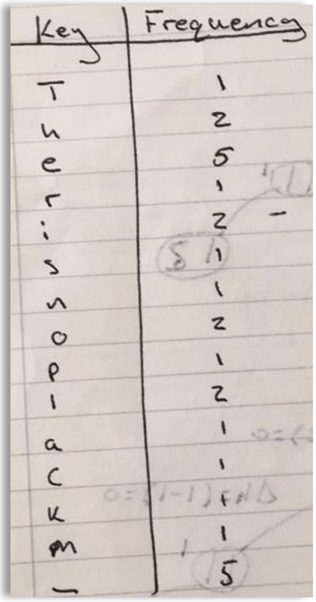
Kier McGuirk
18752609

Task One

Codeword Table

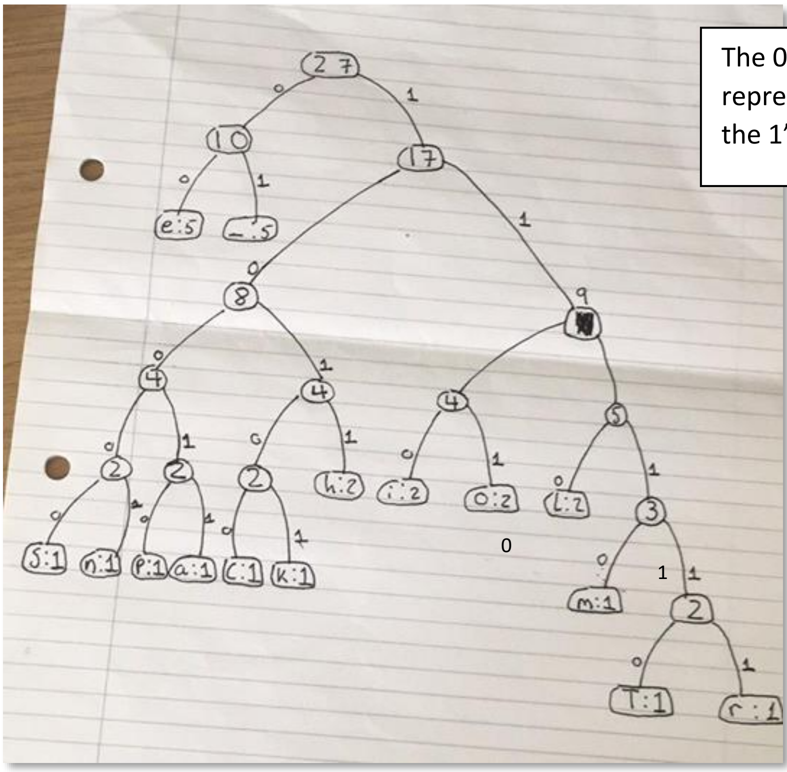
Key	Value	Frequency
T	111110	1
h	1011	2
e	00	5
r	111111	1
i	1100	2
s	10000	1
n	10001	1
o	1101	2
p	10010	1
l	1110	2
a	10011	1
c	10100	1
k	10101	1
m	11110	1
_	01	5

Hand Drawn Frequency Table



Key	Frequency
T	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

Trie Representation



The 0's to the left of each node represent that it is a left child and the 1's represent the right children

Kier McGuirk
18752609

Compressed Bitstring

With Spaces

111110 1011 00 111111 00 01 1100 10000 01 10001 1101 01 10010 1110 10011 10100 00 01 1110 1100 10101 00 01 1011 1101 11110 00

T h e r e _ i s _ n o _ p l a c e _ l i k e _ h o m e

99

Without Spaces

111110101100111111000111001000001100011101011001011101001110100000111101100101010001101111011111000

Kier McGuirk
18752609

Task Two

```
public static void decompress() {
    final long start = System.currentTimeMillis();
    helpDecompress();
    final long end = System.currentTimeMillis();
    System.out.println("The time taken to decompress: " + (end-start) + " ms");
    BinaryStdOut.close();
}
public static void helpDecompress()
{
    // read in Huffman trie from input stream
    Node root = readTrie();

    // number of bytes to write
    int length = BinaryStdIn.readInt();


    // decode using the Huffman trie
    for (int i = 0; i < length; i++) {
        Node node = root;
        while (!node.isLeaf()) //if not leaf
        {
            boolean bTrue = BinaryStdIn.readBoolean(); //get true/false from standard input
bitstream
            if(bTrue)
                node = node.right; //if true set as right
            else node = node.left; //if false set as left
        }
        BinaryStdOut.write(node.ch, 8); //write to standard out
    }
}
```

