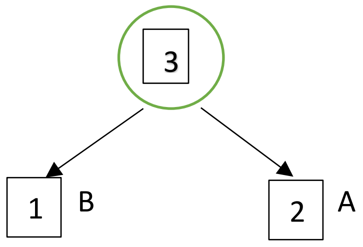
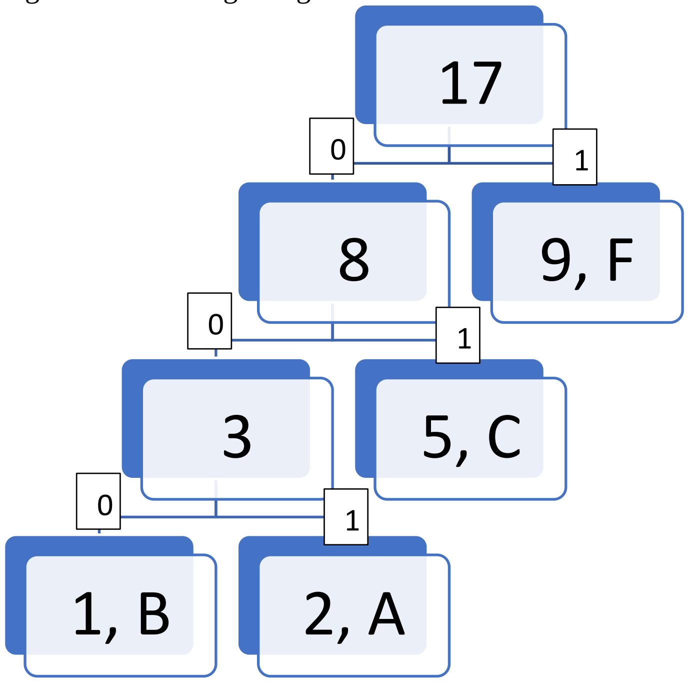
Specification

Literature Review

- **Huffman encoding.** This is a lossless data compression algorithm. Huffman compression algorithm provides efficient code, which analyses the frequencies of characters in the text, creates a tree and assigns code for each character. It can be done by following these steps:
  1. First of all, Huffman coding analyses the text and calculates the frequencies of each character in the text. For example, "B" = 1, "C" = 5, "D" = 2, "F" = 9.
  2. Characters should be sorted according to their frequencies in the ascending order: "B" = 1, "D" = 2, "C" = 5, "F" = 9.
  3. Combine the two characters, which have the least frequencies (first 2 elements of the sorted list) and assign them into an empty node, which will have the value of their combined frequencies. The minimum frequency will be the left node and the second minimum frequency will be the right node.



  4. Remove these characters from sorted list and add the sum of their frequencies to the list.
  5. Repeat step 3 to 4 until no characters left.
  6. Assign 0 to the left edge and 1 to the right edge.



  7. Encoded: F = 1, C = 01, A = 001, B = 000. Original size of those string was 17*8= 136 bits. But after Huffman coding, the size will be (4*8) + 17 + (3+2*3+5*2+9) = 77 bits.

  Decompression. For decompression, we can take the code of the character and traverse through tree. For example, to decode 000, we need to go left, left, left and reach character "B".
- **LZ77** lossless data compression algorithm. For instance, zip and gzip based on this algorithm. This data compression iterates through the input string/text and searches for the matches, if there is a match, it stores it in the search buffer. Also it uses triples to represent (o, l, c), where

o – offset, the number of positions that we need to move backwards in order to find the first matching character, l – length, which is the length of the match, c – character, which is found after the match. For example:
The text is: s, d, s, d, a, d, s, d, s

1. First of all, we will find "s" and triple will be (0,0,s), because this is a first match and we do not need to move backwards (o = 0), search buffer is currently empty (l = 0), and the character is "s" (c = s). We will move l+1 and we will find d and it will have triple (0,0,d).
2. When we move l+1, we will reach "s", we know that we have "s" and "sd" in the search buffer (but not "sda"), which means we need to move backwards 2 times (o = 2) and read to characters (l = 2), the next new character that we found is "a", so the triple (2,2,a).
3. We will move to l+1 character afterwards and reach "s", in the search buffer we have "d", "ds", "dsd, but not "dsds", so we will need to move backwords 4 times to reach first "d" (o = 4) and read next 3 characters (l = 3), the character we find is "s" (c = s), so the triple is (4, 3, s).
4. Finally, the encoded file will be (0,0,s), (0,0,d), (2,2,a), (4, 3, s).

