Text file compression and decompression using Huffman Encoding.

Introduction

In simple terms, each symbol in a text file uses at least 1 byte of memory. Using Huffman coding, we can analyse a text file to determine which symbols are more common and which are less common. We can then fit many of these smaller codes into a single byte.

Data Structures

Binary Tree

My implementation makes use of a binary tree style structure to create, store, encode and decode symbols to and from their Huffman code representation. My implementation is simple and based on Node objects.

Heap

The algorithm uses a heap when constructing tree, to sort and hold Nodes before processing. It is a minimum heap. Symbols with a lower frequency are processed first.

HashMap

A Hash Map style structure Is used to store symbols and their created Huffman codes e.g. ‘a’=>“011”.

Bit Shifting

The program can encode binary representations of multiple symbols in a single byte using bit shifting. This is a bit level manipulation of the value of a byte – we shift the value of the byte to match our Huffman code and then write it to a file.

Execution Sequence

Compression (summary)

1. Count all the symbols in our file, recording their frequency.
2. Create an object representing each symbol and its frequency, place into minHeap.
3. While there are two symbols left:
    a. Poll two symbols from the queue.
    b. Create a new node, this node being the parent of the two nodes. Assign it’s frequency to be the sum of its children.
    c. Place this node back in the queue.
4. The final node in the queue is the root node of our tree.
5. ‘Walk’ along the tree, recording each step. A ‘left’ step is 0, a ‘right’ step 1. When we reach a leaf, the concatenation of all steps is its Huffman code. So if reach leaf node ‘a’ with LLL, its Huffman code is ‘000’.
6. Do this for all Nodes, place all symbols and codes in HashMap.

7. Calculate a string representation of the tree. Encode the string representation using bit shifting. Add to metadata buffer.
8. Capture all symbols of the tree in an arbitrary order. Encode them in bytes. Add to metadata buffer.
9. Write metadata header to file.
10. Read the input file character by character.
    a. For each character, get its Huffman code.
    b. Use bit shifting to write the code to a byte.
    c. When we have shifting a multiple of 8 bits, write this to the ByteBuffer.
    d. Do this for every symbol.
11. When we reach 4096 bytes, or EOF, write the buffer to file.
12. Repeat until the whole file is written.
13. Close the file.

Decompression (summary)

1. Open the file.
2. Fetch the metadata from the file. The size of the metadata in bytes is held as an integer in the first 4 bytes.
3. From the metadata, rebuild the Huffman tree.
4. Read bytes into a buffer.
5. For each byte:
    a. Read a single bit.
    b. Travel down the Huffman tree based on the bit (0 or 1).
    c. If the current node is a leaf, get that node’s symbol. Else go to 1.
    d. Add that symbol to the ByteBuffer for writing. Write the buffer is necessary.
6. Repeat until the whole file is written.
7. Close the file.

Complexity and Test Performance

Theoretical complexity

Compress	
read each symbol into a hashMap	O(nk) or O(n)
build a heap for k nodes	O(k log k)
build a Huffman tree for k nodes	O(k)
read a byte, encode it, n times	O(n log k)
Total	Theory: O(nk) | Practice: O(n log k)
Decompress	
build a heap for k nodes	O(k log k)
read a byte, decode it, n times	O(n log k)
Total	O(n log k)

An upper bound for k

It is always true that k <= n for all cases.

Proof: Suppose that k > n. That means that there are k+1 symbols contained in n bytes. Each symbol requires at least 1 byte. So we need n+1 bytes. But n+1 > n. So we have reached a contradiction.

