# Algorithms Assignment 2 – Huffman Coding

## Student Number: 21738885

### Task 1 – Huffman Tree

Here we create a Huffman tree with the given phrase: “Home is where the heart is”.

First, we have our table of characters and each one’s frequency.

| **Characters** | **Frequency** | **Bit Representation** |
|---|---|---|
| H | 1 | 1100 |
| o | 1 | 11100 |
| m | 1 | 11101 |
| e | 5 | 01 |
| i | 2 | 1001 |
| s | 2 | 1010 |
| w | 1 | 11110 |
| h | 3 | 1000 |
| r | 2 | 1011 |
| t | 2 | 1100 |
| a | 1 | 11111 |
| -(space) | 5 | 00 |

Next up the characters are lined up in descending order and the tree is built up by merging the 2 lowest frequency nodes each time

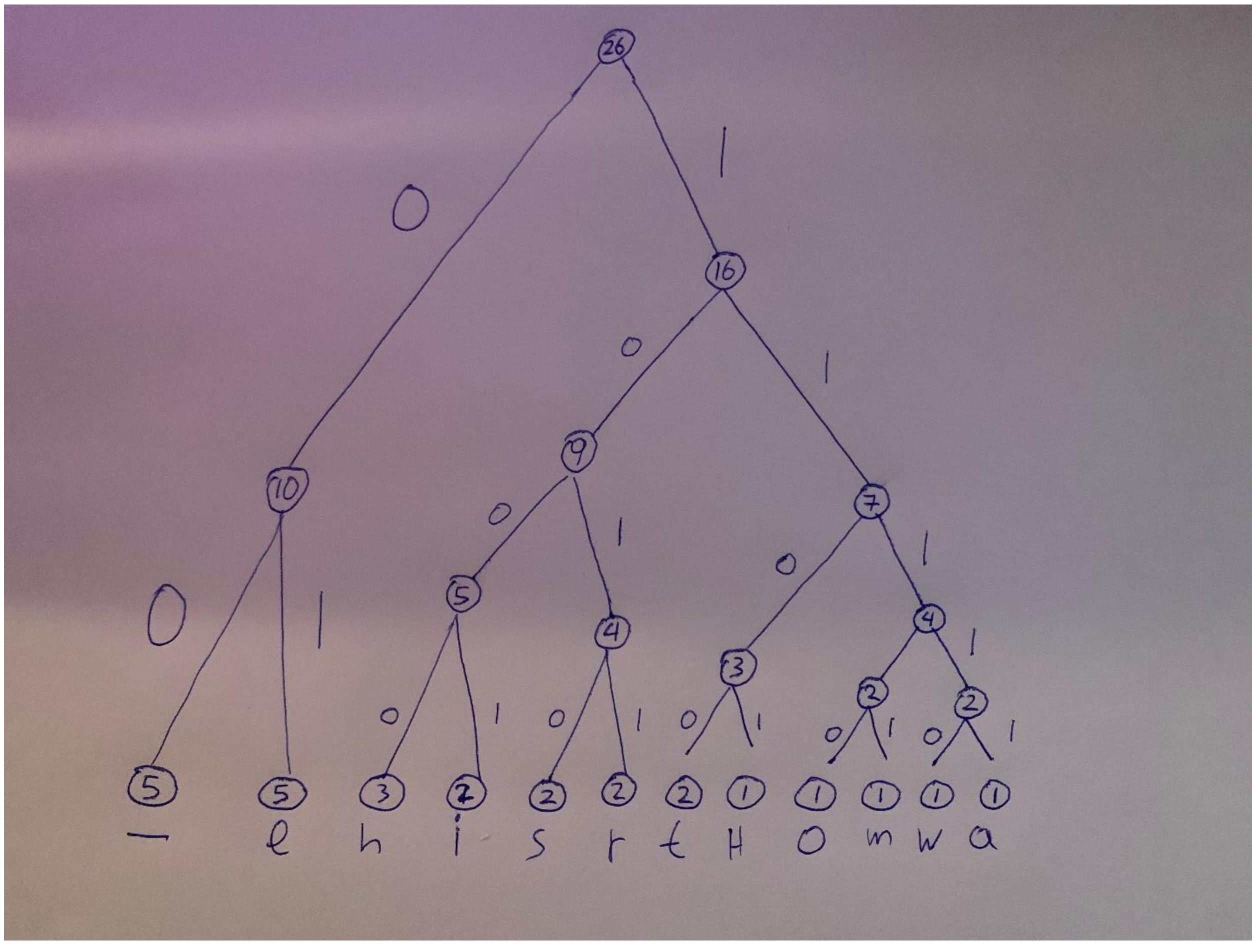


*Figure 1:Huffman tree of given phrase.*

By placing 0 on the left node and 1 on the right node we can write the bit representation which is found in the above table.

We see that the bigger the frequency the smaller the bit representation, and vice versa.

Coded phrase:

“**Home** “: 110111100111010100 = 18 bits

“**is** “: 1001101000 = 10 bits

