Lossless Data Compression Algorithms

Bîrlădeanu Raluca-Monica[1] and Stan Mihai-Cătălin[1]

[1] University "Politehnică" of Bucharest, Bucharest, Romania

Abstract. This paper investigates lossless data compression through the analysis of five key algorithms: Huffman coding, Arithmetic coding, Prediction by Partial Matching (PPM), Run-Length Encoding (RLE) and Lempel-Ziv-Welch (LZW). The study assesses their compression efficiency – through the lenses of both compression and decompression times, and compression ratio –, as well as their complexity and performance across multiple tests. To establish a foundation for understanding the algorithms, the paper provides a brief overview on entropy in the context of Information Theory, as well as descriptions about the algorithms. Ultimately, this paper tries to emphasis the advantages and disadvantages on each chosen algorithm, since the compression methods are crucial in this day of ever-expanding information, in order to be able to more effectively store and transmit data.

Keywords: Data Compression, Probability Coding, Partial Matching

1 Introduction

1.1 Problem solved

In this ever-growing expanse of digital information, the amount of new content that constantly arises at an almost overwhelming rate calls for the optimization of information storage and transmission. Although our computers evolved a lot through time, the sheer amount of content seems to outgrow even the best attempts at perfecting computers or phones. Within this content, compression algorithms emerge as the unsung heroes of our digital landscape, in hopes of solving what is a critical issue: allowing us to transfer and to keep data in a much more optimized manner, hence giving us the opportunity to save a lot of space if done right.

This paper wants to analyze different compression algorithms and study not only the efficiency with which they achieve their purpose, but also the complexity, as each algorithm is carefully crafted as a solution to a certain, more specific problem. Compression algorithms meticulously distill the vast information into compact, but faithful representations. As we delve deeper into a few of the most popular algorithms, this paper aims to explain the underlying principles of each algorithm, as well as compare how they perform in different scenarios and on a multitude of data sets, picked in order to ensure that the testing is done on a varied enough selection so that the results can be conclusive.

1.2 Practical applications

Data compression is crucial nowadays, and it has many practical applications such as:

- File compression – such as ZIP, GZIP, RAR compressed files and folders reduces their size, making them easier to store, transfer and share
- Multimedia compression – used to compress image, audio and video files, reducing their size while still preserving their quality
- Communication Networks – since most social networks have a certain file size limit, compression helps us stay within that limit, but it also allows for faster transmission, reduced network congestion and lower costs
- Database compression – since databases contain a large amount of information, compression can reduce storage requirements and improve query performance
- Cloud storage / Data archiving – compressing files before uploading them to the cloud / archiving them will save storage space and decrease the time required for data transfer

1.3 Chosen solutions

In this paper we are going to analyze: Huffman Coding, Arithmetic Coding, Prediction by Partial Matching (PPM), Run-Length Encoding (RLE) and Lempel-Ziv-Welch (LZW). Since Huffman and Arithmetic Coding are both Probability/Entropy Coding algorithms, we are first going to introduce the notion of Probability Coding, as well as the entropy term.

1.4 Probability Coding and Short Entropy Introduction

Shannon borrowed the definition of entropy from statistical physics, when entropy represents the chaos or the randomness of a system. The higher the entropy, the greater the number of possibilities in which that system can be rearranged. In particular, a system is assumed to have a set of possible states it can be in, and at a given time there is a probability distribution over those states. In the context of Information Theory, Shannon replaces the word “states” with “messages”, while also defining the notion of self-information of the message – the number of bits of information contained in it and, roughly speaking, the number of bits needed to encode the message.

Coding is the job of taking probabilities for messages and generating bit strings based on these probabilities. Based on this definition, we can distinguish between algorithms that assign a unique code (bit-string) for each message (a type of prefix code), and ones that blend the codes together from more than one message in a row. In the first category we have Huffman, which can find the optimal prefix, and in the second category we consider Arithmetic coding. The arithmetic codes can achieve better compression, but can require the encoder to delay sending messages since they need to be combined before they can be sent.

1.5 Prefix Codes

A prefix code is a uniquely decodable code, meaning that no bit-string representing one message is a prefix of another, which prevents ambiguity when decoding. Therefore, this means that given a complete sentence, the receiver can identify each word without requiring a special marker between words [1]. Prefix codes can be represented as binary trees, where each leaf corresponds to a message, and the code is determined by the path from the root to the leaf.

The relationship between prefix codes and entropy is established through the Kraft-McMillan Inequality. This inequality connects the lengths of codewords in a uniquely decodable code to their probabilities. Specifically, it imposes a budgetary constraint on the lengths allocation, with shorter codewords incurring a higher cost, akin to the concept introduced in the context of entropy. In the context of variable-length codes, shorter codewords are assigned to more probable symbols, while longer codewords are assigned to less probably symbols. This is a form of entropy encoding, where the goal is to represent more frequent symbols with shorter codes, thereby reducing the average length of the encoded message. Exponentiating the lengths of valid codewords results in a set of values resembling a probability mass function, where the sum of these exponential values must be less than or equal to 1 for a valid prefix code. This inequality comes with varied properties, where strict inequality implies redundancy in the code, equality signifies a complete code where the entire budget is utilized, and a failure of the inequality indicates non-uniqueness [2].

1.6 Chosen Comparison Criteria

In order to compare the chosen algorithms, we decided to compare the compression and decompression time, as well as the compression ratios obtained. Since we have a lot of different formats, we will be calculating the average data for all of the test files, as well as group the similar types of files in order to get a better idea of what types are compressed in a more efficient and effective way.

Another additional criteria that we will be measuring is entropy (on text files in different languages with roughly the same amount of characters), but only for Entropy Coding algorithms, such as Huffman Coding and Arithmetic Coding.

2 Solutions Analysis

2.1 Huffman Coding

Huffman Codes (Code description)
Huffman codes are optimal prefix codes generated from a set of probabilities by the Huffman Coding Algorithm. David Huffman developed this algorithm in 1951, while he was a student at MIT. While take the Information Theory class, his professor Robert Fano assigned a paper on finding the most efficient binary code. Huffman, unable to prove any of the existing codes were the most efficient, was about to give up when he discovered the idea of using a frequency-sorted binary tree, which quickly proved

to be the most optimal method. Building the tree from the bottom-up guaranteed optimality, unlike the top-down approach of Shannon-Fano coding [3]. The Huffman Tree construction will be detailed in the next section.

Huffman Coding

Huffman coding is a widely used algorithm for lossless data compression, particularly in applications where efficient storage and data transmission are crucial. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters [4].