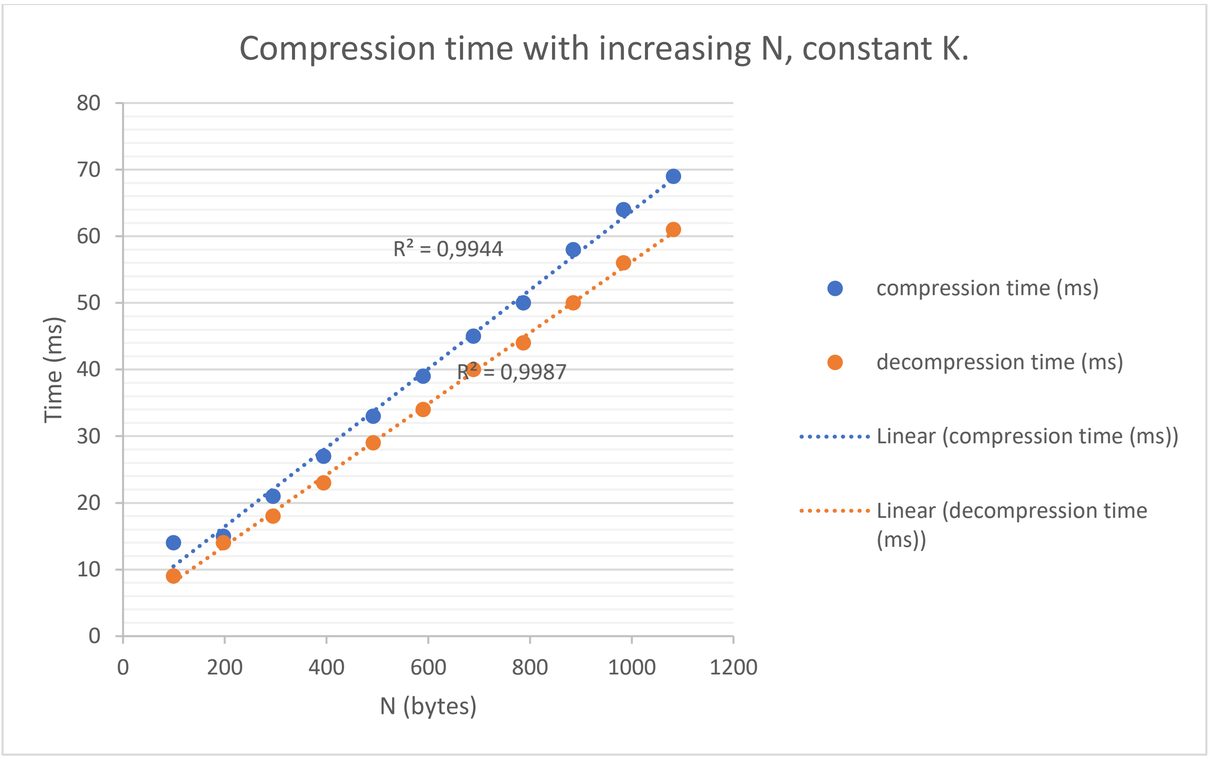
So the worst case complexity of any combination of n and k is for both compression and decompression is:

O(n log n)