   Decompression. When we need to decompress the encoded file, we always need to start with the first triple. (0,0,s), (0,0,d) codes will just give as (s, d). (2,2,a) means move 2 positions to the left and read 2 characters ("sd") and write "c", which will give us the text: s, d, s, d, a. (4, 3, s) means move 4 position to the left and red 3 characters ("dsd") and writhe "d", which will give us: s, d, s, d, a, d, s, d.

- **Run-Length Encoding** is a lossless data compression algorithm, which converts the stream of data into a sequence of counts of consecutive data values in a row. For example, if we have data stream: "AAASSSRRRFF", the encoded data will look like: "3A3S5D4R2F". As you can see it counts for the number of the consecutive characters and encodes it in the form (counter, character).
  Encoding the data using RLE is simple, the code will need to iterate through each character and count the frequencies of consecutive characters. When the next character will be different than the previous, the code will append the counter and the character to the encoding.
  Decompression. When we need to decompress the data, the code needs to iterate through encoded data, if it catches numeric value and then non-numeric character, non-numeric character will be added the <u>numeric value</u> number of times.
  It will save a lot of space if there is a sequence of 4 or more repetitive character in the text. This is effective compression algorithm, if the file has a lot of repetitive data, for example spaces for indentation in the txt file. However, if the number of repetitive characters is low, then it will result in greater size file. For example, suppose "ASDF" takes 4 bytes of memory, after RLE compression the coded data will be "1A1S1D1F", which results in greater file with size 8 bytes.

Data Structures and Algorithms

1. ***Data Structures***

- Dictionary: I used dictionary for storing the characters and their number of occurrences in the text, where the key is the character and the value is its frequency.
- Lists: I used lists to store the characters and their frequencies.
- Tuple: I used tuples for storing the first two elements of the list, which have the lowest frequencies, which I used to create a tree.
- Tree: I used tree for storing the characters, so that most frequently characters had shorter encoding and ensuring that every character will have unique encoding.

2. **Algorithms**
    - Quicksort: I used this sorting algorithm to sort the list by ascending order of their frequencies.
    - Recursive Algorithms: I used recursive algorithms to create character tree, without their frequencies and assign 0 to the left edges and 1 to the right edges.

Weekly log

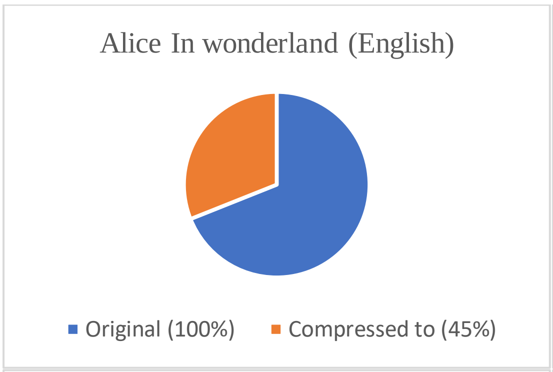
1. On the 1st week, I learnt about module itself, so I was watching 1st week introduction lectures, in order to understand the concept of the module.
2. On the 2nd week, I started learning about algorithms and tried to create Fast Exponential algorithm.
3. On the 3rd week, I read through the coursework specification and watched Coursework Explanation video, in order to have an idea what I should build and made a plan.
4. On the 4th week, I was reading/understanding the Huffman Compression algorithm and watching youtube videos to consolidate my knowledge.
5. On the 5th week, I started creating my algorithm. By the end of the week, I wrote a function for iterating through input and creating dictionary of characters and their frequencies.
6. On the 6th and 7th weeks, I was reading/understanding about the trees in python. Finally, I created a sorted list, which included characters and their frequencies and build a tree inside the list.
7. On the 7th and 8th weeks, I was learning about writing to the binary files (.bin) and converting strings into bytes, in order to write decoded code into the .bin file. By the end of the week, I wrote two functions for compressing (which included saving decoded text and tree itself) and decompressing (which included using the saved tree of the original file) text files
8. On the 9th week, I was testing my program on different language books and datasets, fixed the code (instead of function which converts bytes to string and visa versa, I used pythons bitstring package), writing the Specification and finishing documentation of the code.

