Introduction

The Huffman encoding algorithm is used to compress data by reducing the number of bits it takes to represent the most frequently occurring character in the text. As all text is represented as binary and each character in the text is represented by a certain number of bits, it makes sense to reduce the number of bits it takes to represent the most frequent characters (as they make up most of the text) and assign larger bits to the ones that occur infrequently (to prevent collision).

Tools required

The Huffman encoding algorithm uses two classes to encode and decode files these classes are:

Node Class

The node class creates a node object that contains:

1. Character – each unique character in the text has a node object assigned to it and the character variable of the node class stores that character.
2. Value – this stores the number of times the character occurred in the text.
3. Left – the reference to the left node (which is needed to build the Huffman tree)
4. Right – the reference to the right node (which is needed to build the Huffman tree)

PriorityQueue Class:

This class is used to create a priority queue which comes in use when the creating the Huffman tree. This is explained later.

Algorithm

We first scan the file and create a frequency list of every character that exists in the file. This is then stored in a python dictionary as shown:

```
with io.open("Books/" + aBook, 'r', encoding='utf8') as file
    book = file.read()
dict1 = defaultdict(int)
for i in book:
    dict1[i] += 1
return dict1
```

Once we create the dictionary that contains the frequencies of every unique character in the text, we then create the Huffman tree, this is done with the help of a priority queue. We create a priority queue and we create a node each for the keys in the dictionary (the character is the key and its frequency the value).

We add each node to the priority queue which sorts the queue according to the minimum frequency in each node. Once the priority queue is completely created, we remove the top two nodes and add it to a new node (as the children of the two nodes).

The new node has the sum of the two frequencies of the children node and a special character called "NON_CHAR". This node is added back to the priority queue. This process is repeated till we are left with one node which is the root of the Huffman tree.

```
queue1 = pq()
for key, value in aDict.items():
    queue1.insert_node(node(key, value))
# print(queue1)
while (queue1.get_size() > 1):
    left = queue1.extract_min()
    right = queue1.extract_min()
    root = node("NON_CHAR", left.get_value() + right.get_value())
    root.set_left(left)
    root.set_right(right)
    queue1.insert_node(root)
huffman_root = queue1.extract_min()
return huffman_root
```

The next step is to create the Huffman code for the characters. This is done via a recursive function, we first send the method the Huffman tree, an empty string and an empty dictionary. The method recursively dives down the child nodes of the Huffman tree till it reaches the leaf nodes. While diving down the tree it adds a "0" to the empty string if it goes down the left child or a "1" to the empty string if it goes down the right child.

Once it reaches the leaf, it copies the character and assigns the string to the character as its Huffman code. The character and its code are saved to the empty dictionary.

```
if huffman_root.get_left() is None and huffman_root.get_right() is None \
        and huffman_root.get_character() is not "NON_CHAR":
    aDict[huffman_root.get_character()] = aString
    return aDict
create_huffman_code(huffman_root.get_left(), aString + "0", aDict)
create_huffman_code(huffman_root.get_right(), aString + "1", aDict)
```

Once the Huffman code is created, we then have to read the characters from the file, translate it to the Huffman code and pack the code in bytes. We do this by creating a large string of 0s and 1s (all the

characters including space between words are saved as one large binary string) and then dividing the binary number into chunks of bytes (8 bits or 8 digits).

The eight bit binary numbers are then converted to integers and stored into a bytearray. The last chunk has extra zeros for padding. The bytes in the bytearray written to a text file as an encoded file.

```
with io.open("Books/" + aBook, 'r', encoding='utf8') as file:
    book = file.read()
if (option == 1):
    file = "encoded_books_normal/" + str(aBook) + "_encoded.txt"
elif (option == 2):
    file = "encoded_books_fixed_freq/" + str(aBook) + "_encoded.txt"
binary = ""
for i in book:
    binary += aDict[i]
padded_binary = binary + "0" * (len(binary) % 8)  # padding
count = 0
byte_size = ""
bytes1 = bytearray()
for i in padded_binary:
    if (count % 8 == 0):
        if (count != 0):
            bytes1.append(int(byte_size, 2))
        byte_size = ""
    byte_size += i
    count += 1
with open(file, 'wb') as pen:
    pen.write(bytes(bytes1))
```