Huffman Tree Construction

The algorithm begins by creating a leaf node for each unique symbol in the input data, accompanied by its frequency of occurrence. These leaf nodes are organized into a min-heap, a priority queue where the node with the minimum frequency is the root. Through successive extraction of two nodes with the minimum frequencies, a new internal node is created with a frequency equal to the sum of the extracted nodes' frequencies. This internal node is then inserted back into the min-heap. This process is repeated until only one node remains in the heap, which becomes the root of the Huffman tree.

Decompression

The decompression process involves translating a stream of prefix codes to individual byte values, usually by traversing the tree. The reconstruction of the Huffman tree can be achieved through either a preconstructed approach – at the expense of at least some measure of compression efficiency – or a dynamic method. If the data is compressed using canonical encoding – a particular type of Huffman code with unique properties which allow it to be described in a very compact method: rather than storing the tree specifically, canonical codes are ordered in such a way that it suffices to only story the lengths of the codewords [5] – the compression model can be precisely reconstructed. Once we get the tree, starting from the root, the algorithm follows these steps until a leaf is reached: if the current bit is 0, move to the left node; if the bit is 1, move to the right node. When a leaf node is encountered, the corresponding character is printed, and the iteration continues from step 1. The provided code takes a string input, encodes it, saves it as an encoded string, and then successfully decodes it, printing the original string [6].

Complexity

During compression, the algorithm goes through all the characters from the input file in order to get the frequencies of each character, which is done in O(n) time. In order to build the Huffman tree, the algorithm goes through the characters, adding each node into a priority queue sorted after the frequency in O(log n) time. Therefore, the time complexity of the Huffman coding algorithm is O(n log n), where n is the number of characters in the input string, and the auxiliary space complexity is also O(n).

During decompression, the algorithm moves to the right if the current bit is 1, or to the left is the current bit is 0. If it gets to a leaf, it means it decoded a symbol and it can restart the algorithm starting from the beginning of the tree. Therefore, the decompression complexity is O(n), where n is the number of bits from the encoding.

Advantages and Disadvantages

Advantages:

- Efficient Compression: Huffman coding is efficient in terms of compression. It assigns shorter codes to more frequent symbols, leading to better compression ratios.
- Variable-Length Codes: Huffman coding generates variable-length codes, allowing for more compact representation of frequently occurring symbols and better space utilization.
- No Ambiguity: The generated Huffman codes are uniquely decodable, ensuring that there is no ambiguity in decoding the compressed data.
- Simple Implementation: The basic concept of Huffman coding is relatively simple, making it easy to implement and understand.
- Widely Used: Huffman coding is widely used in various applications, including file compression algorithms like ZIP and JPEG.

Disadvantages

- Prefix Property: The variable-length codes must satisfy the prefix property, meaning no code should be a prefix of another. Ensuring this property adds complexity to the encoding and decoding processes.
- Lossless Compression Only: Huffman coding is suitable for lossless compression, but it may not be as effective for lossy compression as other methods like JPEG for images.
- Requires Prior Knowledge: For decompression, the recipient needs to have prior knowledge of the Huffman tree used for compression. Transmitting this information efficiently can add overhead.
- Not Always Optimal: While Huffman coding is generally efficient, it may not always produce the optimal compression for every type of data. Some other algorithms might achieve better results in specific cases.
- Variable-Length Codes Overhead: The use of variable-length codes introduces some overhead in terms of encoding and decoding complexity compared to fixed-length codes.

2.2 Arithmetic Coding

Description

Arithmetic coding is a form of entropy encoding used in lossless data compression. Unlike fixed-length coding methods, such as ASCII, arithmetic coding assigns variable-length representations to characters based on their probabilities in the message

sequence. The key concept involves representing the entire message as a single fractional number between 0.0 and 1.0. This number corresponds to a specific range on the number line, with frequently occurring characters occupying smaller ranges and infrequent characters occupying larger ones. This specific technique allows the information from the messages in a message sequence to be combined to share the same bits, which allows the total number of bits sent to asymptotically approach the sum of the self-information of the individual messages. As a result, it achieves more efficient compression, especially when dealing with messages where certain characters have significantly higher probabilities [7].

In the encoding process, the encoder considers the next symbol to be encoded, the current interval (which dynamically changes), and the probabilities assigned to each possible symbol in the given context. The current interval is subdivided into subintervals proportional to the symbol probabilities, and the one corresponding to the next symbol becomes the new interval for the subsequent step. This process continues until all symbols are encoded, resulting in a final interval that uniquely identifies the original symbol sequence. During decompression, it is not necessary to transmit the entire final interval; rather, transmitting a fraction within that interval suffices. By sending enough digits to uniquely identify the interval, the decoder can reconstruct the original symbol sequence [8].

Complexity

Arithmetic coding is based on the idea of representing a sequence of characters as a fraction. In order to construct the frequency table, it processes the input data to determine the probabilities of each symbol, which is typically done in O(n), were n is the size of the input data. After that, a cumulative probability distribution is constructed based on the frequency table, also in O(n).

During encoding, the [0, 1] interval is divided into subintervals based on the cumulative probabilities and the interval is updated for each symbol in the input, operation that is generally considered to be done in O(n). During decoding, te symbols are reconstructed from the compressed bitstream using the cumulative probabilities, done in O(n) time, where n is the size of the compressed data.

Advantages and Disadvantages

Advantages:

- High Compression Efficiency: Arithmetic coding typically achieves higher compression efficiency compared to Huffman coding and other entropy encoding techniques. It can represent fractional bits, resulting in more compact codes.
- Adaptability: Arithmetic coding is adaptive, meaning it can dynamically adjust its model during the encoding process based on the observed frequencies of symbols. This adaptability allows for efficient compression of data with varying symbol probabilities.

- No Fixed Code Lengths: Unlike Huffman coding, which uses fixed-length codes, arithmetic coding generates variable-length codes for symbols, minimizing redundancy and improving compression ratios.

Disadvantages:

- Complexity: Arithmetic coding is more complex to implement than simpler methods like Huffman coding. Both encoding and decoding processes involve more intricate mathematical operations.
- Cumulative Probability Update: The cumulative probability values must be continuously updated during the encoding process, which adds computational overhead and complexity

2.3 Run-Length Encoding (RLE)

Description

Run-length encoding is a lossless data compression algorithm in which runs of data – sequences in which the same data value occurs in many consecutive data elements – are stored as a single data value and count, rather than as the original run. Once the messages was transformed, a probability coder (such as the Huffman coder) is used to code both the message values and the count. This is most efficient on that data contains many such runs, such as simple graphic images with a small amount of colours, line drawings, simple animations. However, for files that do not have many such runs, RLE could increase the file size [9].