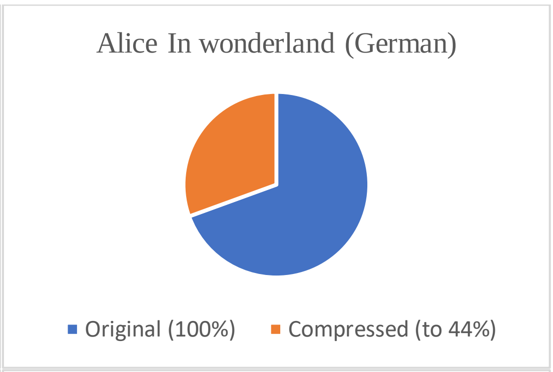
Performance analysis

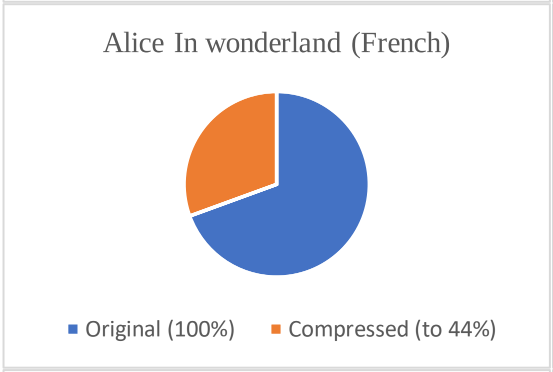
In order to analyse the performance of my program, I run it through 2 books and each of the book in 3 different languages. Below table and pie charts show the performance of my program.

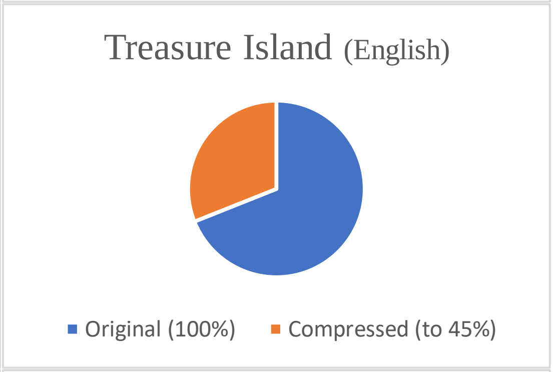
Table of Perfomance

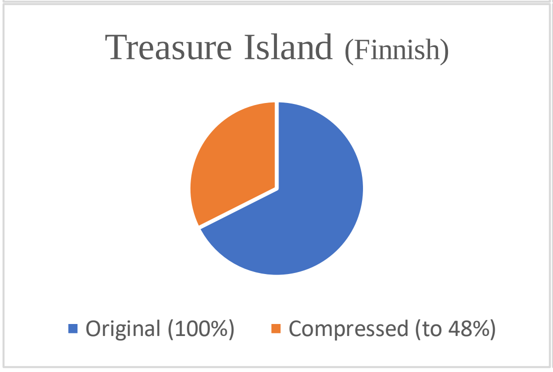
Book name	**Language**	***<u>Original size (bytes)</u>***	***<u>Compressed size (bytes)</u>***	***<u>% of the original size</u>***
Treasure Island	English	400129	218718	45.0
Treasure Island	German	447098	246969	45.0
Treasure Island	Finnish	427254	222888	48.0
Alice in wonderland	English	174693	95538	45.0
Alice in wonderland	German	183772	104813	44.0
Alice in wonderland	French	184669	103484	44.0