“**where** “: 1111010000110110100 = 19 bits

“**the** “: 110010000100 = 12 bits

“**heart** “: 100001111111011110000 = 21 bits “**is**“: 10011010 = 8 bits

**Encoded:**

“**Home is where the heart is**”:
11011110011101010010011010001111010000110110100110010000100100001111111011111000010011010 = **88 bits**

**Original:**

“**Home is where the heart is**”: 26 characters x 8 bits = **208 bits**

**Compression ratio = compressed size / original size =**

**88 bits /208 bits = 42% Compression ratio.**

## Task 2 – Huffman Implementation.

To run Huffman class, change terminal directory to src

Then Compile all classes using: javac  *.java

For compression: java Huffman - <filename.txt> compressed.txt

For decompression: java Huffman + <filename.txt> decompressed.txt

**Note**: I have imported a java util import Priority Queue instead of the MinPQ.java file by Robert Sedgewick and Kevin Wayne, therefore there is no file dependency on it.

## Task 3 – Analysis and Comparison of Compression

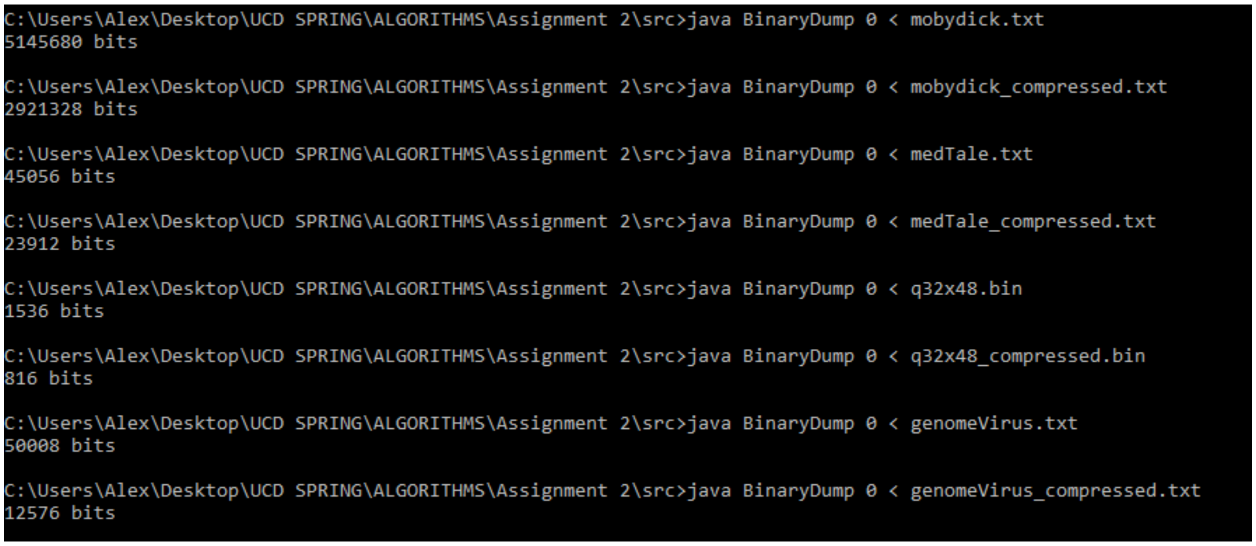
### Q1 – File Compression.