A real-world use of RLE is for the ITU-T T4 standard for fax machines. Since fax machines transmit black-and-white images, the RLE algorithm can be used optimally. The T4 standard specifies the start color by placing a dummy white pixel at the front of each row so that the first run is always assumed to be a white run. Because we only have black and white runs, the algorithm has a lot of runs of data with the same value, and to account for runs of more than 64, T4 has separate codes to specify multiples of 64 [10].

Complexity

During compression, each symbol is visited once and the consecutive runs of the same symbol are identified. The algorithm then replaces each run with a symbol count and the value itself. Therefore, the operation is done in O(n) time, where n is the size of the input data.

During decompression, the original data is reconstructed by expanding each run according to the count and the corresponding symbol. Decompression is also linear, being done in O(n) time, where n is the size of the compressed data.

Advantages and Disadvantages

Advantages:

- Simple Implementation: RLE is a straightforward and easy-to-implement compression algorithm. It involves counting consecutive identical symbols or values and encoding them as a single symbol/value followed by the count.
- Compression for Repetitive Data: RLE is particularly effective when dealing with data that has long runs of repeated symbols. In such cases, RLE can achieve significant compression ratios.
- Low Computational Overhead: RLE has low computational complexity, making it efficient for encoding and decoding operations. It is suitable for real-time applications or environments with limited computational resources.
- Useful for Simple Graphics: RLE is commonly used in the compression of simple graphics and images, where large areas of the same color or intensity occur.

Disadvantages:

- Inefficiency for Diverse Data: RLE is less effective when applied to data that lacks long runs of repeated symbols. In such cases, the compressed output may end up bigger than the original data.
- Lack of Adaptability: RLE does not adapt to changes in the input data. If the data characteristics shift, RLE may not be the optimal compression method.
- Increased Storage Overhead: In scenarios where runs are short or absent, the overhead of encoding run lengths and symbols individually can lead to increased storage requirements compared to more sophisticated compression algorithms.
- Not Suitable for Complex Data: RLE is not well-suited for continuous data, such as audio or video streams, where consecutive values may not be repeated frequently. For complex images and animations, where there aren't a lot of redundant sections, RLE may make the files bigger rather than smaller.

2.4 Prediction by Partial Matching (PPM)

Description

Prediction by Partial Matching (PPM) is an adaptive statistical data compression algorithm designed to efficiently encode information by leveraging the context of previous symbols in the uncompressed data stream [11]. This method is else called prediction by Markov Model of order n [12]. The fundamental idea of PPM is to predict the next symbol in the sequence based on a partial context of preceding symbols. It achieves

this by dynamically updating its prediction model as it encounters new symbols. In PPM, a contextual probability distribution is maintained, representing the likelihood of each symbol given its context. The escape character serves a crucial role in handling situations where the encoder encounters a new context that has not been seen before. The escape character is defined at each level and for each distinct context at that level, except for the initial one. Its frequency of occurrence is assumed to be equal to the number of distinct characters encountered for a given context at that level in the text encoded so far. When the encoder encounters a new context, it signals the decoder by sending an escape character, indicating that the context length needs to be reduced by 1 [13]. The decoder uses the escape characters as a signal to adjust the probability distribution for zero-length contexts when decoding the subsequent bits.

After we obtain the probability distributions, they can by used by a Huffman or Arithmetic coder to generate a bit sequence, but because the probabilities tend to be high, Arithmetic codes work much better than Huffman codes for this approach.

In terms of decoding, as long as the context precedes the character being coded, the decoder will know the context and therefore know which probability distribution to use.

Complexity

During compression, the frequency tables involve processing the input data and they are done in O(n) time, where n is the size of the input data. However, training the PPM model involves updating and maintaining context tables, and the time complexity depends on the model's order. High-order models may have a higher time complexity, but they often provide better compression. Therefore, the overall time complexity for compression is often expressed as O(n) or O(n * m), where m is the order of the PPM model.

During decompression, reconstructing the frequency tables involves processing the compressed data, which is also done in O(n). Decoding the compressed data involves updating and maintaining the context tables, which leads us to a similar complexity as for the compression, either O(n) or O(n * m).

Advantages and Disadvantages

Advantages:

- Adaptability: PPM is adaptive, meaning it dynamically adjusts its model based on the current input. This allows it to capture and adapt to varying patterns and contexts in the data.
- High Compression Ratios: PPM algorithms, when applied to datasets with predictable patterns, can achieve high compression ratios. They exploit contextual information effectively.
- Context Modeling: PPM models use a context-based approach, considering the previous symbols in the uncompressed stream to predict the next symbol. This allows for efficient modeling of dependencies in the data.

- Variable-Length Coding: Unlike fixed-length coding methods, PPM can represent different symbols with varying numbers of bits based on their probability. This flexibility contributes to better compression.

Disadvantages:

- Computational Complexity: PPM algorithms can be computationally expensive, especially as the context length increases. The time and memory required for encoding and decoding may become impractical for real-time applications or large datasets.
- Memory Usage: PPM requires maintaining a dynamic and potentially large model of the data, leading to high memory usage. This can be a limitation, particularly in resource-constrained environments.
- Initial Learning Period: PPM needs an initial learning period to build its context model. During this phase, its compression performance might not be optimal, and it could take some time to adapt to the data patterns.
- Not Suitable for All Data Types: PPM may not perform well on all types of data. Its efficiency depends on the inherent patterns and regularities present, and it may not be the best choice for highly random or irregular datasets.

2.5 Lempel-Ziv-Welch

Description

The Lempel-Ziv algorithms compress by building a dictionary of previously seen strings. Unlike PPM, which uses a dictionary in order to predict the probability of each character and codes each character individually based on the context, the Lempel-Ziv algorithms code groups of characters of varying lengths. The algorithms can simply be described as follows: given a position in a file, look through the preceding part of the file to find the longest match to the string starting at the current position, then output some code that refers to the match. Lastly, move the cursor past the match.

