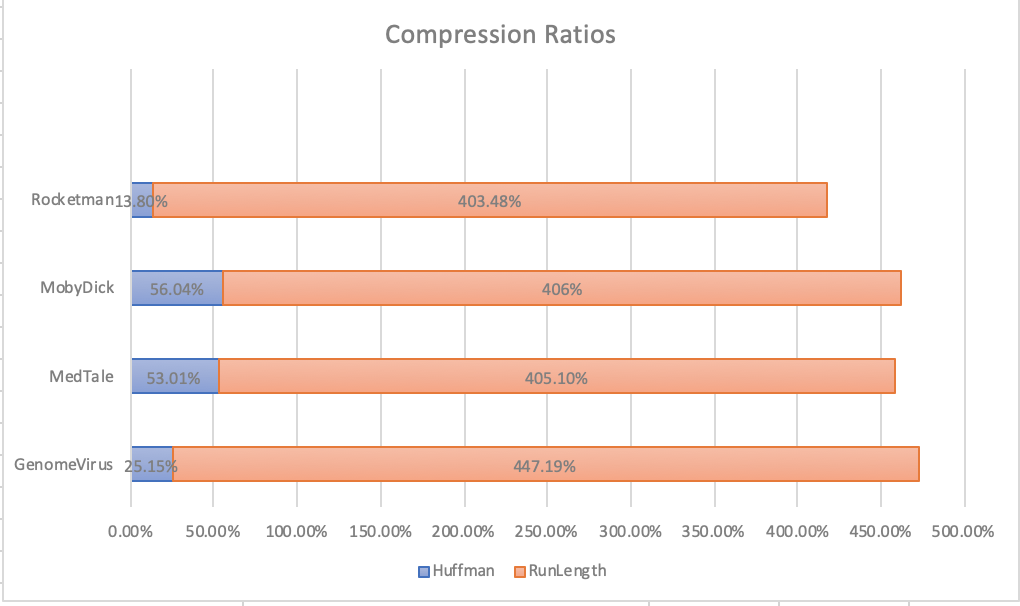
HuffmanAlgorithm :

|  |  |
| --- | --- |
|  | Time taken for compression:  (milliseconds) |
| genomeVirus.txt | 86 |
| medTale.txt | 58 |
| mobydick.txt | 14,028 |
| rocketman.txt | 64 |

RunLength Algorithm :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Time taken for compression : (milliseconds) | Original Bits : | Compressed Bits : | Compression Ratio : |
| genomeVirus.txt | 7 | 50008 Bits | 223632 Bits | 447.19% |
| medTale.txt | 5 | 45056 Bits | 182520 Bits | 405.10% |
| mobydick.txt | 101.75 | 9531704 Bits | 38698936 Bits | 406% |
| rocketman.txt | 9 | 12632 Bits | 50968 Bits | 403.48% |





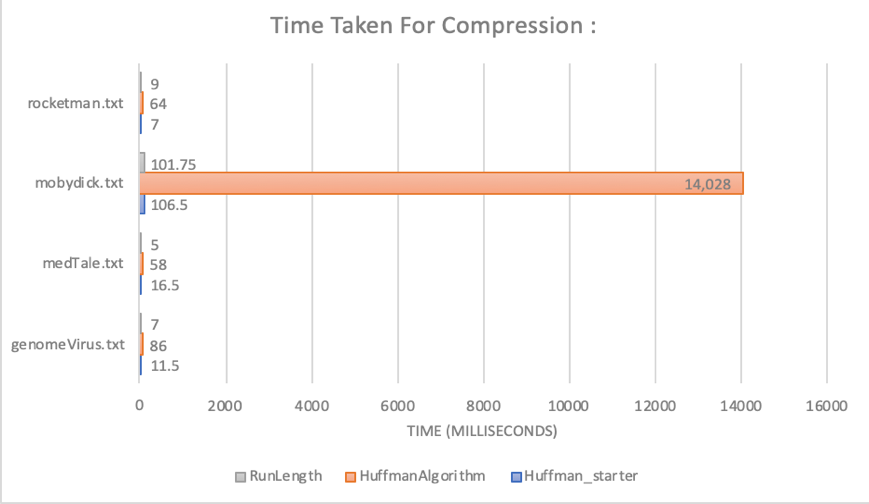
Calculating Time Taken For Decompression and Final Bits :