| **Input File** | **Output File** | **Original Bits** | **Compressed Bits** | **Compression Ratio** |
|---|---|---|---|---|
| mobydick.txt | mobydick_compressed.txt | 5145680 bits | 2921328 bits | **56.77%** |
| medTale.txt | medTale_compressed.txt | 45056 bits | 23912 bits | **53.07%** |
| q32x48.bin | q32x48_compressed.bin | 1536 bits | 816 bits | **53.12%** |
| genomeVirus.txt | genomeVirus_compressed.txt | 50008 bits | 12576 bits | **25.15%** |

### Notes:

- **File Size counted with command java BinaryDump 0 < filename.txt**
- **Compression Ratio calculated by: Compressed Bits / Original Bits.**

**For ease of reference:**



```
C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < mobydick.txt
5145680 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < mobydick_compressed.txt
2921328 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < medTale.txt
45056 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < medTale_compressed.txt
23912 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < q32x48.bin
1536 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < q32x48_compressed.bin
816 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < genomeVirus.txt
50008 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < genomeVirus_compressed.txt
12576 bits
```

*Figure 2: Console commands used to find size of files.*

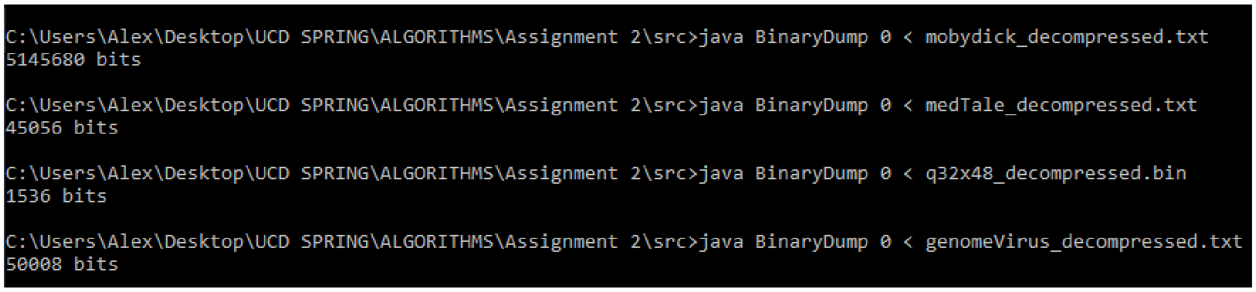
## Q2 – File Decompression.

| Input file | Output file | Decompressed bits |
|---|---|---|
| mobydick_compressed.txt | mobydick_decompressed.txt | 5145680 bits |
| medTale_compressed.txt | medTale_decompressed.txt | 45056 bits |
| q32x48_compressed.bin | q32x48_decompressed.bin | 1536 bits |
| genomeVirus_compressed.txt | genomeVirus_decompressed.txt | 50008 bits |

## Notes:

- **Decompressed size is the exact same as original size, meaning there was no data loss.**

**For ease of reference again.**



```
C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < mobydick_decompressed.txt
5145680 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < medTale_decompressed.txt
45056 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < q32x48_decompressed.bin
1536 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < genomeVirus_decompressed.txt
50008 bits
```

*Figure 3: Console commands used for finding decompressed bits.*

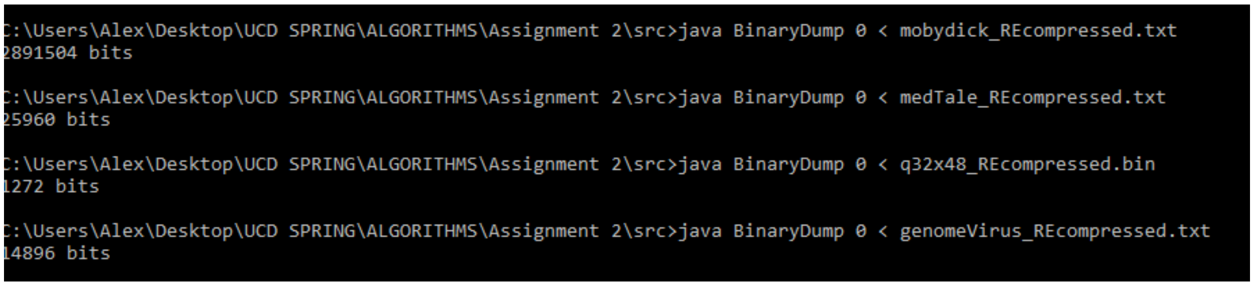
## Q3 – Analysis of results and Double Compression.