The Lempel-Ziv-Welch (LZW) algorithm, a variant of LZ78, is a widely used lossless data compression algorithm, typically used in GIF and optionally in PDF and TIFF [14]. When the encoding begins, the dictionary is initialized with entries for each possible byte value (256 entries) and as the algorithm progresses, it identifies repeated sequences of data and dynamically adds them to the code table. The decoder reconstructs the original message by maintaining its own copy of the dictionary and using the received indices to look up the corresponding strings. LZW achieves compression by replacing repetitive sequences with shorter codes, facilitating efficient representation of repeated patterns in the data. The algorithm's efficiency lies in its ability to adaptively update the dictionary during encoding and decoding, striking a balance between compression effectiveness and computational complexity. Despite its widespread use, LZW may face challenges with large dictionaries and potential delays

between the encoder and decoder due to the dictionary update process, but the efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.

Complexity

During compression, the algorithm does through each character in the input file and replaces them with new sequences that it adds to a dictionary. The insertion and the lookup is done in O(1) time, but the dictionary must be reset after it gets full so the time complexity ends up being O(n) time.

During decompression, the algorithm does the hashing on a number, not on a string, so the complexity is also O(n).

Advantages and Disadvantages

Advantages:

- Efficient Compression: LZW is known for its efficient compression capabilities, especially for text and data with repetitive patterns. It can achieve good compression ratios, reducing the size of the compressed file.
- No Need for a prior Knowledge: LZW doesn't require prior knowledge or pre-built dictionaries. The dictionary is built dynamically during the encoding process, allowing for adaptability to the specific content of the data.
- Widely Used in Practice: LZW is widely used and implemented in popular file compression formats such as GIF and the early versions of the UNIX compress utility. Its practical application has contributed to its continued use.
- Fast Decoding: The decoding process in LZW is relatively fast since it involves simple table lookups. This makes it suitable for applications where decoding speed is crucial.
- Handles Variable-Length Strings: LZW encodes strings of varying lengths, which allows it to efficiently represent longer repetitive patterns with shorter codes.

Disadvantages:

- High Memory Usage: LZW requires maintaining a dictionary that grows as more unique strings are encountered. This can lead to high memory usage, making it less suitable for environments with limited memory resources.
- Patent Issues: LZW compression was patented in the 1980s, and for many years its use was subject to licensing fees, which limited its adoption in some applications.

- Compression Speed: LZW compression can be slower than some other compression algorithms, particularly for large files, due to the need to constantly update the dictionary.
- Limited Applicability: LZW compression is particularly effective for text-based data, but may not be as effective for other types of data, such as images or video, which have different compression requirements.

3 Testing

3.1 Chosen Data Set

We tried to pick a variety of tests in order to check what problem each algorithm solves in an optimal way, so the tests vary from text files, csv, json, all the way to different picture, video and audio formats. Since they are all mentioned in detail in the upcoming table, we would like to explain some of the choices we made regarding the files.

Among the text files, we have decided to use multiple alphabets/symbols: just English alphabet (Macbeth), a combination between the English alphabet along with some ASCII art, but also a dictionary containing both Chinese and English. The choice was made with entropy in mind, since Chinese has a higher number of characters than English has letters, so Chinese would therefore have a higher entropy. This way, we wanted to test if the higher entropy that we introduced would actually influence the compression rate.

In terms of images, we picked both compressed and non-compressed formats, in order to see what happens when we apply the compression algorithms on something that was already compressed (png/jpg), as well as test the compression efficiency for bmp and pbm files. Some of the pictures were chosen with certain algorithms in mind, since some of them work better when there are more repeated characters, or more repeated runs of characters. Hence, the tests containing both black and white images, and colored images. We also picked a multitude of audio and video formats in order to test the behaviours.

Lastly, tests 20-23 are created by concatenating multiple books in English, Chinese, Korean and Japanese. This way, they are both conclusive real life examples, since there are people who read e-books quite frequently, but by concatenating more books we managed to create big enough tests of about 600k characters, in order for the results to be more conclusive. These tests were created with Huffman and Arithmetic coding in mind, so that we can more better analyze the effect entropy has on compression. Tests 24 and 25 are also synthetic, with 600k characters, one of them generated manually with just a’s and b’s, another one generated with a script in order to contain a sequence as random as possible. In this way, we are able to test how efficient each algorithm is on a really repetitive test, as well as on a random one.

Table 1 Tests and their corresponding details

Test no.	File details
1	Dictionary containing Chinese words and their English definition
2	Text file containing a multitude of ASCII art (The complete guide to cows)
3	C++ code taken from GeeksForGeeks (huffman.cpp)
4	Valgrind executable binary
5	Csv file containing imdb movie ids and keywords
6	Text file containing Macbeth
7	Bmp picture
8	Jpg picture
9	Png picture
10	Json file containing subtitles from Game of Thrones Season 1
11	Ado – Crime and Punishment – aac format
12	Ado – Crime and Punishment – mp3 format
13	Song of Lament – aiff format
14	Song of Lament – wav format
15	Ado – World Tour Announcement – video mp4 format
16	Ado – World Tour Announcement – video mov format
17	Ado – World Tour Announcement – video avi format
18	Pbm file with text white on black
19	Pbm file with a scenery
20	Entropy test1 using ~600k characters of english literature
21	Entropy test2 using ~600k characters of chinese literature
22	Entropy test3 using ~600k characters of korean literature
23	Entropy test4 using ~600k characters of japanese literature
24	Synthetic file using ~600k very repetitive characters (a's and b's)
25	Synthetic file using ~600k random characters with rand.sh

3.2 PC Specifications

CPU: 11th Gen Intel® Core™ i5-11400H @2.7Ghz
RAM: 16.0GB
GPU: NVIDIA GeForce RTX 3050 Laptop GPU

3.3 Graphs + Interpretation

Compression Ratio

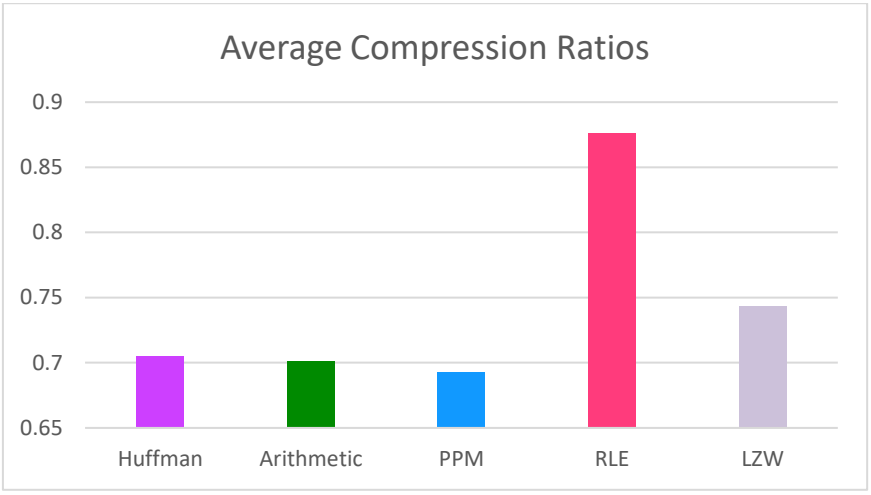
Table 2 Compression ratio by algorithm and test