helpDecompress is a helper function, allowing for a neater way to time the decompression algorithm which is performed in decompress()

```
public static void compress() {
    final long start = System.currentTimeMillis();
    helpCompress();
    final long end = System.currentTimeMillis();
    System.out.println("The time taken to compress: " + (end-start) + " ms");
    BinaryStdOut.close();
}
public static void helpCompress()
{
    /// read the input
    String s = BinaryStdIn.readString();

    char[] sChars = s.toCharArray();

    // tabulate frequency counts
    int[] fArr = new int[R];
    for (int i = 0; i < sChars.length; i++)
    {
        fArr[s.charAt(i)]++;
    }

    // build Huffman trie
    Node root = buildTrie(fArr);

    // build code table
    String[] sArr = new String[R];
```

HelpCompress is a helper function, allowing for a neater and more concise way to time the compression algorithm.

Kier McGuirk
18752609

```java
    buildCode(sArr, root, "");

    // print trie for decoder
    writeTrie(root);

    // print number of bytes in original uncompressed message
    BinaryStdOut.write(sChars.length);

    // use Huffman code to encode input
    for (int i = 0; i < sChars.length; i++)
    {
        String code = sArr[sChars[i]];
        for (int j = 0; j < code.length(); j++)
        {
            if (code.charAt(j) == '0')       //false if 0, true if 1
                BinaryStdOut.write(false);
            else if (code.charAt(j) == '1')
                BinaryStdOut.write(true);
            else throw new IllegalArgumentException("Unacceptable command, state - for
compression + for decompression");
        }
    }
```

Algorithm can be found in the repository: Assignment -> helper_code -> HuffmanAlgorithm.java

Kier McGuirk
18752609

Task Three

Summary Tables

File Name	Compression Time (ms)						Decompression Time(ms)					
	1	**2**	**3**	**4**	**5**	**AVG**	**1**	**2**	**3**	**4**	**5**	**AVG**
genomeVirus.txt	13	13	15	13	14	14	10	10	9	10	10	10
medTale.txt	15	14	15	16	14	15	10	10	10	10	10	10
Mobydick.txt	3042	3050	3057	3077	3011	3048	279	310	248	323	253	283
Q32x48.bin	7	6	8	8	6	7	2	3	2	3	3	3
Csw.txt	15	17	16	17	16	16	3	3	3	3	4	3

File Name	Binary Dump			Hex Dump			Compression Ratio
	Uncompressed	**Compressed**	**Decompressed**	**Uncompressed**	**Compressed**	**Decompressed**	
genomeVirus.txt	50008	12576	50008	50008	12576	50008	0.25
medTale.txt	45056	23912	45056	45056	23912	45056	0.53
Mobydick.txt	9531704	5341208	9531704	9531704	5341208	9531704	0.56
Q32x48.bin	1536	816	1536	1536	816	1536	0.53
Csw.txt (my chosen file)	68248	34160	68248	68248	34632	68248	0.5

What happens if you try to compress one of the already compressed files? Why do you think this occurs?

Compressing one of the already compressed files causes the file to get larger. The command java helper_code/HuffmanAlgorithm -< data/genome_virusCompressed.txt | java helper_code/BinaryDump resulted in 1736 bits – greater than the already compressed file. This is because compression algorithms are unable to make a file smaller, every single time. If so, this would mean that a file could be repeatedly compressed, so that it reaches close to 0 bytes of storage whilst still retaining its data, which is an absurdity. Trying to compress already compressed data with the same algorithm will result in the algorithm trying to create a Trie structure and code table to encode the already compressed data which will often result in increased file size.

Kier McGuirk
18752609

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?

Compression Algorithm	File Size		Compression Ratio
	Uncompressed	Compressed	
Run Length	1536	816	0.53
Huffman	1536	1736	1.13

It can be seen that the Huffman Algorithm is far superior in compressing the q32xq48.bin file. In fact, Run Length even enlarged the file. This is because Run Length is not a compression algorithm for ASCII text and performs much better with sequences of 1s and 0s, like Bit Maps

Kier McGuirk
18752609

Analysis

Compression

Compression Ratio

The best performing compression was the compression of “genomeVirus”. It had the most superior compression ratio compared to the other files. This is probably due to the fact that the genome is a long sequence of repeated characters. This means that fewer characters need to be encoded and represented in the Trie structure and the Code Table, which is indicative of a good compression ratio.