The process of decoding involves the reverse process, we take the dictionary from the encoding process and reverse the keys and values. We read from the encoded file, break the file into pieces of eight bits using the .rjust() method. Once we do this we convert the bytes to binary numbers and then run them through the reverse dictionary to print their equivalent characters to the decoded file.

```
dict2 = {}
for key, value in dict1.items():
    dict2[value] = key
binary_String = ""
for i in book:
    binary_num = bin(i)
    binary_num = binary_num[2:].rjust(8, '0')
    binary_String += binary_num
book_string = ""
num1 = ""
for i in binary_String:
    num1 += i
    if (dict2.get(num1)):
        book_string += dict2[num1]
        num1 = ""
file = ""
if (option == 1):
    file = "decoded_books_normal/" + str(aBook) + "_decoded.txt"
elif (option == 2):
    file = "decoded_books_fixed_freq/" + str(aBook) + "_decoded.txt"
with open(file, 'w', encoding='utf-8') as pen:
    pen.write(book_string)
```

Fixed Frequencies

The idea of generating frequencies by parsing the document can be avoided by taking fixed frequencies. The reason for this approach is that all English language documents are made of standard English words and if we can plot the frequency with which these characters appear in the most commonly used English words, we will have a frequency closest to the actual frequencies of the characters in the document.

Research done on frequencies of characters like the ones done by Mark Mayzner[1] in his book[2] and by another research done by University of Notre Dame[3] show us a list of English characters and their frequency. However, these researches do not provide us with a difference in frequencies between characters in uppercase vs characters in lower case. Additionally, it does not provide frequencies of characters like the space between two words, period and exclamation mark etc. Thus, I looked for research that would provide fixed frequencies for all possible characters as the text I wished to encode and decode had characters beyond the English alphabets.

---

[1] **English Letter Frequency counts** - http://norvig.com/mayzner.html

[2] **Mayzner, M. S., & Tresselt, M. E. (1965).** Tables of single-letter and digram frequency counts for various word-length and letter-position combinations. (https://books.google.com/books/about/Tables_of_Single_letter_and_Digram_Frequ.html?id=FI7BHgAACAAJ)

[3] **Univerity of Notre Dame** - https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html

Textware Solutions[4] has developed a product called Instant Text which helps their clients auto complete the text they write for a document. Their research consists of studying all their client's documents (a year's worth of documents) and obtaining frequencies[5] for all characters. This list has all the characters (uppercase, lowercase numbers and special characters) and their frequencies. Using this table, we can construct the occurrence of each character by multiplying its frequency with the length of the text.

$$Character\ Count\ =\ Fixed\ Frequency\ \times\ Length\ of\ Text$$

Once we have the number of times a character appears, we follow the same process as above to create the Huffman tree and encode and decode the file.

Compression Ratios and time taken

The documents I have taken are text files from Project Gutenberg[6], these are novels written by famous author and their file size range from 80kb to 800kb. Each of these files are more than a page long. They are compressed by the methodologies (normal Huffman and fixed frequency Huffman), written as encoded files. To check the compression ratio, we measure the file size of the uncompressed size to the ratio of the compressed size[7].

$$Compression\ Ratio = \frac{uncompressedSize}{compressedSize}$$ [8]

The compression ratios are checked by the method compare_compression():

```
s1 = os.path.getsize("encoded_books_normal/" + aBook + "_encoded.txt")
s2 = os.path.getsize("Books/" + aBook)
s3 = os.path.getsize("encoded_books_fixed_freq/" + aBook + "_encoded.txt")
compression_normal = (s2 / s1)
compression_fixed = (s2 / s3)
print("*" * 10, aBook, "*" * 10)
```

Once we check the compression ratio we then decode the encoded file using the same dictionary to write out the decoded file. Once this is done we use python set class to check if there is any difference in the texts. We use the set difference method and if there is no difference between the uncompressed and decoded text, it returns an empty set, we test this condition and print "No Difference" if the set is an empty set.

---

[4] **Textware Solutions** https://textware.com
[5] **Characters and Frequency** http://www.fitaly.com/board/domper3/posts/136.html
[6] **Project Gutenberg -** https://www.gutenberg.org/browse/scores/top
[7] **Data Compression ratio** - https://en.wikipedia.org/wiki/Data_compression_ratio
[8] **Compressed ratio definition** - https://en.wikipedia.org/wiki/Data_compression_ratio

```
199         difference_normal = set(book_original).difference(book_decoded_normal)
200         difference_fixed = set(book_original).difference(book_decoded_fixed)
201         diff_fixed = "No difference" if len(difference_fixed) == 0 else str(difference_fixed)
202         diff_normal = "No difference" if len(difference_fixed) == 0 else str(difference_normal)
```

Here are the average run times and the average compression ratios (over 10 runs) for the 50 encoded and decoded novels from project Gutenberg. The result of the individual runs are printed in a file called results.txt. The final average result is printed as a dictionary where the file name is the key and the value is a list that consists of compression ratios (for normal Huffman encoding and fixed frequency Huffman encoding) and their execution times (in the next page).