Test No.	Huffman	Arithmetic	PPM	RLE	LZW
1	0,7341	0,7306	**0,3848**	1,0551	0,5585
2	0,3477	0,344	**0,1982**	0,5317	0,2492
3	0,6015	0,7051	**0,3315**	0,9735	0,4261
4	0,6233	0,6536	0,5538	0,7481	**0,5215**
5	0,5937	0,5902	**0,1587**	1,0158	0,343
6	0,6243	0,6265	**0,3544**	1,0163	0,5298
7	0,9571	0,9547	**0,5113**	0,9721	0,8668
8	**1,0004**	1,0008	1,3403	1,0036	1,2373
9	1,0004	**1,0001**	1,4636	1,0064	1,3649
10	0,5708	0,5702	**0,2449**	0,9233	0,4245
11	0,9991	**0,9968**	1,4382	1,0016	1,3421
12	0,9979	**0,9954**	1,4546	1,0049	1,3456
13	0,8951	**0,8924**	1,1114	0,9876	1,1365
14	0,8952	**0,8924**	1,1114	0,9876	1,1364
15	0,9933	**0,9914**	1,4021	0,9937	1,1325
16	0,9978	**0,9961**	1,4435	0,9975	1,3341
17	0,9967	**0,9944**	1,4177	0,9947	1,322
18	0,1465	0,0514	**0,0414**	0,0843	0,0405
19	0,1853	0,1522	**0,0302**	0,0589	0,0322
20	0,5552	0,5521	**0,3116**	1,0183	0,5146
21	0,7209	0,7177	0,3099	1,0085	**0,5916**
22	0,6632	0,6591	0,2915	1,0062	**0,5257**
23	0,5972	0,5945	**0,2716**	1,0121	0,4957
24	0,1794	0,1251	**0,0753**	0,4735	0,0802
25	**0,7482**	0,7456	1,0564	1,0157	1,0299
Avg. Ratio	0,7049	0,7012	**0,6923**	0,8756	0,7432
Avg. Audio/Video	0,9678	**0,9655**	1,3398	0,9953	1,2498
Avg. Text Files	0,5815	0,5857	**0,3494**	0,9075	0,4838
Avg. Pictures	0,6579	0,6318	0,6773	**0,6250**	0,7083
Avg png/jpg	**1,0004**	**1,0004**	1,4019	1,005	1,3011
Avg other pics	0,4296	0,3861	**0,1943**	0,3717	0,3131

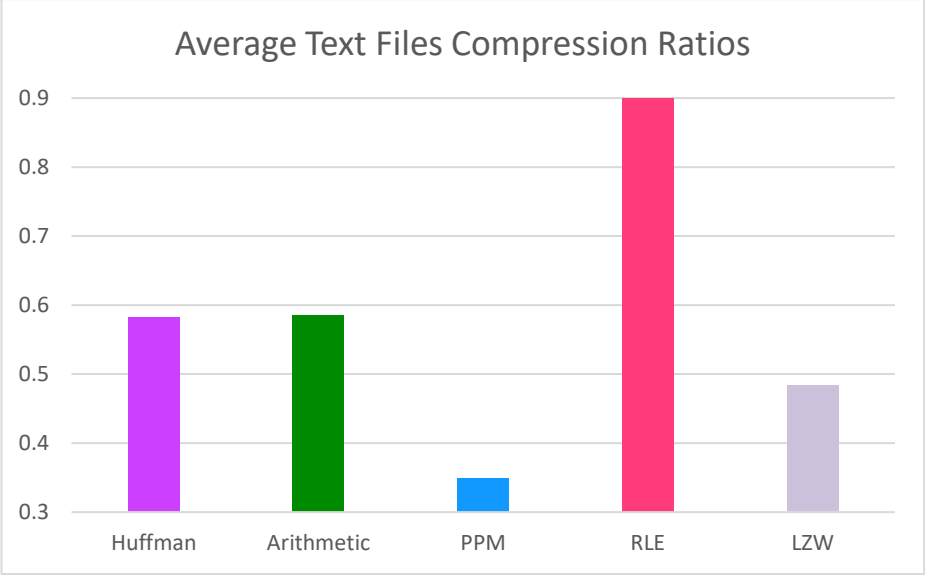
Initial Size vs Compressed Size

Table 3 Size comparison before and after compression

		Huffman	Arithmetic	PPM	RLE	LZW
Test No.	Initial Size	Comp Size	Comp Size	Comp Size	Comp Size	Comp Size
1	2 512 110	1 844 325	1 835 484	**966 779**	2 650 504	1 403 229
2	211 543	73 556	72 774	**41 940**	112 497	52 723
3	7 170	4 313	5 056	**2 377**	6 980	3 055
4	22 680	14 138	14 825	12 562	16 966	**11 829**
5	6 231 943	3 700 099	3 678 382	**989 510**	6 330 742	2 137 655
6	130 414	81 429	81 705	**46 224**	132 543	69 098
7	11 059 338	10 585 824	10 559 095	**5 655 550**	10 751 502	9 586 844
8	511 285	**511 491**	511 712	685 284	513 173	688 904
9	3 752 905	3 754 768	**3 753 309**	5 492 901	3 777 049	5 122 586
10	363 029	207 220	207 035	**88 920**	335 186	154 107
11	4 634 625	4 630 470	**4 620 009**	6 665 642	4 642 290	6 220 481
12	5 793 165	5 781 238	**5 766 606**	8 426 522	5 822 027	7 795 811
13	8 953 438	8 015 107	**7 990 451**	9 951 213	8 842 998	10 175 608
14	8 953 530	8 015 202	**7 990 546**	9 951 283	8 843 104	10 175 608
15	2 391 568	2 375 728	**2 371 134**	3 353 440	2 376 626	3 169 284
16	3 560 708	3 553 121	**3 547 155**	5 140 175	3 551 919	4 750 692
17	3 611 230	3 599 582	**3 591 351**	5 119 646	3 592 358	4 774 166
18	453 018	66 403	23 317	**18 766**	38 208	18 377
19	1 340 053	248 389	204 068	**40 535**	78 996	43 263
20	607 099	337 109	335 157	**189 183**	618 238	312 466
21	2 623 402	1 891 449	1 882 812	813 068	2 645 823	**1 552 025**
22	1 484 901	984 854	978 694	432 976	1 494 112	**780 708**
23	1 855 113	1 107 999	1 102 926	**503 889**	1 877 564	919 761
24	603 485	108 301	75 471	**45 487**	285 780	48 413
25	600 005	448 960	**447 598**	633 874	609 427	617 997