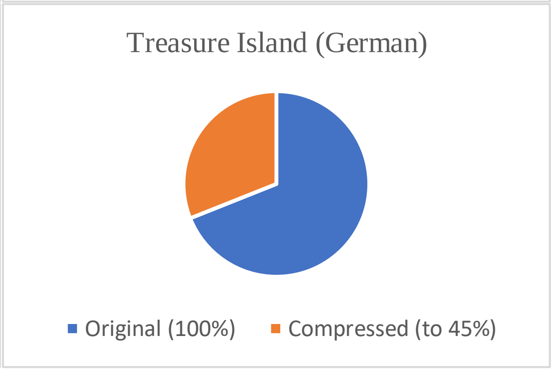












My algorithm builds separate tree for each language and book, because each language has own unique characters and it would take a lot of memory if I stored each character in the same tree. As you can see from the table, each book has almost the same percentage of the compression (around 45%). Furthermore, I used my algorithm to decompress datasets, which had the same performance, however datasets with size 200MB took up to 5 minutes to decompress the file.
This is how my algorithms works:

1. Function iterates through the whole text and builds the frequency dictionary, where the key is a character and the value its frequency.
2. Function inserts each character and its frequency into the list and sorts (quicksort) them, in order to get the least frequencies at the start.
3. Function creates a tree by selecting first two elements in the list, assigning them to the node, which has value of their frequencies. It will loop until, there will be no character left.
4. Recursive function which deletes all frequencies in the list, leaving just tree with the characters.
5. Recursive function which assigns the left edge "0" and the right edge "1". It saves codes of each character in the dictionary.
6. Function, which combines all functions, by iterating through each character in the text, it assigns each character a code using encoding dictionary and encodes the whole text. Encoded text is converted to bytes and saved in the .bin file, while tree, which will be used to decompress the file, is saved in the .pickle file
7. Function, which reads the .bin (encoded text) and .pickle (tree) files and decompresses the text by iterating through encoded text and using a tree decodes the text ("0" left edge, "1" right edge).

List of References

- Tom Scott, "How Computers Compress Text: Huffman Coding and Huffman Trees", 2019, [Online], https://www.youtube.com/watch?v=JsTptu56GM8
- Karleigh Moore and Jimin Khim, "Huffman Code", 2021, [Online], https://brilliant.org/wiki/huffman-encoding/
- "How can I implement a tree in Python?", [Online], https://stackoverflow.com/questions/2358045/how-can-i-implement-a-tree-in-python
- Keno Leon, "Making Data Trees in Python", 2020, [Online], https://medium.com/swlh/making-data-trees-in-python-3a3ceb050cfd
- Michael Sambol, "Quick sort in 4 minutes", 2016, [Online], https://youtube.com/watch?v=Hoixgm4-P4M
- "Convert binary string to bytearray in Python 3", [Online], https://stackoverflow.com/questions/32675679/convert-binary-string-to-bytearray-in-python-3
- "Python File Write", [Online], https://www.w3schools.com/python/python_file_write.asp
- "Writing bits to a binary file", [Online], https://stackoverflow.com/questions/21220916/writing-bits-to-a-binary-file
- "what is the best way to save tuples in python", [Online], https://stackoverflow.com/questions/35090264/what-is-the-best-way-to-save-tuples-in-python
- "Getting file size in Python?", [Online], https://stackoverflow.com/questions/6591931/getting-file-size-in-python
- "Lossless compression", [Online], https://en.wikipedia.org/wiki/Lossless_compression#General_purpose
- Dhanesh Budhrani, "How data compression works: exploring LZ77", 2019, [Online], https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097
- Scott Robinson, "Run-Length Encoding", [Online], https://stackabuse.com/run-length-encoding/