### Analysis of above:

- The Huffman algorithm for compression seems to be very effective; it reduced the size of all files by almost half with the **highest compression ratio** being **56.77%** and in the case of the genomeVirus.txt, it is reduced to almost ¼th the size with **25% compression ratio (being the lowest).** This is due to the nature of this file of course, which contains only 4 repeating characters: ‘**G**’,’**A**’,’**T**’,’**C**’.
- The decompression is also effective as the **decompressed bits were equal to our original file’s bits**, meaning no data was lost.
- The compression ratio seems to depend on the variance of characters used in each file rather than the actual size of the file, meaning **higher use of characters** gives a **higher compression ratio** in comparison to **lesser use of characters** which **reduces compression ratio**.

## Double Compression.

| Input file | Output file | recompressed bits |
|---|---|---|
| mobydick_compressed.txt | mobydick_REcompressed.txt | **2891504 bits** |
| medTale_compressed.txt | medTale_REcompressed.txt | **25960 bits** |
| q32x48_compressed.bin | q32x48_REcompressed.bin | **1272 bits** |
| genomeVirus_compressed.txt | genomeVirus_REcompressed.txt | **14896 bits** |



```
C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < mobydick_REcompressed.txt
2891504 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < medTale_REcompressed.txt
25960 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < q32x48_REcompressed.bin
1272 bits

C:\Users\Alex\Desktop\UCD SPRING\ALGORITHMS\Assignment 2\src>java BinaryDump 0 < genomeVirus_REcompressed.txt
14896 bits
```

*Figure 4:Finding recompressed bits.*

After recompressing all our files, we get some interesting results.

For readabilities sake ill outline the differences.

**mobydick.txt**: **original** = **5145680** bits, **compressed** = **2921328** bits, **recompressed** = **2891504** bits.

**medTale.txt**: **original** = **45056** bits, **compressed** = **23912** bits, **recompressed** = **25960** bits.

**q32x48.bin**: **original** = **1536** bits, **compressed** = **816** bits, **recompressed** = **1272** bits.

**genomeVirus.txt**: **original** = **50008** bits, **compressed** = **12576** bits, **recompressed** = **14896** bits.

Analysis:

What we see is that in pretty much all cases, with the exception of mobydick.txt,

Recompressing will actually have a negative result, making the file size bigger instead of smaller this is the most evident in **q32x48.bin** with a compression ratio of **82%** which is horrible in comparison to our original **53%** (**a 30% increase!!!**).

It is likely this happens as a result of re encoding the text that is already encoded, meaning it will re encode the symbols we are already using to encode the original file, therefore growing the Huffman tree as we now have a higher use of characters, recompressing will also probably corrupt our file and result in losing our original data.

## Q4 – RLE vs HUFFMAN.

| **Algorithm** | **Input File** | **Output File** | **Bit size** |
|---|---|---|---|
| **Huffman** | **q32x48.bin** | **q32x48_compressed.bin** | **816 bits** |
| **RLE** | **q32x48.bin** | **q32x48_RunLengthCompression.bin** | **1144 bits** |

We see here that the Huffman Algorithm has a compression ratio of **53.12%**,

Where as RLE has a compression rate of **74.5%!!!**

The **Huffman** algorithm is by far the **better** option as its compression ratio is **20% less**, this is because Run Length is only efficient for data with large sequences of repeating characters, however our Huffman algorithm can very effectively compress data with high frequencies especially if there is low use of characters regardless of if there are repeating sequences or not (**Higher frequency** means **smaller code representation** as seen in **Task 1**, which is **ideal in this case**).