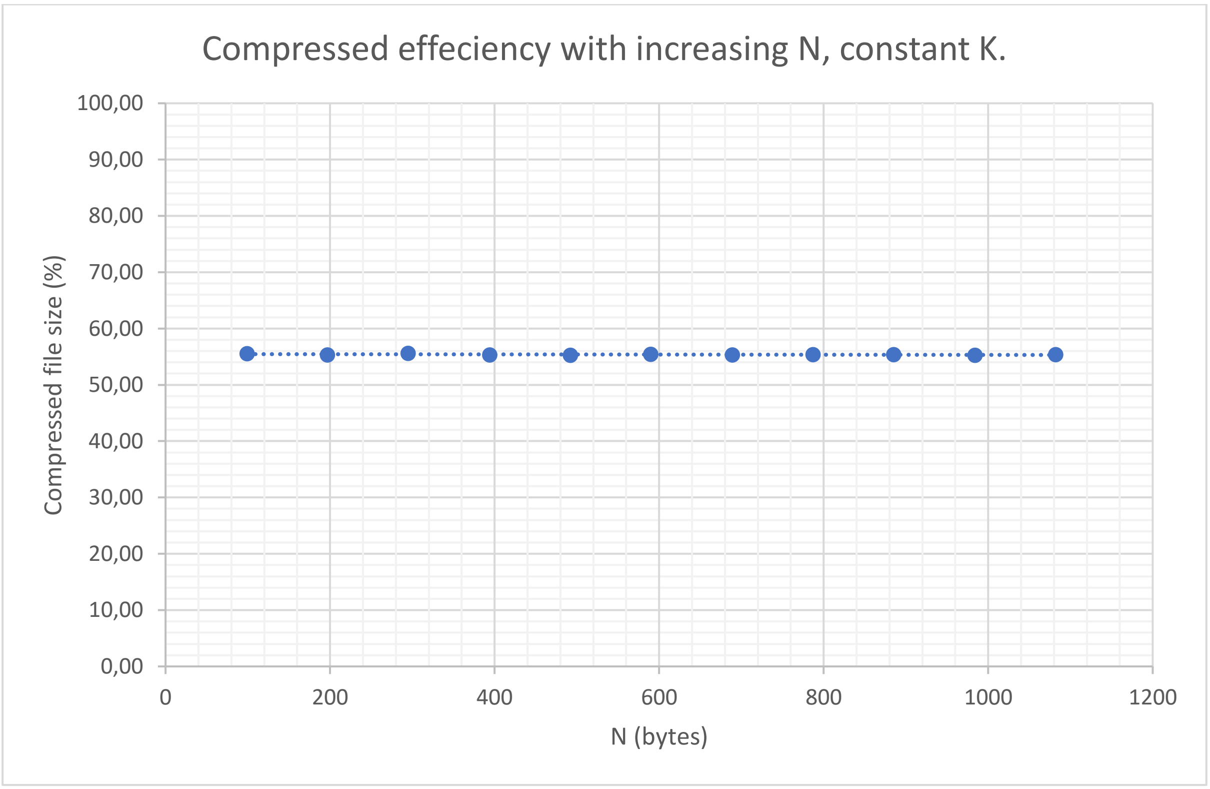
Observed Performance

Constant K, Linear N						
File	K	N (kb)	compression time (ms)	compressed size (kb)	Compressed file size (%)	decompression time (ms)
lorem1	43	99	14	55	55,56	9
lorem2	43	197	15	109	55,33	14
lorem3	43	295	21	164	55,59	18
lorem4	43	394	27	218	55,33	23
lorem5	43	492	33	272	55,28	29
lorem6	43	590	39	327	55,42	34
lorem7	43	689	45	381	55,30	40
lorem8	43	787	50	436	55,40	44
lorem9	43	885	58	490	55,37	50
lorem10	43	984	64	544	55,28	56
lorem11	43	1082	69	599	55,36	61

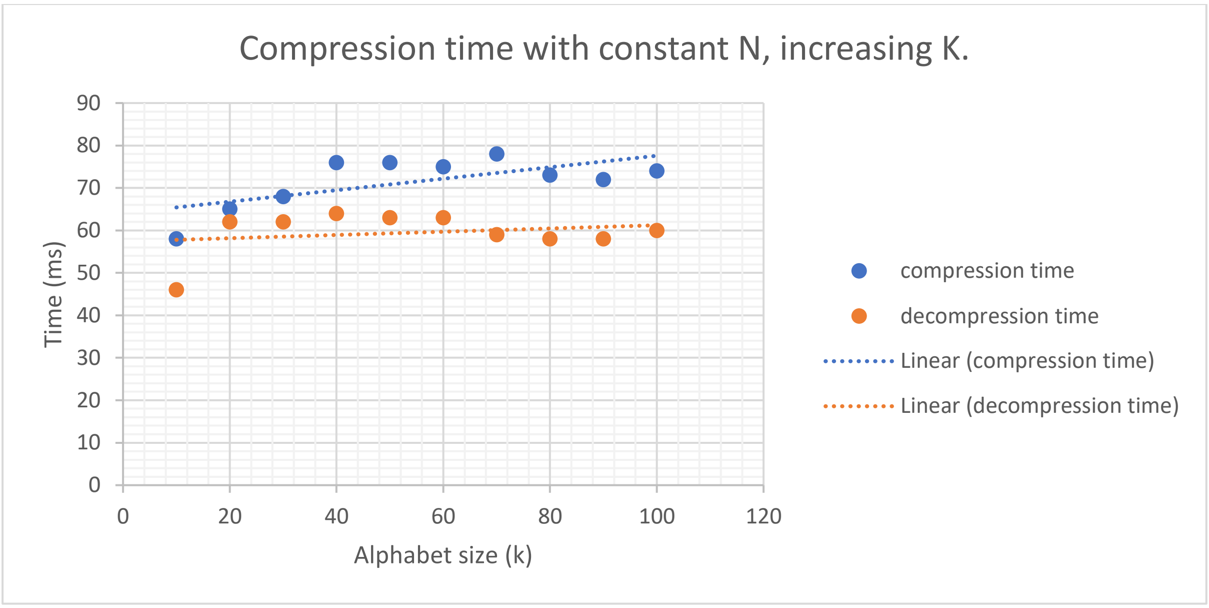
Linear K, Constant N						
File	K	N (kb)	compression time	compressed size (kb)	Compressed file size	decompression time
symbols10	10	976	58	440	45,08 %	46
symbols20	20	976	65	561	57,48 %	62
symbols30	30	976	68	627	64,24 %	62
symbols40	40	976	76	625	64,04 %	64
symbols50	50	976	76	615	63,01 %	63
symbols60	60	976	75	601	61,58 %	63
symbols70	70	976	78	601	61,58 %	59
symbols80	80	976	73	611	62,60 %	58
symbols90	90	976	72	619	63,42 %	58
symbols100	100	976	74	627	64,24 %	60