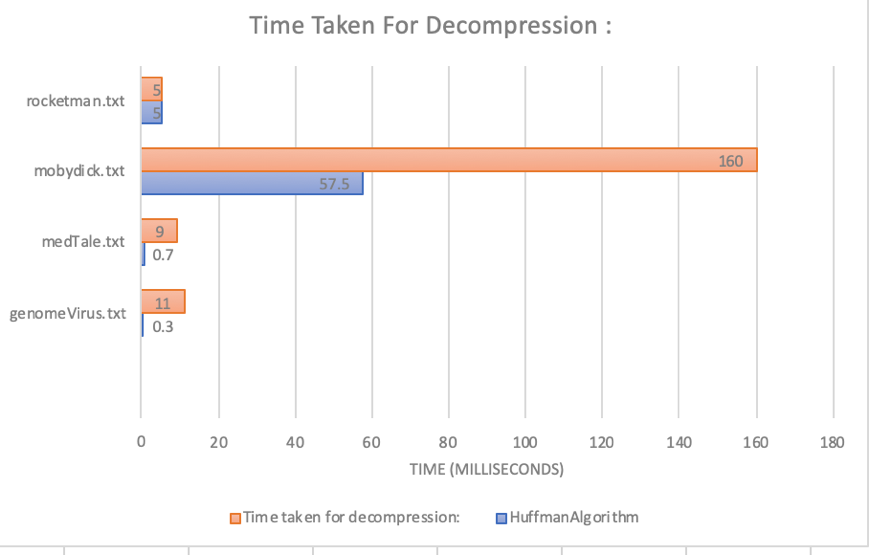
HuffmanAlgorithm :

|  |  |  |
| --- | --- | --- |
|  | Time taken for decompression:  (milliseconds) | Final Bits : |
| genomeVirus.txt | 0.3 | 50008 Bits |
| medTale.txt | 0.7 | 45056 Bits |
| mobydick.txt | 57.5 | 9531704 Bits |
| rocketman.txt | 5 | 12632 Bits |

RunLength Algorithm:

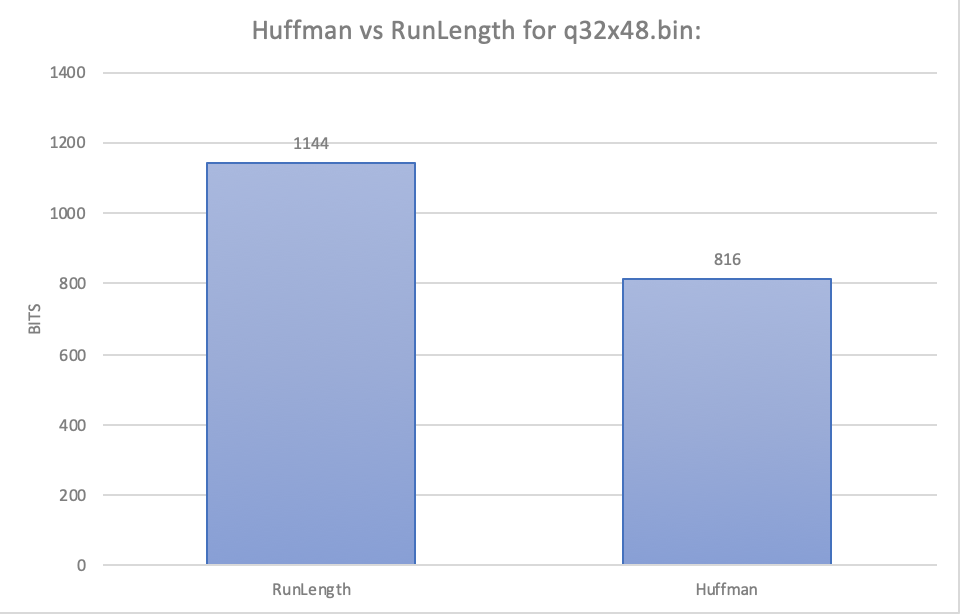
|  |  |
| --- | --- |
|  | Time taken for decompression:  (milliseconds) |
| genomeVirus.txt | 11 |
| medTale.txt | 9 |
| mobydick.txt | 160 |
| rocketman.txt | 5 |





