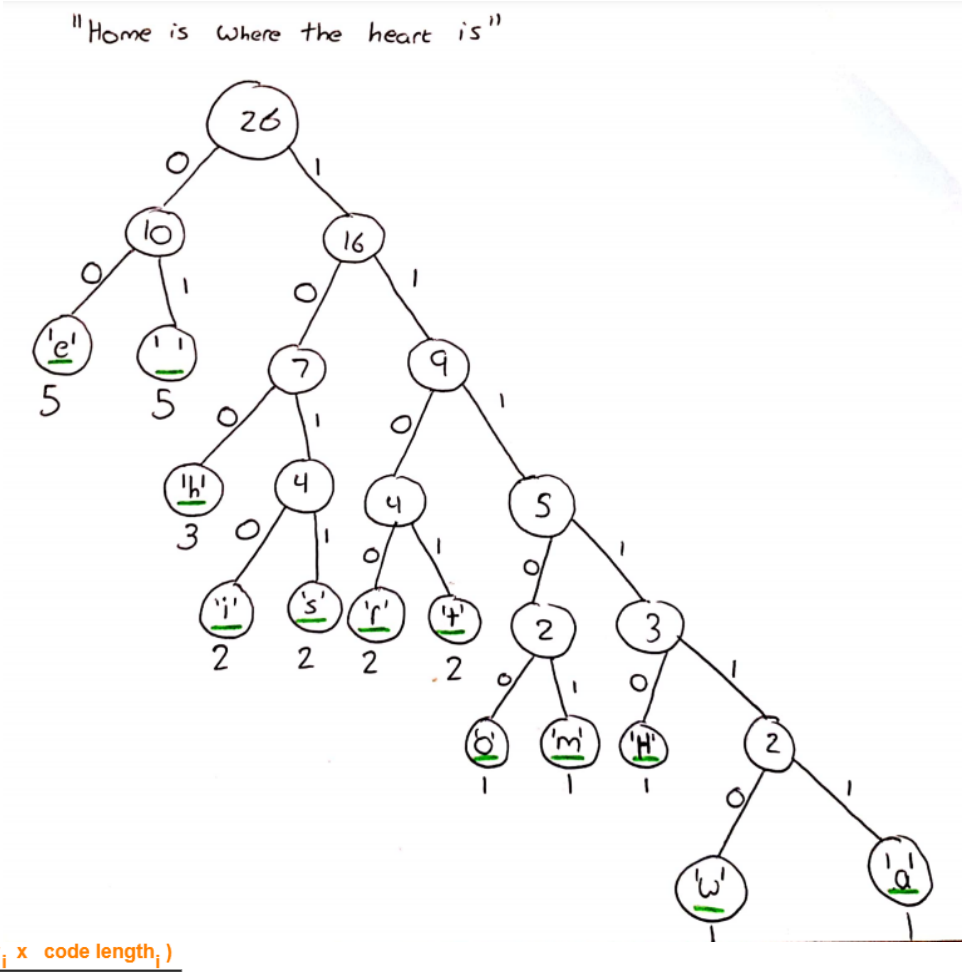
Huffman Compression-19476496

A huffman code is an optimal prefix code developed by david huffman. It is used in lossless compression.

Exercise 1: Drawing huffman tree for phrase “Home is where the heart is”(Note this is one of a list of valid huffman encodings depending on how is it built.)

