Huffman Coding

For Huffman coding we first take the whole block of text and split it up into a long list of characters, this includes the spaces and grammar, we then take this list of characters and count how many of each character exist within the text. This is then used to create a binary tree, we take the two lowest pairs of characters and attach those two nodes to a parent node with a value equal to the character count of the first and second node. This is then continued whilst readding the nodes that we create and hence pairing them off too until we are left with one root node. This root node will then be are start point, from here we can both encode and decode the text we have been given. In order to encode the text, we will start from the root of the tree and take the branches down too the character we are looking for, giving us a 1 when we take a right branch and a 0 when taking a left branch, although we can swap the values of the left and right branch for different trees it is import that in the given tree we keep the value of the left and right branch consistent. Once you reach the character you will have the encoding for it in binary form, for example, 101, or any other binary function. This is now a much more compressed version of text, as although some characters maybe over the usual value of 8 bits that they are encoded at usually, most of the characters are below this value and most importantly the most used characters have the lowest bit count and hence are much more compressed than before. Then to decode the text we will need to have a saved version from the tree and we simply take our long encoded file as a string and start from the first 0 or 1 going down the tree and every time you get to a character that will be the next character in the message, then you start again at the top, looping through this until you reach the end of your encoded message and hence have a decoded message at the end.

Strengths:

- Is the most efficient way of encoding single characters, as proven by Huffman.
- Once encoded it is extremely quick to decoded as you simply follow down the tree
- Easy to implement in many programming languages and requires very little core data structures to get it too work.
- Lossless compression and hence is reversible to the exact input file.

Weaknesses:

- Can be very computationally expensive to create the first encoding as you must loop through every character and count them before encoding them hence if the file is very large you need to process a huge data set twice to perform the encoding.
- Required to keep the tree for each specific set of data in order to decode it otherwise the encoded output is effectively useless.

Dictionary-Based Encoding

Dictionary based encoding is quite simple what you do initially is split up every word in the block of inputted text and we then take all these words and give them a unique id in a dictionary. With this we can then encode the text by assuming that every word has a space between them since we have split the string at spaces we can simply create an encoding by giving a numerical value for every word. Then too encode the document we convert words and spaces to just a long list of numbers with spaces between them which will highly decrease the character count in a document and hence create a much more efficient encoding of the document. This differs from the Huffman coding method as instead of continually looping through the text and encoding it only needs to split at every space and hence only requires one run through of the initial document to create the encoding dictionary and another to actually encode the document, it also only requires a dictionary-based data structure as opposed to the binary tree and linked lists required of the Huffman coding technique. To decode we simply take the dictionary we created earlier and compare the values we are given in the encoded document to this dictionary writing down the words we get out of it keeping the spaces in the originally encoded document.

Strengths:

- Is very efficient when analysing many of the same texts, as you can create a shared dictionary and hence if you have a company with many documents of similar content this method can be extremely quick.
- Is also lossless and hence all the data is preserved.
- When compressing large text files such as books where there are plenty of repeated words then the performance is much better as the dictionary will contain plenty of words already in the list.
- When using large text files much less computationally expensive than the Huffman Coding as it only needs to go through the file once to create the encoding and is not required to count anything.

Weaknesses:

- Not great with punctuation as they will be either included with the words and hence add a lot of duplicate words or it will be put separately alike the spaces and left un-encoded hence adding to the size of the file.
- Since the spaces between words don’t get encoded since doing so would create confusion as too where one number ends and another starts, the performance for larger texts will always be limited due to a high number of spaces.
- When encoding you simply replace text with numbers rather than a binary representation like Huffman coding hence the performance will be limited in its effectiveness.
- Will be very limited in uses when used in languages other than English as many languages such as French have gendered words and hence the dictionary would be far larger and less useful for compressing text.

Relative Lempel-Ziv Encoding

Lempel-Ziv encoding is a method that takes after old code book like approaches to sending messages where the sender and receiver would have a same shared code book of common phrases they can use to compress their messages to just a few letters or one word. This obviously had the downfall of being only select phrases and hence you could not send anything unique. Lempel and Ziv suggested using a different method where the code book is created on the fly with the encoded message and hence you won’t need to send the code book across as you will be able to decode the message by creating the same code book. This is done by scanning across the inputted string and every time you come across a letter or pattern you haven’t encountered before it will mark this as the first time it has been seen, this is usually in a hash map of letters and their codes. Then you move on and when you find the same letter you will start a sequence from that letter and stop when it finds a new letter creating a sequence. When encoding the message it will place encode letters to their same value the first time they are seen and then every other time it will be encoded as it’s place in the code book. This then works with any sequence as it will create first the numerical values of the characters already seen and then the character that wasn’t already found, this then will be saved as another pattern, for example, AAB, would come out as, A1B, with the encoding being A = 1 and AB = 2. To decode we simple follow this process backwards starting at the first character and create our own code book which will help decode the encoded data. This is very different from Huffman and Dictionary-Based encoding as it doesn’t require the encoding to store and send the binary tree or dictionary along with the encoded message hence cutting down greatly on the size of the data needed to be sent. It also means that the raw file contains all that is needed to decode the message and hence is a great standalone method for encoding files especially when they are longer containing similar patterns.