Name of the text	compression ratio (normal frequencies)	Time taken normal (seconds)	compression ratio (fixed frequencies)	Time taken fixed (seconds)
a conneticut yankee.txt	1.823	2.876	1.738	2.969
Alice in wonderland.txt	1.823	0.476	1.709	0.519
american journal.txt	1.781	0.371	1.713	0.372
anthem.txt	1.808	0.324	1.729	0.346
awakening.txt	1.829	1.211	1.734	1.284
beowulf.txt	1.802	0.857	1.688	0.970
brothers grimm.txt	1.858	1.971	1.749	2.158
candide.txt	1.774	0.661	1.703	0.657
civil disobedience.txt	1.844	2.559	1.756	2.728
desert healer.txt	1.855	1.690	1.754	1.864
dorian gray.txt	1.805	1.568	1.725	1.609
drJekyll.txt	1.815	0.430	1.727	0.455
dubliners.txt	1.830	1.309	1.728	1.400
four mystery plays.txt	1.840	2.681	1.733	2.901
FrankenStein.txt	1.838	1.478	1.744	1.576
hard times.txt	1.836	2.290	1.731	2.500
Hear of Darkness.txt	1.799	0.641	1.727	0.681
huckelberry fin.txt	1.807	2.404	1.701	2.511
kama.txt	1.799	1.089	1.723	1.160
kings mountain.txt	1.746	0.348	1.683	0.344
meditations by empreror.txt	1.824	1.333	1.747	1.421
metamorphosis.txt	1.829	0.411	1.749	0.435
occurrence at Owl Creek.txt	1.762	0.134	1.700	0.136
on liberty.txt	1.841	0.953	1.755	0.992
peter pan.txt	1.827	0.790	1.719	0.843
robinson crusoe.txt	1.850	2.488	1.748	2.840
scarlet letter.txt	1.833	1.847	1.738	1.925
sense and sensibility.txt	1.814	2.790	1.733	2.954
sherlock.txt	1.804	2.298	1.717	2.536
short stories africa.txt	1.825	0.904	1.732	0.947
short stories france.txt	1.829	0.764	1.740	0.772
siddhartha.txt	1.820	0.667	1.734	0.730
Souls of Black Folk.txt	1.818	1.460	1.745	1.535
study in scarlet.txt	1.826	0.794	1.731	0.858
than frome.txt	1.830	0.610	1.734	0.623
the african.txt	1.824	1.686	1.746	1.713
the devils dictionary.txt	1.774	1.265	1.700	1.355
the hound of baskervilles.txt	1.842	1.177	1.741	1.184
the legend of sleepy hallow.txt	1.800	0.289	1.728	0.281
the prophet.txt	1.796	0.299	1.684	0.286
the secret garden.txt	1.868	1.621	1.739	1.674
the sign of four.txt	1.802	0.745	1.724	0.756
the turn of the screw.txt	1.838	0.703	1.729	0.804
time machine.txt	1.821	0.555	1.736	0.587
Tom Sawyer.txt	1.800	1.467	1.705	1.633
top of the world.txt	1.802	0.489	1.712	0.536
tragical hisroty.txt	1.711	0.480	1.624	0.459
war of worlds.txt	1.828	1.209	1.740	1.273
wizard of oz.txt	1.808	0.666	1.716	0.670
Yellow Wallpaper.txt	1.741	0.180	1.660	0.187

References

English Letter Frequency counts - http://norvig.com/mayzner.html

[2] **Mayzner, M. S., & Tresselt, M. E. (1965).**
Tables of single-letter and digram frequency counts for various word-length and letter-position combinations.
(https://books.google.com/books/about/Tables_of_Single_letter_and_Digram_Frequ.html?id=FI7BHgAACAAJ)

[3] **Univerity of Notre Dame** -
https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html

[4] **Textware Solutions**
https://textware.com

[5] **Characters and Frequency**
http://www.fitaly.com/board/domper3/posts/136.html

[6] **Project Gutenberg**
https://www.gutenberg.org/browse/scores/top

[7] **Data Compression ratio**
https://en.wikipedia.org/wiki/Data_compression_ratio

[8] **Compressed ratio definition**
https://en.wikipedia.org/wiki/Data_compression_ratio