Character	Frequency	Encoding
‘e‘	5	00
‘ ’	4	01
‘h’	3	100
‘i’	2	1010
‘s’	2	1011
‘r’	2	1100
‘t’	2	1101
‘H’	1	11110
’o’	1	11100
‘m’	1	11101
‘w’	1	111110
‘a’	1	111111



Average code length per character = $\frac{\Sigma\ (\text{frequency}_i \times \text{code length}_i)}{\Sigma\ \text{frequency}_i}$

$= \Sigma\ (\text{probability}_i \times \text{code length}_i)$

$$AvgCodeLength = \frac{9}{26} * 2 + \frac{1}{26} * 3 + \frac{8}{26} * 4 + \frac{5}{26} * 5 = 3Bits$$

Thus we can tell the avg amount of bits in a compressed text

$$TotalEncodedBits \approx frequency\ X\ codeLength = 3Bits * 26 = 78bits$$

Or we can just multiply frequency by individual codeLength since our text is small.

$$TotalEncodedBits = 9 * 2 + 3 * 3 + 8 * 4 + 5 * 5 = 84 \text{ bits}$$

Thus our compression ratio is $\frac{84}{26*8} * 100 = \frac{84}{308} * 100 = 40.3\%$

As such our Huffman encoding compressed the text quite a bit.
Exercise 2: Coding a huffman algorithm in java.

Much of the class is explained in the code however the code can be found inside the package HuffmanCompression in ‘src’

Note to run the file it must be in the form like this:

java Huffman compress InputFile OutputFile
java Huffman decompress inputFile OutputFile

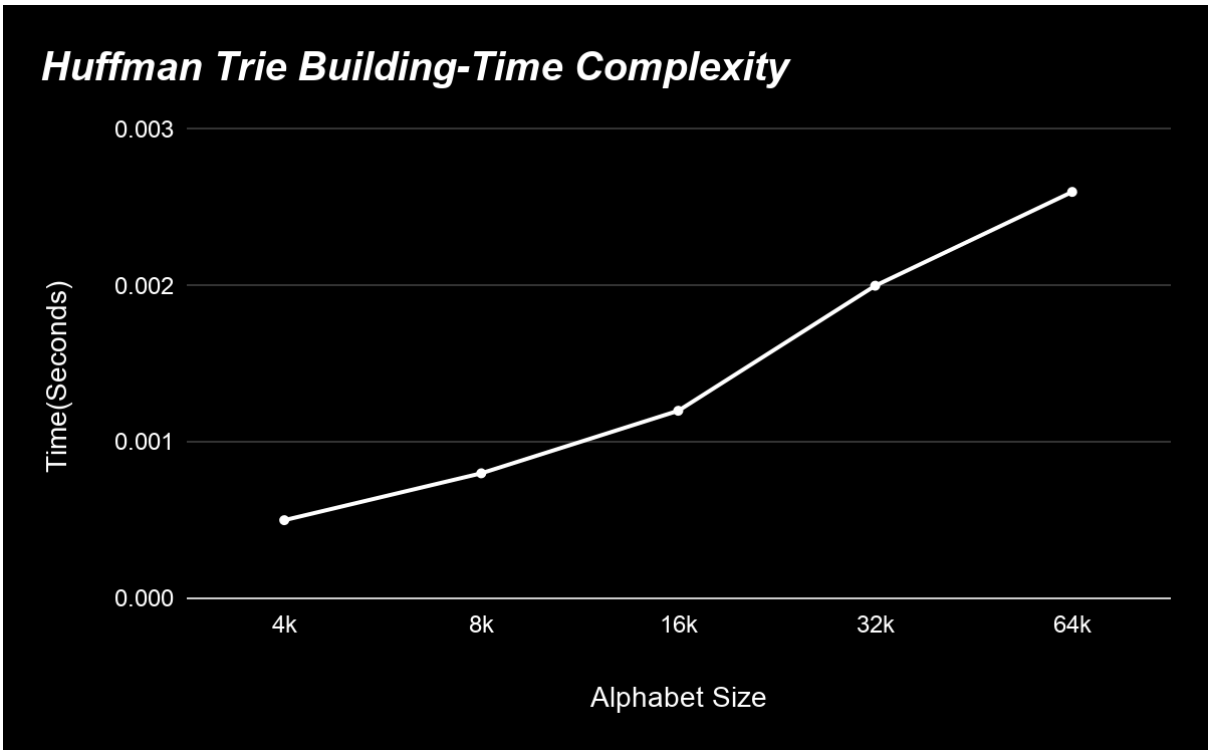
You should see on output similar to this:(make sure console is wide enough to display correctly)

```
C:\Users\redbr\Desktop\Compression>java Huffman compress mobyDick.txt compressedOutput
Compressing
      InputFile(Original)   OutputFile(Compressed)            Time Elapsed       Original bits      CompressedBits     Ratio
             mobyDick.txt         compressedOutput                0.125293             9531704             5341208     56.04%

C:\Users\redbr\Desktop\Compression>java Huffman decompress compressedOutput UncompressedMobyDick
DeCompressing
     InputFile(Compressed  OuputFile(Decompressed)        Time Elapsed       Original bits    DecompressedBits
        compressedOutput     UncompressedMobyDick            0.070126             5341208             9531704
```