Strengths:

- When sending data you don't need to attach a dictionary or binary tree and hence can be decoded by any user without prior knowledge of the data.
- When encoding large blocks of continuous text it is very useful as it will end up with a similar sized approach too the dictionary based of several unique words but it will also contain many smaller patterns that are commonly used within many words.
- Is very useful when it comes to medium sized text files as it will still give a strong encoding of the message without a large computational cost to the user.

Weaknesses:

- When encoding smaller messages it makes very little difference especially if it contains a variety of letters and patterns the message will be very close to the inputted message.
- If the user wants to decode a large encoded message it can take much longer as you must both produce a large code book and then look back at the code book whenever decoding messages.
- When using spaces and punctuation it can create lots of smaller sequences with the spaces and punctuation that are only required once or twice and hence very expensive.

<u>List of all data structures and algorithms used in my implementation;</u>

- Array's: This was used in only one case, when generating the two lowest nodes in the system I would then place them into an array of size two and return them. I used this as I know it will always be an array of size two, although the two elements dynamically change as we load every node in therefore, we won't use a tuple here and hence this is the most efficient structure for just returning two values.
- Hash Tables: I used hash tables in order to hold the value of each character's encoding, this is due to the fact when encoding the values it is very efficient to use a hash table to look up the character as a key and then the encoding as their value. As well as using a hash table when splitting a string up into its individual characters and saving them as the key with the number of times that character occurs as it's value.
- Linked Lists: Again only used in one specific time in the implementation where we are sorting the characters by their values and clumping together those that have the same value before creating nodes for them. Although not strictly needed by doing this we can improve efficiency by starting from the first two nodes we will already have the smallest two and hence if we have a lot of nodes it won't take a long time to find the smallest in the list. Although in the case of sorting all the current nodes it would've possible been a better idea to use a stack, I decided as when there are lots of similar values it becomes more difficult to keep the stack consistent hence using a linked list allows an easier albeit more computationally heavy method for sorting the nodes.
- Binary Tree: This was used for the bulk of the Huffman Coding as every letter is modelled as a node on the tree. This means that the binary tree consists of all the nodes for letters as well as some for places where the node is simple an aggregate score of the two nodes below it. This was the simplest option for data structure despite in my implementation having to use an external class to setup the nodes it is for less computationally expensive and when decoding the Huffman Coding it is much more efficient to use a binary tree as we simply follow the branches using the encoding we have and stop when we are at a letter.
- Merge sort: Although not actually the proper implementation of merge sort as I am not outputting a sorted list rather a sorted binary tree, we split the lowest two and pair them and continually do that until the entire tree is sorted, I use this method instead of the raw merge sort as some data points in the sort are actually a combination of two points below them and therefore if using a pure merge sort we wouldn't be able to do a Huffman Coding but by merge sorting each branch we can quickly create a tree. Although quick sort may be

applicable here it is better to define this as a merge sort since we are separating the list of nodes into smaller two item arrays and then working back to a sorted binary tree.

Progress Log

Day 1:

- Research into the Huffman Coding technique using online videos and resources to create a better picture of what is required.
- Draw out a simple Huffman binary tree and work out what is required to create a Huffman technique.

Day 2:

- Create java file for the initial creation of the methods for sorting the text and creating hash maps of the outputted text with the characters and their count.
- Adding the has maps to sorted lists and creating an initial test of the method for creating a binary tree. Failed due to improper node setup for the tree (attempting to make the nodes inside the initial class)

Day 4:

- After a day of researching and writing on compression methods, came back with a better idea of what to implement.
- Creating two more java files one for the nodes within the tree and another for the driver of the project.
- Adding a method to find the two lowest nodes and then combine them into a new node.
- First successful output of a Huffman encoded text string, although current only outputs the actual encoding for each letter.

Day 5:

- Creating a method to import from the input.txt file hence allowing much larger text files to be simple loaded into the method.
- Added a method for encoding the input text and giving it as a set of zeros and ones.
- Created a method for outputting the text version of the encoding to a file, needs to be changed to output the actual binary to a file.