The worst performing was the compression of “mobydick.txt”. This is to be expected – Moby Dick is a novel and would have far more combinations of words and characters than any of the other files, meaning a larger Trie structures and coding table.

Med Tale had a similar ratio to Moby Dick, which is unsurprising, because they are both written texts. Med Tale is slightly smaller because Moby Dick would have more combinations of characters and words due to it being a much larger text

CSW (the file I chose which is all the words in the English language that start with ‘A’) also had a similar ratio to the aforementioned files. This is again, due to similar reasons being that it is a text file of words from the English language. It has slightly less of a compression ratio, probably due to the fact that it has repetitive beginning of words (it is in ascending order), which would provide more efficient implementation of the tree structure.

Compression Sizes

Regardless or compression ratio, it was clear that the bigger the file would result in ‘more’ storage space saved. In other words, Moby Dick had the worst compression ratio, but because it was the biggest file, it had the most data to compress, meaning that it “saved” the most physical storage space.

Compression Times

The time to compress would essentially be the time it takes to encode each character and to populate the Trie.

There was a clear relationship between file size and compression time. This is unsurprising due to the fact it would take longer for the algorithm to encode every character in the file. However, it can also be seen that due to the fact that CSW is ordered alphabetically it was compressed much faster than a other txt files like Med Tale that would have disordered series of characters and words, since it would be more efficient to implement the Trie structure this way.

The average of 5 run times was calculated – there wasn’t much fluctuation in runtimes per iteration which is to be expected.

Kier McGuirk
18752609

Decompression

Decompression Sizes

Huffman is a lossless compression technique, meaning that all of the files were restored to their original file size once decompressed.

Decompression Time

The time it takes to decompress the files is contingent on the Trie size and the size of the file. This is reflected well because the bigger the Trie (for example in Moby Dick, there would be more characters -> bigger trie) and the bigger the file size (Moby Dick being a novel) the longer the file will take to decompress.

Moby Dick had the longest decompression time which is expected, given the aforementioned reasons.

My csw file had a very quick decompression time, which is indicative that the Trie was efficiently implemented.

It could also be seen that files with similar compression times will share similar decompression times.

The average of 5 run times was calculated – there wasn't much fluctuation in runtimes per iteration which is to be expected.

Commands

Finding Num Bits

```
java helper_code/BinaryDump  < Data/genomeVirus.txt
java helper_code/BinaryDump   < Data/medTale.txt
java helper_code/BinaryDump   < Data/q32x48.bin
java helper_code/BinaryDump < Data/mobydick.txt
java helper_code/BinaryDump < Data/csw.txt
java helper_code/HexDump  < Data/genomeVirusCompressed.txt
java helper_code/HexDump   < Data/medTaleCompressed.txt
java helper_code/HexDump   < Data/q32x48rle.bin
java helper_code/HexDump < Data/mobydickCompressed.txt
java helper_code/HexDump < Data/cswCompressed.txt
```

Time to Compress / Decompress

```
java helper_code/HuffmanAlgorithm -< Data/genomeVirus.txt
java helper_code/HuffmanAlgorithm -< Data/medTale.txt
java helper_code/HuffmanAlgorithm -< Data/q32x48.bin
java helper_code/HuffmanAlgorithm -< Data/mobydick.txt
java helper_code/HuffmanAlgorithm -< Data/csw.txt
java helper_code/HuffmanAlgorithm +< Data/genomeVirusCompressed.txt
java helper_code/HuffmanAlgorithm +< Data/medTaleCompressed.txt
java helper_code/HuffmanAlgorithm +< Data/q32x48rle.bin
java helper_code/HuffmanAlgorithm +< Data/mobydickCompressed.txt
java helper_code/HuffmanAlgorithm +< Data/cswCompressed.txt
```

Create Compressed File

```
java helper_code/HuffmanAlgorithm  -<  Data/q32x48.bin > Data/q32x48rle.bin
java helper_code/HuffmanAlgorithm  -<  Data/genomeVirus.txt > Data/genomeVirusCompressed.txt
java helper_code/HuffmanAlgorithm  -<  Data/mobydick.txt > Data/mobydickCompressed.txt
java helper_code/HuffmanAlgorithm  -<  Data/medTale.txt > Data/medTaleCompressed.txt
java helper_code/HuffmanAlgorithm  -<  Data/csw.txt > Data/cswCompressed.txt
```