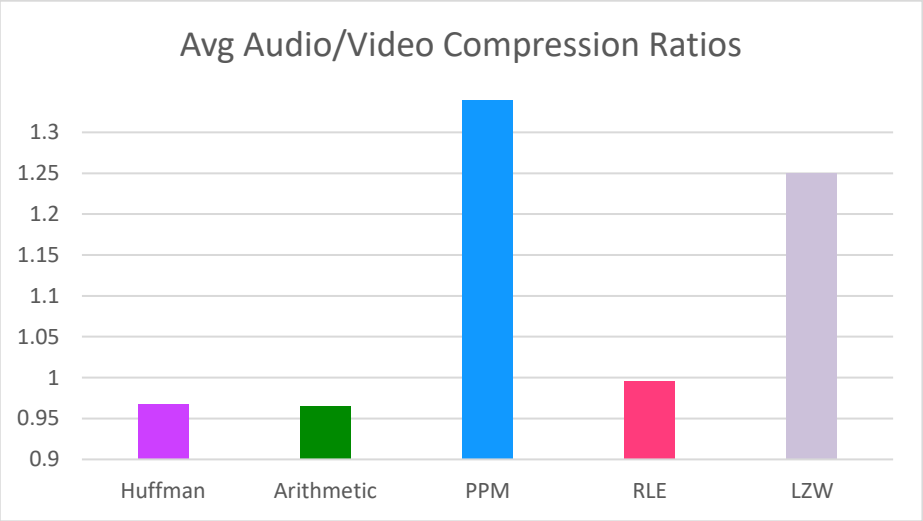


In the above table we can see the compression ratios for each algorithm, as well as the average ratio for all the file types, and the average ratios for similar file types. In the second table we decided to include a more precise overview of the file sizes – the initial size and the compressed one. Since the image compression had a very different behaviour depending on the file type, we decided to compile both the average ratio for png/jpg, and for the other types as well. Because of the multiple file types, analyzing the average ratio isn't so valuable, so we will delve deeper and analyze the behaviour for each type. However, at a simple glance we can see that most algorithms achieve a pretty similar compression overall, with PPM being the best, and RLE being the one with the least compression ratio. Huffman and Arithmetic coding are both Entropy Coding algorithms so we can see that they behave in a really similar way, while LZW is somewhere in the middle.

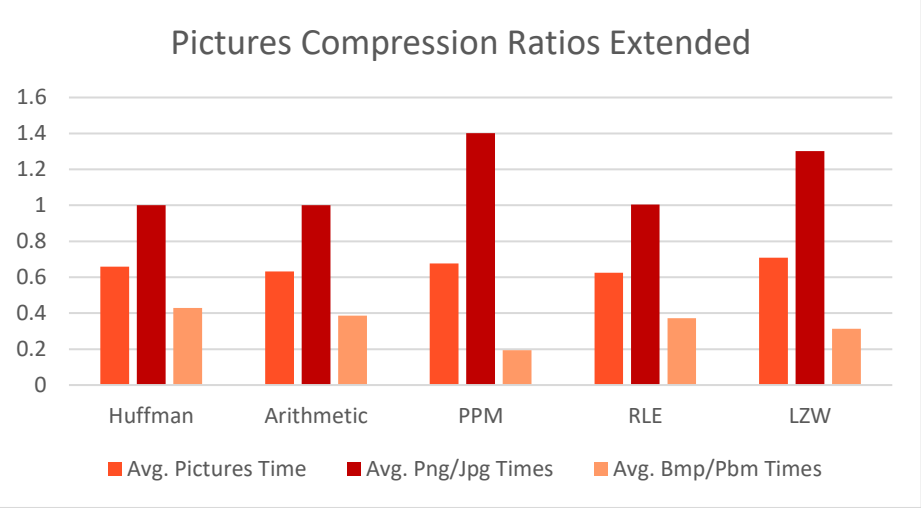


We can easily see that text files have the best compression ratio: on average, the compressed file size is lower than the initial size, in most cases. Since RLE sometimes produces a bigger file when there aren't many identical runs, its compression ratio is the lowest, and PPM is clearly in the lead. The algorithms act the same way as they did when we analyzed the average data based on all the file types, except we can more clearly see the compression ratio.

Along with these text files we also have two special ones, synthetic tests that were generated especially to prove that files with repetitive sequences have a better compression ratio. The 24th test was made using only the characters 'a' and 'b' and has a total of 600 000 characters, regardless it was compressed to ~7% of its original size by the PPM algorithm, which is by far one of the biggest compressions we've seen in this analysis, along with the pbm files, which are black and white images containing only pixels with 0 and 1 values, so they only prove this point further. The 25th test was the exact opposite of the 24th, containing 600 000 randomly generated characters (using the script from the input folder), and has the worst compression ratio out of all the text files tested, with a size of ~74% of the original file, exceeding even literature texts.



When it comes to audio and video files, however, the situation takes a huge twist, since PPM, who was before in the lead, now has the worst compression ratio, making the file much bigger. This might be because the partial matching doesn't work really well for audio and video files, since they might be harder to predict. Huffman and Arithmetic coding are leading, having once again a very similar compression rate, actually managing to compress the file, although the differences are really small. The difference in compression is even smaller when it comes to RLE, but the algorithm is now performing second best after the Entropy coding ones, actually managing to slightly compress the file almost every time. LZW is still somewhat in the middle, but, just like PPM, it doesn't work properly on audio or video files, making the result file larger.