Day 7:

- Starting work on a user interface, simple text inputs, to allow the user to encode and then decode the message without it getting simply stopping the program.
- Looked into how I should save the actual binary tree, or at least the codes for each letter so I can decode any Huffman Coding.
- Create a better output so it actually writes bytes to a binary file.

Day 8:

- Created a method to decode the file from the generated binary tree, will only work on a file I am currently working on.

Day 9:

- Created a decoding method to allow decoding of the file's contents.
- Added a header to the encoded file so I can keep the encodings of each character in there.

- Finally added the ability to properly decode any encoded file made and then output that too a text file.

<u>Day 10:</u>

- Cleaned up the file removing any unused imports.
- Worked on the menu and input for the user allowing them to both encode and decode files as well as general flow between screens with a quit function.
- Completed tests on the encoding with different data sets (see below).
- Remade all for loops adding to strings with string builders, times slashed from 30+ minutes to 30 milliseconds, this was found after testing where the slowest part of the build was and researching how strings are hugely inefficient when using += notation.

<u>Day 11:</u>

- Worked on git hub repository for the code with instructions for use in the readme.
- Record video using the system, although the decoding seems to lose data as the size decreases of the file after decoding this is due to lost of some blank space within the file which is largely unavoidable

<u>Performance Analysis:</u>

Books via project Gutenberg's online collection, the byte sizes given are taken from the document's property panel on file explorer and is the documents size as opposed to its size on disk.

<u>Table for books:</u>

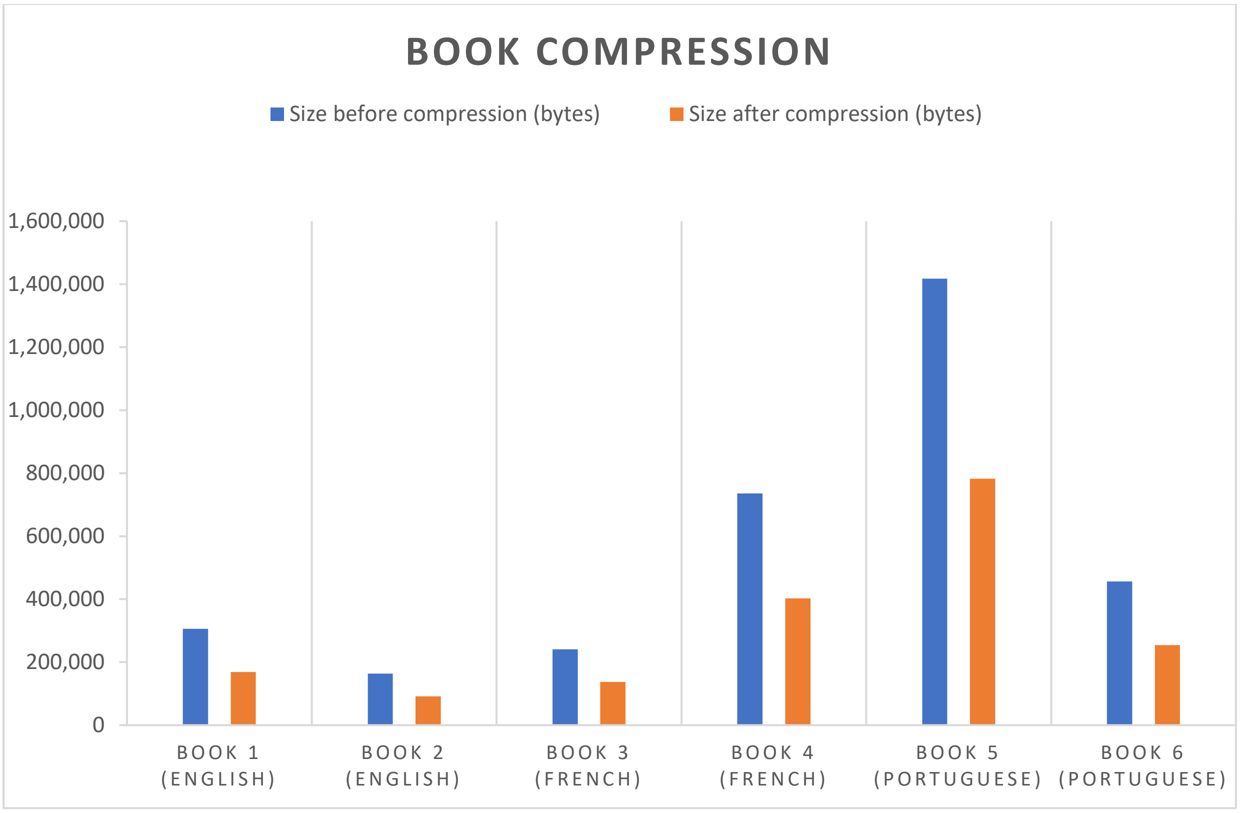
Book name	Language	Size before compression (bytes)	Size after compression (bytes)	Compression as a percentage of original size (%, 1dp)
The Great Gatsby by F. Scott Fitzgerald	English	306,258	168,929	55.2
The Strange Case of Dr. Jekyll and Mr. Hyde by Robert Louis Stevenson	English	163,465	91,679	56.1
Candide, ou, l'optimisme by Voltaire	French	240,614	137,170	57.0
Madame Bovary by Gustave Flaubert	French	736,140	402,336	54.7
Os Maias: episodios da vida romantica by Eça de Queirós	Portuguese	1,417,056	782,478	55.2
A Cidade e as Serras by Eça de Queirós	Portuguese	456,777	254,446	55.7