Number of original bits for q32x48.bin : 1536 bits

|  |  |  |
| --- | --- | --- |
|  | Compressed Bits for q32x48.bin : | Compression Ratio for q32x48.bin: |
| RunLength | 1144 Bits | 74.48% |
| Huffman | 816 Bits | 53.125% |



Analysis of results :

There were two of us working on this assignment. We compared the Huffman and RunLength algorithm, which are two common compression techniques, to analyse and examine the performance of our Huffman code. We implemented our own Huffman algorithm called HuffmanAlgorithm.java as well as the Huffman\_starter.java given to us.

We can see from the results, compressing the files using the Huffman algorithm had better compression ratios compared to the RunLength algorithm. The compression ratios for the Huffman algorithm were lower indicating that this algorithm worked more efficiently in order to compress the given text. Therefore, it reduced a lot of the space that was needed to store data. The compression ratios for the text files compressed by the Huffman algorithm were between 13% to 53%, whereas with the RunLength algorithm the compression ratios were far higher, starting from 405% and going up until 447%. This is due to the fact that the RunLength compression algorithm is ineffective when it comes to compressing ASCII characters in comparison to Huffman. As a result the compressed bits were higher than the number of original bits for RunLength, giving larger compression ratios.

As you can see from the table we have two sets of times for compression using Huffman. The first set of data is the time it took to compress the file using the Huffman\_starter code and the second is using our implementation, HuffmanAlgorithm. After carrying out the tests and from the statistics we obtained, we can see that the HuffmanAlgorithm was the slowest performing algorithm compared to Huffman\_starter and RunLength. This does not mean that it is less efficient, as we can see when we discussed above how the compression ratios for RunLength were much greater when set side by side to the compression ratios for Huffman. This is not the result that is favourable as lower compression ratios are more desirable and as a result the time taken for compression and decompression does not say much about the performance of the code. Another observation we gathered from the data was how the times for the decompression in Huffman was lower than the compression times, but this was the other way around for RunLength. The times for decompression were higher than the compression times except for the rocktman.txt. There was a drop in the decompression time by 4 milliseconds.

The RunLength compression algorithm worked well to compress the q23x48.bin file as it had a much lower compression ratio compared to when this algorithm was used to compress the other text files. The compression ration using RunLength was 74.48%, which was not far off 53.125%, which was the compression ratio using the Huffman algorithm. Since the q23x48.bin file is a binary file which consists of 0s and 1s, the RunLength algorithm worked well in order to reduce the space compared to the other text files we examined that contained ASCII characters.

After compressing an already compressed file to a new text file, we discovered that is was possible to do. When we tried to do this, we noticed that it resulted in a slightly bigger compressed file. The reason we think this is, is due to the fact that the compression algorithm can’t always do perfect compressions and since we are compressing an already compressed file, it has to change from that data to another set of data about the data itself. Overall, it is not a good idea to compress a file that has already been compressed as it defeats the purpose of compression, as it increases the size of the compressed file. It may only increment the size by a small amount, but if compression is repeated multiple times on a file, it may give a higher compression ratio over time.

In conclusion, based on our testing and data, we can see how the Huffman Algorithm was an adequate compression algorithm compared to RunLength when it came to compressing text files consisting of ASCII characters and also .bin files. It had far lower compression ratios, which is the outcome we wanted to achieve as this results in more space being saved and therefore reduces the time taken to locate data on the drive.