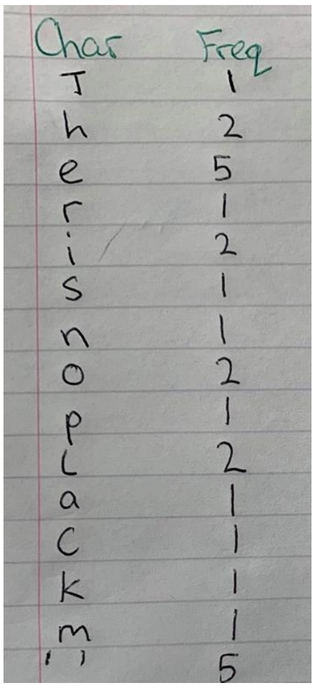
COMP20290 Algorithms – Huffman Compression Assignment

Iva Mechkarova – 18345221

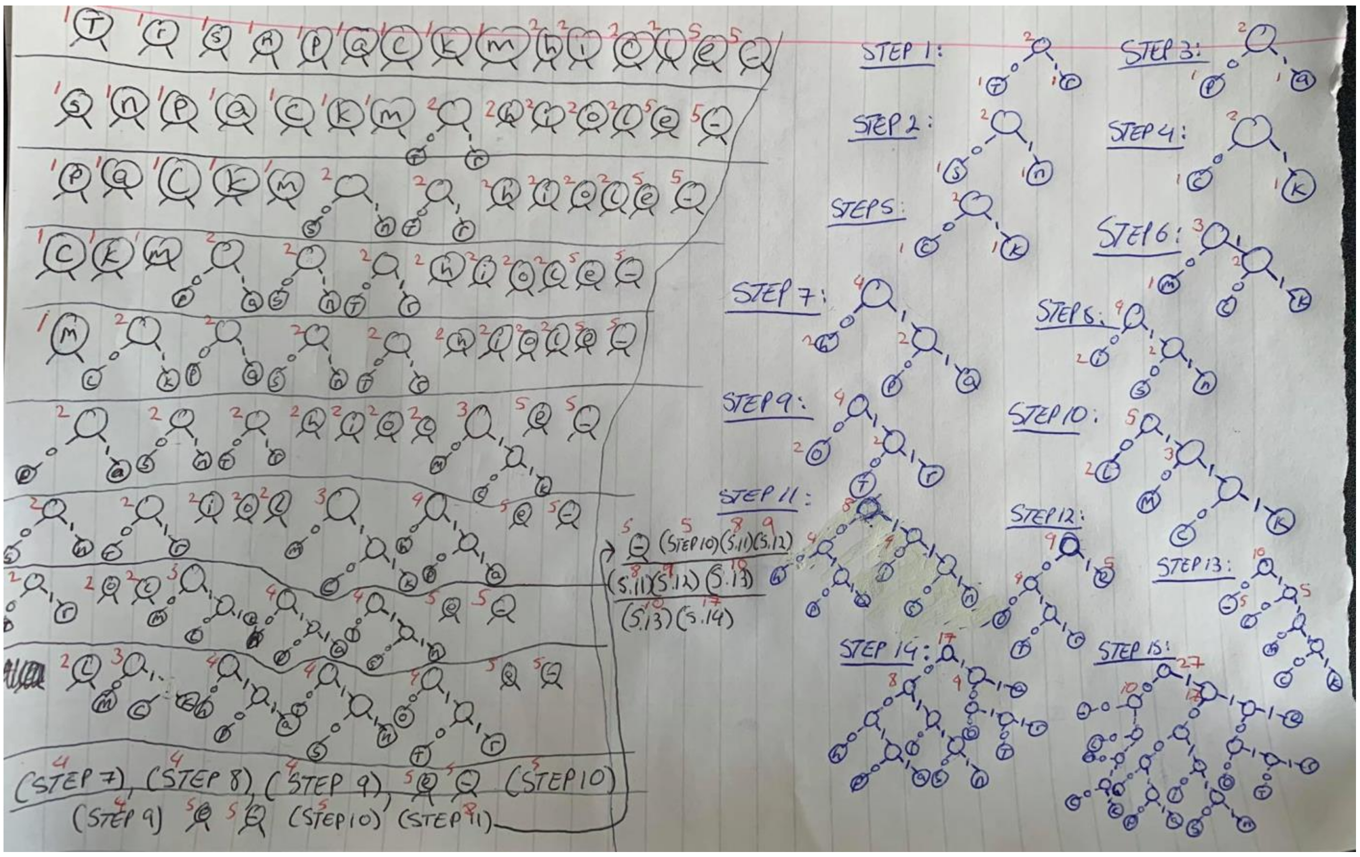
Task 1:

Frequency table:

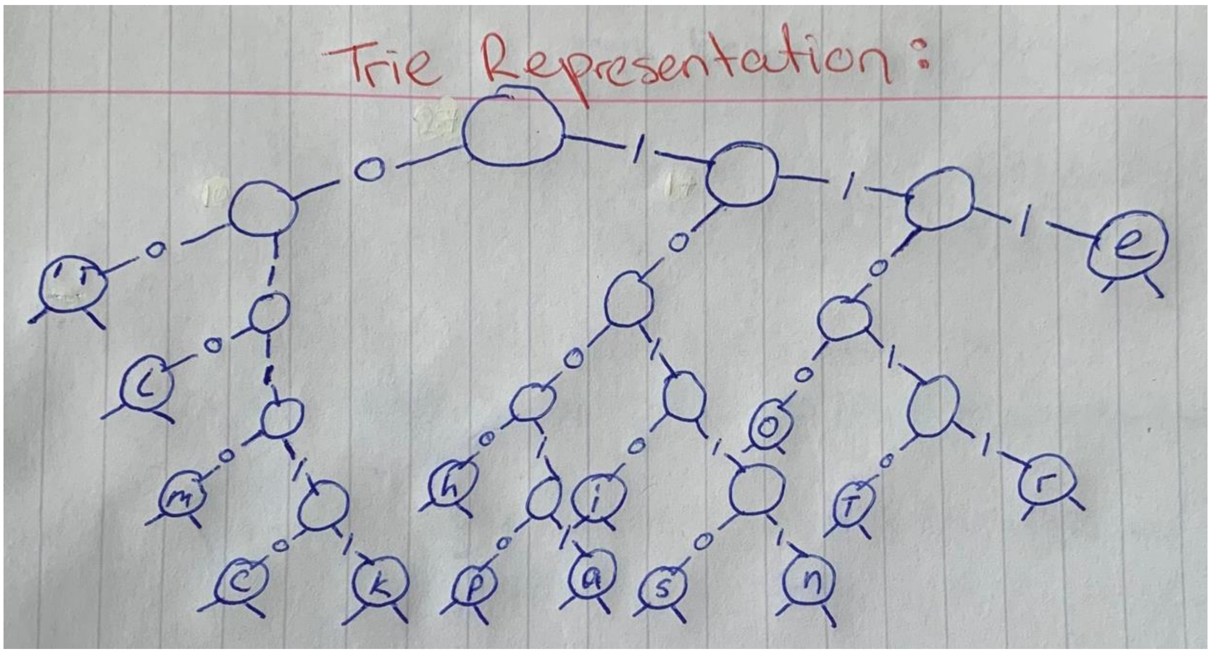


Char	Freq
J	1
h	2
e	5
r	1
i	2
s	1
n	1
o	2
p	1
l	2
a	1
c	1
k	1
m	1
' '	5

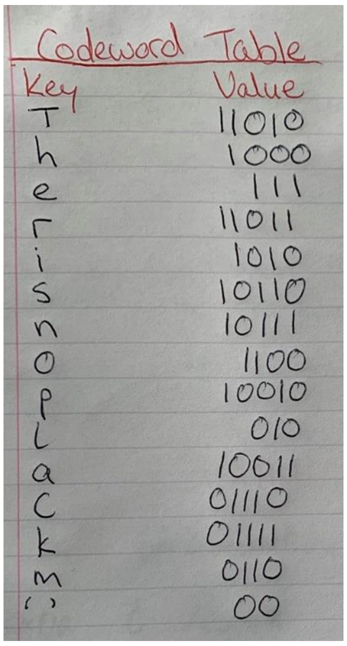
Rough work that I conducted to obtain the Huffman Tree:



Huffman Tree:



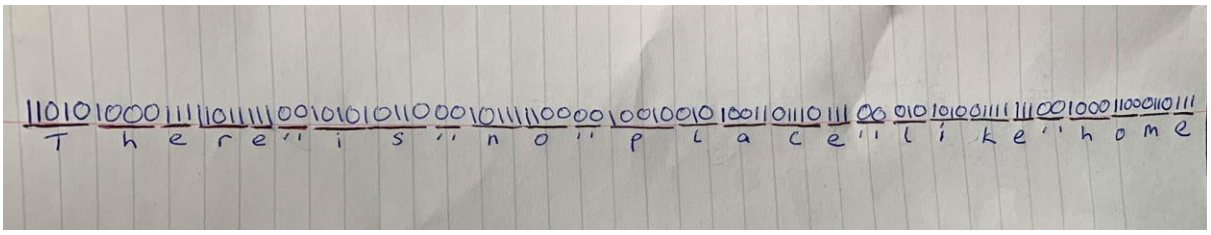
Codeword table:



Codeword Table

Key	Value
T	11010
h	1000
e	111
r	11011
i	1010
s	10110
n	10111
o	1100
p	10010
l	010
a	10011
c	01110
k	01111
m	0110
''	00

Compressed Bitstring: 99 bits



```
11010 1000 111 11011 111 00 1010 10110 00 10111 1100 00 10010 010 10011 01110 111 00 010 1010 01111 111 00 1000 1100 0110 111
T     h    e   r     e   '' i    s     '' n     o    '' p     l   a     c     e   '' l   i    k     e   '' h    o    m    e
```

Task 3:

File Name	**Time taken to compress (in nanoseconds)**			
	1	**2**	**3**	**Average**
genomeVirus.txt	14268700ns	14189700ns	13237600ns	13898667ns
medTale.txt	16532200ns	15321300ns	15719600ns	15857700ns
mobydick.txt	921276400ns	1143678700ns	1129062799ns	1064672633ns
q32x48.bin	2670201ns	2652600ns	2650400ns	2657734ns
beeMovieScript.txt	52631500ns	49513901ns	46060300ns	49401900ns

File Name	**Time taken to decompress (in nanoseconds)**			
	1	**2**	**3**	**Average**
genomeVirus.txt	4223400ns	4239600ns	4291100ns	4251367ns
medTale.txt	4782800ns	4528200ns	4770000ns	4693667ns
mobydick.txt	225691500ns	223350200ns	226911400ns	225317700ns
q32x48.bin	1762899ns	1878600ns	1718799ns	1786766ns
beeMovieScript.txt	15636599ns	16199800ns	15520901ns	15785767ns

SUMMARY TABLE						
File Name	**Original Bits**	**Compressed Bits**	**Decompressed Bits**	**Compression Ratio (rounded to 2 decimal)**	**Average compress time**	**Average decompress time**
genomeVirus.txt	50008	12576	50008	12576/50008 = 0.25	13898667ns	3161734ns
medTale.txt	45056	23912	45056	23912/45056 =0.53	15857700ns	3663067ns
mobydick.txt	9531704	5341208	9531704	5341208/9531704 =0.56	1064672633ns	225317700ns
q32x48.bin	1536	816	1536	816/1536 = 0.53	2657734ns	1786766ns
beeMovieScript.txt	401112	231976	401112	231976/401112 =0.58	49401900ns	15785767ns

The additional file that I chose to run the experiments on was the entire Bee Movie script. I ran the compress and decompress methods on all of the files three times, in order to find the average time that it takes for each file to be compressed/decompressed. The compression ratio for genomeVirus.txt is significantly better than the rest of the compression ratios – the file got 4 times smaller after compressing. This is because the file had a lot of repetitive chars. As you can see from the results above, it is clear that the bigger a file is, the longer it takes to be compressed. For example, mobydick.txt had a size of 9531704 bits and it took an average of 1064672633ns to be compressed. However, q32x48.bin had a size of 1536 bits and it only took 2657734ns to be compressed, which is significantly quicker.

From the results above, we can also conclude that it is quicker to decompress a compressed file, rather than to compress a file. This is because it takes longer to build a Trie (O(n + RlogR), where R is the alphabet size) rather than to read the Trie (O(n)). The above results were obtained by compressing each of the files and saving this compressed data into a new file (the command "java helper_code/HuffmanAlgorithm -<*originalFile* >

compressedFile" was used to do this). Then, BinaryDump was called on both the original file and on the new compressed file to obtain the size of each and therefore, calculate the compression ratio. Then, to find how long it would take to decompress each file, the decompress method was run on the compressed file and this decompressed data was saved into a new file (the command "java helper_code/HuffmanAlgorithm +< *compressedFile* > *decompressedFile*" was used to do this). Then, BinaryDump was called on the decompressed file to obtain the size – this size should be the same as the original file and as seen from the summary table above, this was true for all of the files.

I tried running an experiment to check what would happen if we tried to compress an already compressed file. I did this by calling the compress method on one of the already compressed files and calling BinaryDump on the result (the command "java helper_code/HuffmanAlgorithm -<*compressedFile.txt* | java helper_code/BinaryDump 8" was used to do this). When doing this, the size of the files seemed to increase. For example, the compressed genomeVirus.txt file has a size of 12576 bits but when we try to compress the compressed file again, it becomes 14896 bits. Similarly, the medTale.txt file becomes 25960 bits when trying to compress again. This is because when you have a file e.g. genomeVirus.txt, the uncompressed, original file just contains the original data. Then, when you compress it the first time using the Huffman Algorithm, the compressed file contains the Huffman Trie, Byte count and compressed data. Then, if you try to compress this compressed file using the Huffman Algorithm, it will contain the Huffman Trie, Byte count, Compressed Data of (Huffman Trie, Byte count, compressed data), Huffman trie, byte count and original compressed data. Therefore, if a file is fully compressed by using the Huffman Algorithm it will already have all redundancies removed, then if you try to compress it again it will get bigger.

I also conducted an experiment to compare the result of using the provided RunLength function to compress the q32x48.bin map to using my Huffman Algorithm to compress the map. When using the RunLength function to compress q32x48.bin, the compressed map is 1144 bits. However, when using the HuffmanAlgorithm function it is only 816 bits. This is because Run Length encoding only compresses consecutive chars which are the same. E.g. "aaabbbaa" will be compressed to "3a3b2a". However, with the Huffman Algorithm, all occurrences of "a" will be encoded to the same value that is assigned to "a". Therefore, Run Length encoding only works on files with repetitive, consecutive chars but this is not guaranteed in maps/images.