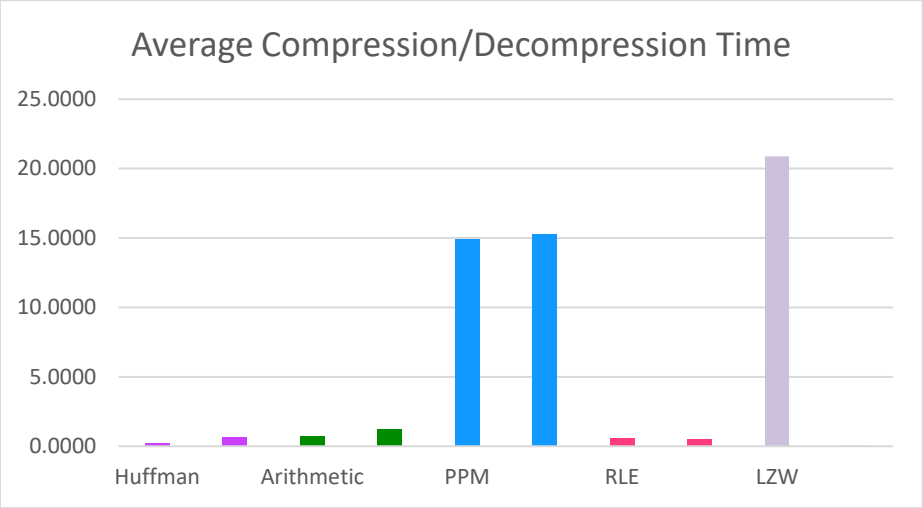


Last, but certainly not least, we have the images. For these, we decided to separate the png and jpg files, since they are already compressed, so applying a compression algorithm on them will most of the times have virtually no effect (when it comes to Huffman, Arithmetic and RLE), or even enlarge the file (with PPM, LZW). However, bmp and pbm files are compressed in a much more effective way, with PPM being the fastest on average, LZW second, followed closely by RLE and Arithmetic coding, with Huffman coming last in terms of the compression ratio. It was interesting seeing how different the compression ratio was between already compressed formats and non-compressed once, the difference being so drastic that we actually ended up getting a decent compression ratio on average, even when including png and jpg formats in our calculations.

Compression and Decompression Times

Table 4 Compression and Decompression Times by Algorithm

	Huffman		Arithmetic		PPM		RLE		LZW	
Test No.	Comp Time	Decomp Time	Comp Time	Decomp Time	Comp Time	Decomp Time	Comp Time	Decomp Time	Comp Time	Decomp Time
1	**0.1863**	0.4933	0.4603	0.8384	6.5642	6.9110	0.3344	0.3543	16.974	**0.0421**
2	**0.0163**	0.0240	0.0365	0.0622	0.4533	0.4890	0.0356	0.0196	1.3349	**0.0038**
3	**0.0034**	0.0051	0.0043	0.0047	0.0328	0.0310	0.0042	0.004	0.0289	**0.0016**
4	**0.0046**	0.0096	0.0078	0.0150	0.1429	0.1690	0.0068	0.0059	0.1129	**0.0027**
5	**0.4087**	1.0419	1.3552	2.3026	10.5144	12.6569	0.8938	0.9139	39.9304	**0.0767**
6	**0.0167**	0.0295	0.0343	0.0471	0.3956	0.3507	0.0349	0.0342	1.0782	**0.0042**
7	**0.8532**	2.6968	3.1588	5.3327	46.2891	43.6327	2.1965	2.1941	59.2001	**0.2351**
8	**0.0521**	0.1432	0.1958	0.2414	5.8616	5.7312	0.0845	0.1024	3.7724	**0.0162**
9	**0.3627**	1.0354	0.9942	1.4134	30.2931	33.8648	0.8072	0.7398	26.5479	**0.1117**
10	**0.0289**	0.0589	0.0710	0.1155	0.8464	0.8716	0.0822	0.0747	2.6037	**0.0087**
11	**0.3689**	1.3468	1.0919	1.8530	38.4450	37.3676	0.9331	0.9355	33.5926	**0.1107**
12	**0.4551**	1.5968	1.4479	3.1612	48.2671	49.4226	1.1614	1.0409	39.7353	**0.1451**
13	**0.6663**	2.1535	2.5094	4.5007	43.8046	48.2771	1.4526	1.5259	82.1342	**0.2232**
14	**0.6709**	2.1368	1.9164	3.4800	45.3873	45.6218	1.8899	1.7403	78.966	**0.2048**
15	**0.1848**	0.6538	0.5898	0.9677	18.6164	19.4473	0.4489	0.493	16.5511	**0.0687**
16	**0.2853**	0.9809	0.8451	1.6593	27.4726	27.2144	0.6736	0.7122	25.66	**0.092**
17	**0.3026**	1.0193	1.2382	2.0394	29.4308	28.2026	0.67717	0.7728	26.6928	**0.0848**
18	**0.0241**	0.0194	0.0645	0.1617	0.8546	0.8057	0.0713	0.0179	1.2098	**0.0051**
19	**0.0635**	0.0568	0.1697	0.4799	2.1761	2.3986	0.2372	0.0447	4.5351	**0.0108**
20	**0.0460**	0.0932	0.1615	0.2641	1.1682	1.2508	0.1358	0.1324	4.7633	**0.0109**
21	**0.1957**	0.5316	0.5772	0.9167	5.4011	5.8737	0.5803	0.4982	21.2357	**0.0505**
22	**0.1064**	0.2769	0.3205	0.6343	2.8373	3.1997	0.3361	0.3546	11.6776	**0.0269**
23	**0.1260**	0.3218	0.5386	0.6641	3.3050	3.5628	0.3389	0.4192	14.5502	**0.0408**
24	**0.0347**	0.0307	0.0739	0.1448	0.9630	1.0297	0.1273	0.0674	3.3477	**0.0066**
25	**0.0533**	0.1288	0.1242	0.2185	2.9399	3.3905	0.1375	0.1216	5.7534	**0.0144**
Avg. Time	0.2207	0.6754	0.7195	1.2607	14.8985	15.2709	0.5472	0.5328	20.8795	0.0639
Avg. Audio/Video	0.4192	1.4126	1.3770	2.5230	35.9177	36.5076	1.0338	1.0315	43.3331	0.1328
Avg. Text Files	0.0944	0.2343	0.2896	0.4791	2.7357	3.0605	0.2344	0.2308	9.4916	0.0223
Avg. Pictures	0.2732	0.7689	0.9565	1.6648	12.0460	11.9689	0.6867	0.6410	21.1907	0.0664