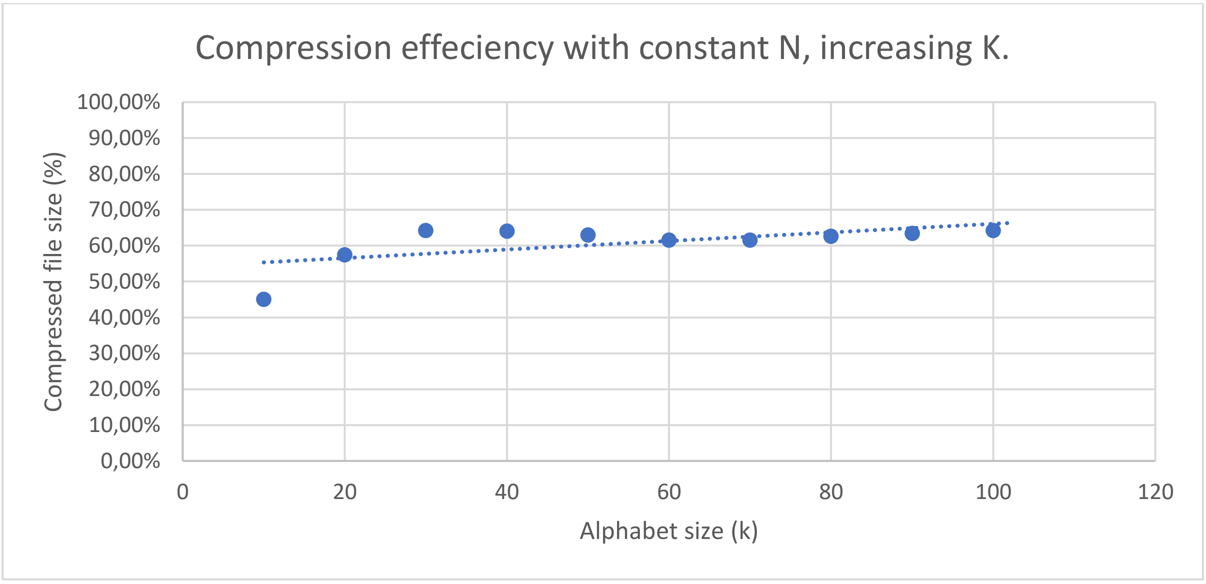


Compression time with increasing N, constant K.
Time (ms)
N (bytes)
R² = 0,9944
R² = 0,9987
compression time (ms)
decompression time (ms)
Linear (compression time (ms))
Linear (decompression time (ms))



Compressed effeciency with increasing N, constant K.
Compressed file size (%)
N (bytes)





Observations

- Compression/decompression observed efficiency is logarithmic w.r.t N.
- Compression/decompression observed speed is linear w.r.t N.
- Compression/decompression observed efficiency is logarithmic w.r.t K.
- Compression/decompression observed speed is logarithmic w.r.t K.

Unit testing

Unit test coverage has been tested using PIT. The latest PIT report can be found here:

PIT

When implementing data structures, a test-driven development approach was followed. This is reflected in the high coverage in these classes.

In addition to unit tests, there is a basic UI test which verifies expected functionality.