From the repetitive corpus dataset of repetitive texts,

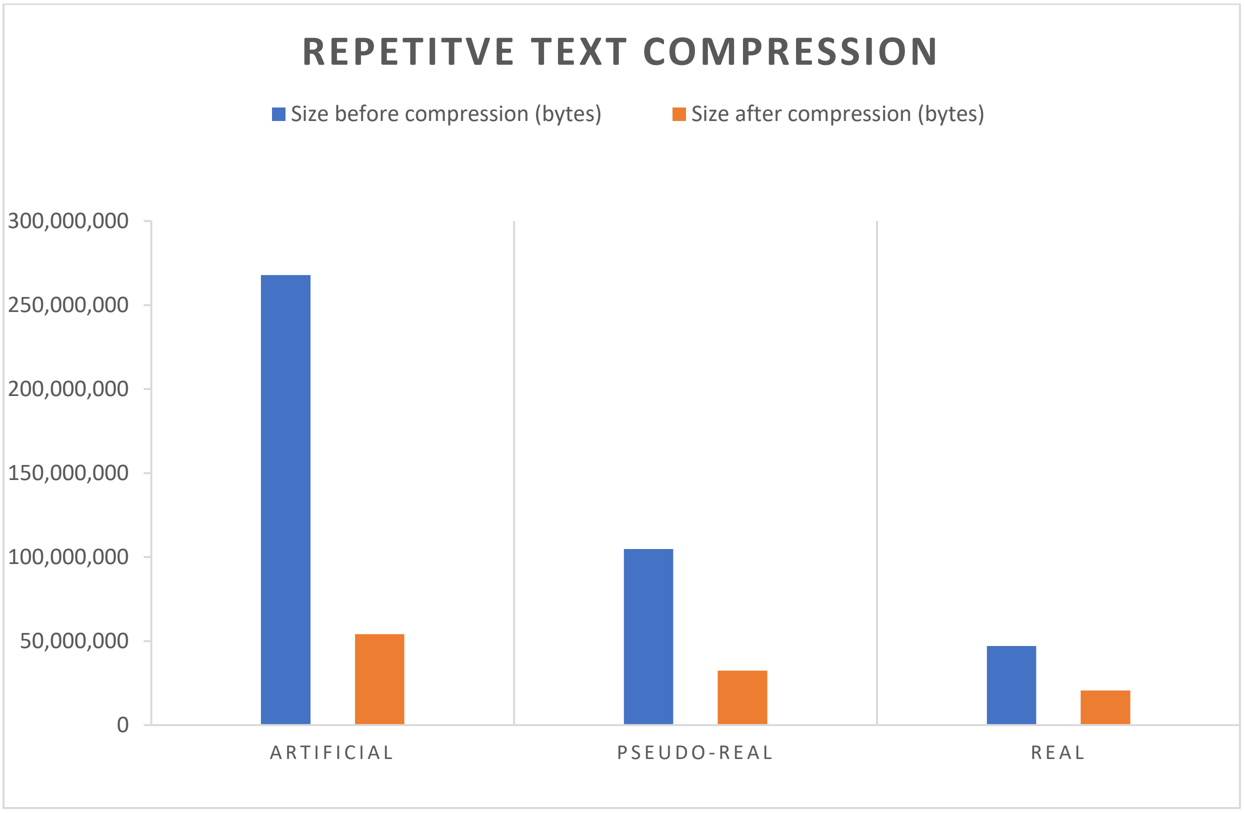
<u>Table for repetitive corpus:</u>

Type	Size before compression (bytes)	Size after compression (bytes)	Compression as a percentage of original size (%, 1dp)
Artificial	267,914,296	54,186,886	20.2
Pseudo-Real	104,857,600	32,342,953	30.8
Real	46,968,181	20,519,562	43.7

Graph for books:



Graph for repetitive corpus:



Git hub for the project at, https://github.com/Priptide/huffman-compression