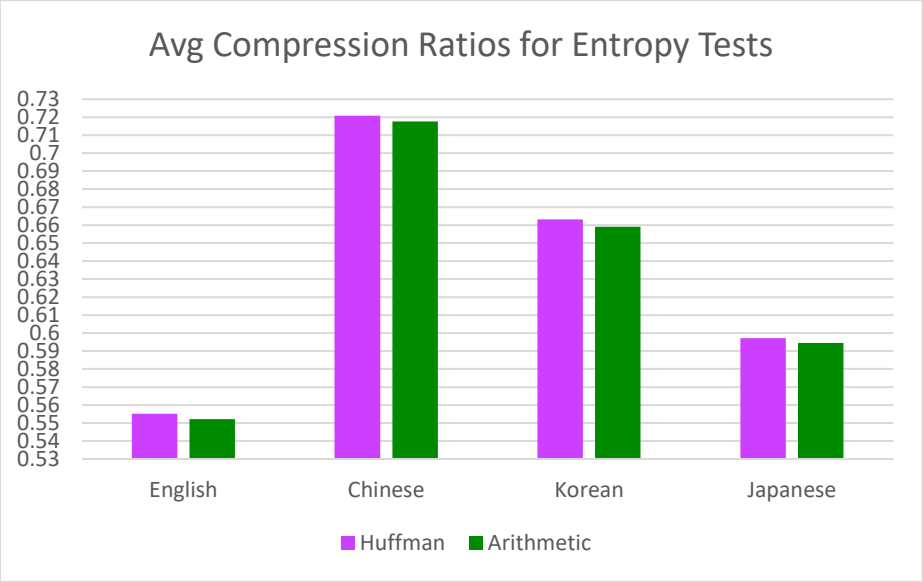
When it comes to the compression and decompression, the statistics stay consistent among all file types, so we no longer see the algorithms very so much according to the format of the test. We can easily see that Huffman Coding has the best compression time, while LZW has the best decompression time, both of them happening in virtually almost no time at all, the two lines of the graph being barely visible. Although the algorithms stay consistent in comparison to each other, the average compression and decompression times do vary according to the file format: we can see that the text files are the fastest, followed by images, and lastly followed by audio and video formats. Most of the compression is done quite quickly on all file formats, but we can see that PPM and LZW encounter some problems when in comes to the compression/decompression, and compression, respectively in case of LZW.

Entropy Analysis for Huffman Coding and Arithmetic Coding

The last comparison criteria can only be applied to the Entropy coding algorithms, our theory being that the higher the alphabet entropy, the harder the compression will be. Therefore, we expect to obtain a better compression ratio for alphabets with lower entropy.

Tabel 5 Compression Ratios for the entropy tests

Test No.	Huffman	Arithmetic
20	0,5552	0,5521
21	0,7209	0,7177
22	0,6632	0,6591
23	0,5972	0,5945



First of all, when it comes to the two algorithms, they both yield pretty similar results, the compression ratio for Huffman Coding being only slightly better. Second of all, our theory regarding the relation between compression and entropy proves to be true, since English has the best compression ratio, having the lowest entropy because of its rather limited characters. Japanese follows it, since it has more characters but still a form of simplicity to it. Since an upper bound estimate for Korean entropy is 6.01 bits per Korean syllable [15], it doesn't come as a surprise that the Korean language has the third best compression ratio. The hardest to compress is definitely Chinese, since the Shannon entropy for the Chinese language is of 9.59 bits per character [16]. The difference between its compression ratio obtained compared to especially the English one being really noticeable. Regardless of the language, the algorithms still do a pretty good job at achieving a reasonable compression, despite the challenges these more complex languages pose.

3.4 Conclusions

As data continues the grow in both complexity and volume, compression algorithms became a key component that allows us to optimize the way we store and transmit data. In contemporary practices, a combination of algorithms is often used, leveraging their individual traits in order to provide better compression results.

In conclusion, the analysis of various data compression algorithms reveals nuanced considerations for practical applications. Huffman Coding, with its efficient prefix codes, proves beneficial for scenarios where simplicity and speed are paramount, making it suitable for general-purpose compression tasks. Arithmetic Coding, though computationally more intensive, excels in achieving higher compression ratios, mak-

ing it ideal for applications where space optimization is critical, albeit at the cost of increased processing time.

Prediction by Partial Matching (PPM) showcases strengths in handling repetitive patterns, making it a valuable choice for compressing text and certain structured data. Run-Length Encoding (RLE) is particularly effective in compressing sequences of repeated characters, finding applications in scenarios where redundancy is prevalent.

Lempel-Ziv-Welch (LZW), renowned for its adaptability to various data types, is widely used in practice, especially in file compression formats like GIF and ZIP. Its versatility and efficiency in handling diverse data sets make it a popular choice for real-world applications.

Choosing the appropriate algorithm depends on the specific requirements of the task at hand. For general-purpose compression, Huffman Coding and LZW are commonly employed. In situations where high compression ratios are imperative, Arithmetic Coding may be preferred despite its computational overhead. PPM and RLE find niche applications where their respective strengths align with the characteristics of the data.

References

1. https://en.wikipedia.org/wiki/Prefix_code
2. https://en.wikipedia.org/wiki/Kraft%E2%80%93McMillan_inequality
3. https://en.wikipedia.org/wiki/Huffman_coding#cite_note-5
4. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3
5. https://en.wikipedia.org/wiki/Canonical_Huffman_code
6. https://www.geeksforgeeks.org/huffman-decoding/
7. https://www.cs.cmu.edu/~15750/notes/compression_guy_blelloch_book_chapter_draft.pdf
8. https://en.wikipedia.org/wiki/Arithmetic_coding
9. https://en.wikipedia.org/wiki/Run-length_encoding
10. https://www.cs.cmu.edu/~15750/notes/compression_guy_blelloch_book_chapter_draft.pdf
11. https://en.wikipedia.org/wiki/Prediction_by_partial_matching
12. https://compressions.sourceforge.net/PPM.html
13. https://www.cs.ucf.edu/courses/cap5015/ppm.pdf
14. https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique
15. https://doi.org/10.1093/llc/11.3.141
16. https://www.johndcook.com/blog/2019/10/18/chinese-character-entropy/
17. Macbeth: https://www.gutenberg.org/ebooks/1533
18. Keywords: https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset?resource=download&select=keywords.csv
19. Cows: http://artscene.textfiles.com/asciiart/
20. Jpeg: https://ro.pinterest.com/pin/179158891417358927/
21. Json: https://www.kaggle.com/datasets/gunnvant/game-of-thrones-srt?select=season1.json
22. Dictionary: https://github.com/ssb22/CedPane
23. Binary: https://onlinefiletools.com/generate-random-binary-file
24. https://bgibhopal.com/blog/different-types-of-data-compression-algorithms-and-their-applications/
25. https://www.cs.cmu.edu/~15750/notes/compression_guy_blelloch_book_chapter_draft.pdf