Exercise 3: Results:

Input	Precompressed bits	Compressed bits	deCompressed bits	Compression ratio	Compress Time(S)	Decompress Time(S)
MobyDick.txt	9531704	5431208	9531704	56.04%	0.125	0.068
GenomeVirus.txt	50008	12576	50008	25.15%	0.011	0.003
medTale.txt	45056	23912	45056	53.07%	0.013	0.004
q32x48.bin	1536	816	1536	53.13%	0.007	0.001
test.txt	944	736	944	77.97%	0.006	0.001



Exercise 3 - 1 page of Analysis:

The time complexity for encoding and sending to the file is n + rlog(r).
Where r is the number of unique characters and n is total number of characters(For frequency Generation).

In the implementation given the building of the trie runs in O(n) due the minPQ needing to be filled in a loop of O(n) where n is given by ASCII_SIZE final variable.(This could be increased in speed by using a bottom up heap construction).

I graphed the function above by changing the ASCII_SIZE variable.

Analysis of Files given:

1. MobyDick.txt- This is quite a large file of 9million bits and such it takes quite a long time to compress, however the compression is still quite good at 54% due to the fact that the english language is not arbitrary hence giving many patterns of repetition to compress.
2. genomeVirus.txt- Genomes are only made up of a 4 character alphabet(ACTG), which are often quite evenly occurring. As such there is usually a rather balanced encoding list compared to that of english where letters like e might be skewed. This gives us a high compression ratio of 25.15%.
3. medTale.txt - Again similar to mobyDick, pretty good compression of 53% due to the patterns in english language
4. Q32x48.bin - Since this is a bitMap file it is made up of long strings and patterns of 0s and 1s meaning it can easily be read and converted giving a compression ratio of 53%.
5. Test.txt- This was a short file of english that I made without a lot of repetition to show that Huffman relies mostly on repetition and patterns in language given a compression ratio of 78% - higher than previous.

Analysis of decompression:

As seen from our decompression results we prove that huffman is a lossless algorithm as each time we decompress our file we get back the original amount of bits in the uncompressed files. We can also see that the running time is also much lower due to the fact that we do not need to generate the trie but only read it and convert the text. As such the speed is based on the size of the text.

Compressing an already Compressed file

Recompressing medTale - 108.56%. Recompressing mobyDick.txt - 98.54%
As we can see from our results compressing the file again only results in the file getting larger, or results in minimal compression. This is mainly due to the fact that huffman tries to read in the premade trie as ascii characters and, as we know the trie is made up of optimal prefix codes with no repeats. Hence there will not be much repetition since there are now more characters which are less frequent so we generate a larger trie.

RunLength vs Huffman Analysis

RunLength compression result: 74% - 1144 bits
Huffman compression result: 53% - 816 bits.
When working with bitmaps there are often long runs which are suitable for runLength, however in comparison to huffman runLength if we image a run of 00000001 which appears often in the file, runLength will need to generate a code for this each time it occurs, however since the huffman algorithm can read this in as an 8 bit ascii character it can store it using less bits. We see the result of this as huffman compresses the file about 20% better than runLength.