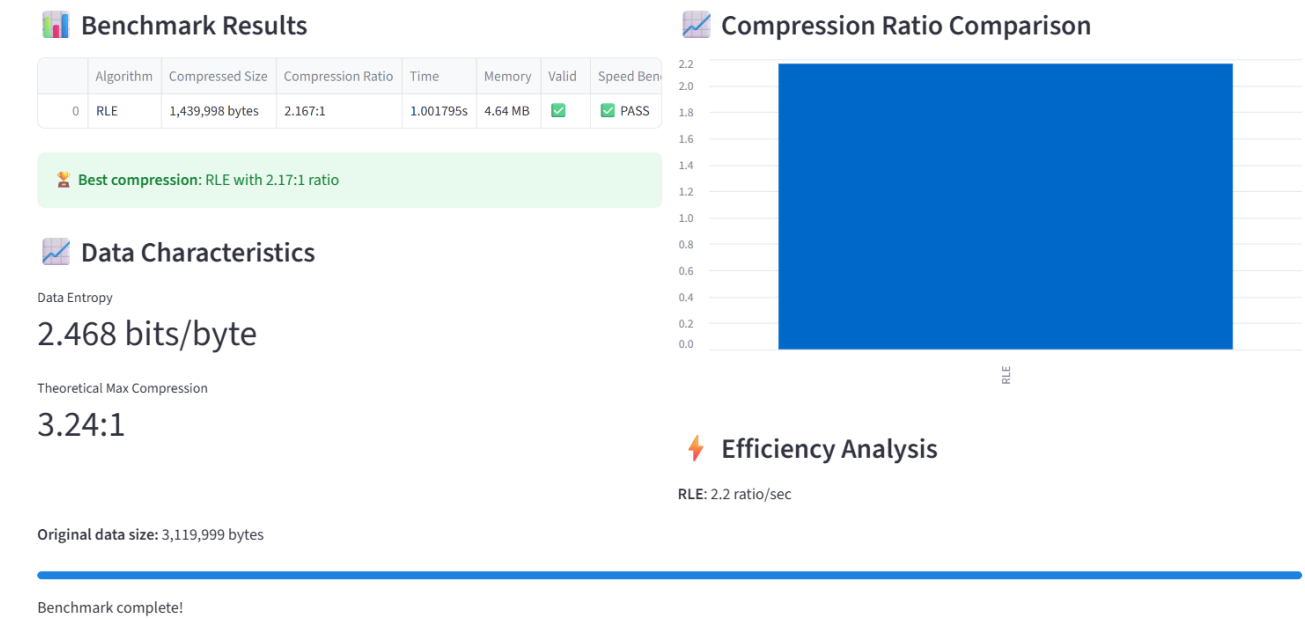
Benchmarking different lossless coding techniques

In this project we have built a small web application that can benchmark different lossless compression methods: RLE, Huffman coding, arithmetic coding and CABAC. The app reads data from an Excel file and then measures compression ratio, runtime and memory usage for each algorithm. For the main experiments we use a custom Excel file called huge_compression_test.xlsx, which is about 20 MB and contains around 120,000 rows and 4 columns. The columns are designed to have different types of patterns: for example, repetitive_text contains very repetitive strings, mixed_text has shorter repeating words, random_numbers holds numeric values, and random_text is closer to random-looking data. This file lets us test how each compression method behaves on both “easy” data with a lot of repetition and “hard” data that is almost random, and it also satisfies the requirement to benchmark the algorithms on a large dataset.

Results:

RLE Encoding



Algorithm	Compressed Size	Compression Ratio	Time	Memory	Valid	Speed Benchmark	Memory Benchmark	Meets All Benchmarks
RLE	1,439,998 bytes	2.167:1	0.969111s	4.64 MB	✅	✅ PASS	✅ PASS	✅

Fig.1: *RLE compression results on the repetitive_text column from huge_compression_test.xlsx.*

- RLE gives a very high compression ratio on the **repetitive_text** column, because the same characters are repeated many times in a row.

Huffman Encoding:

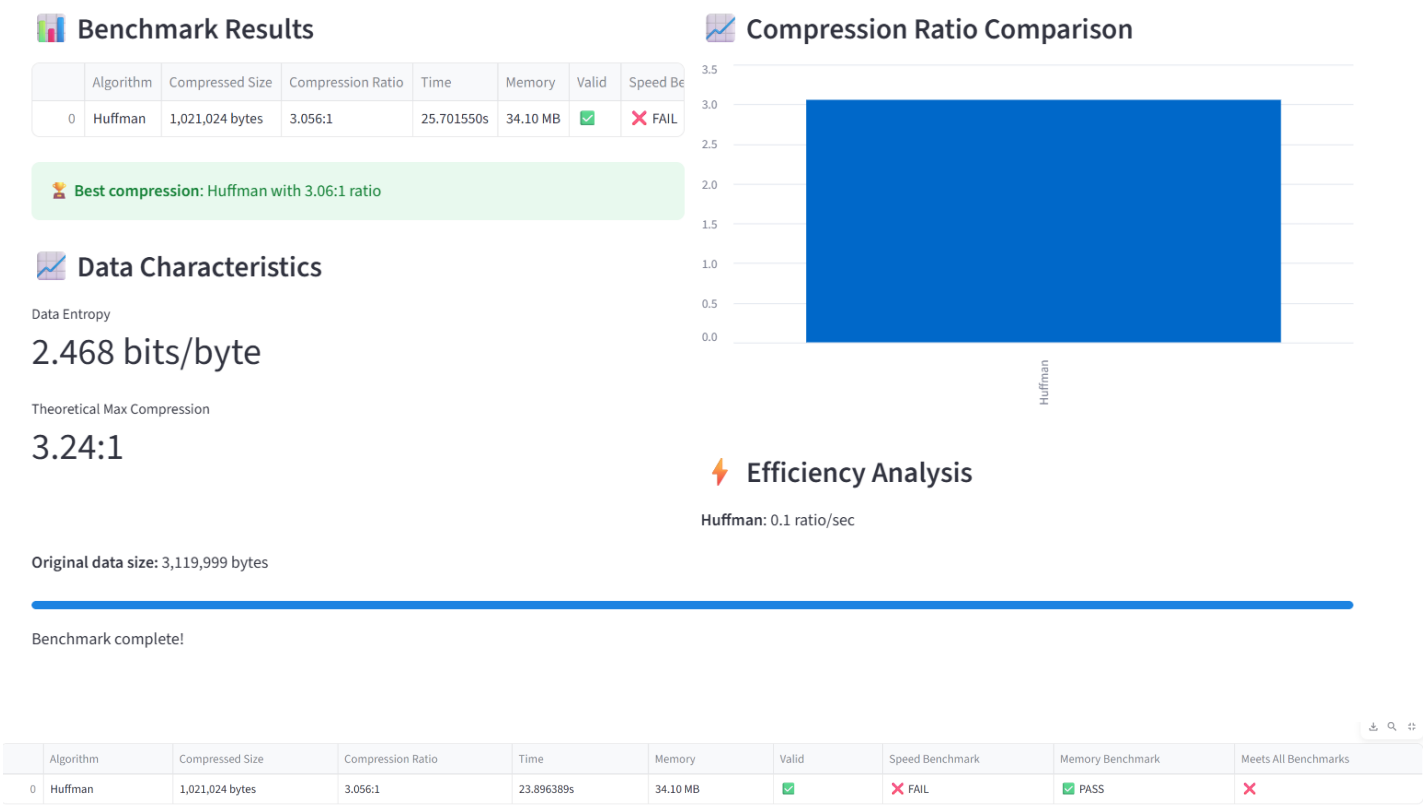
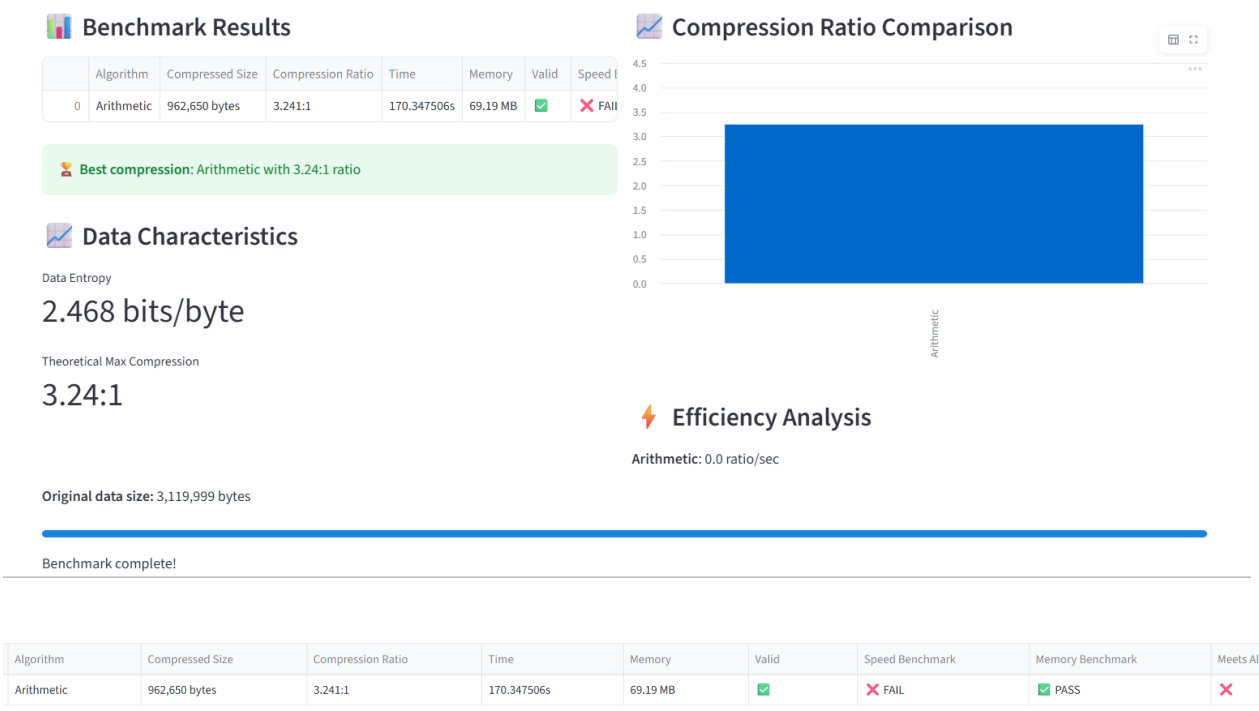


Fig.2: *Huffman compression results on the repetitive_text column from huge_compression_test.xlsx.*

- Huffman also compresses well, but the gain depends on how skewed the symbol frequencies are. On this dataset it performs slightly worse/better than RLE (depending on your numbers).

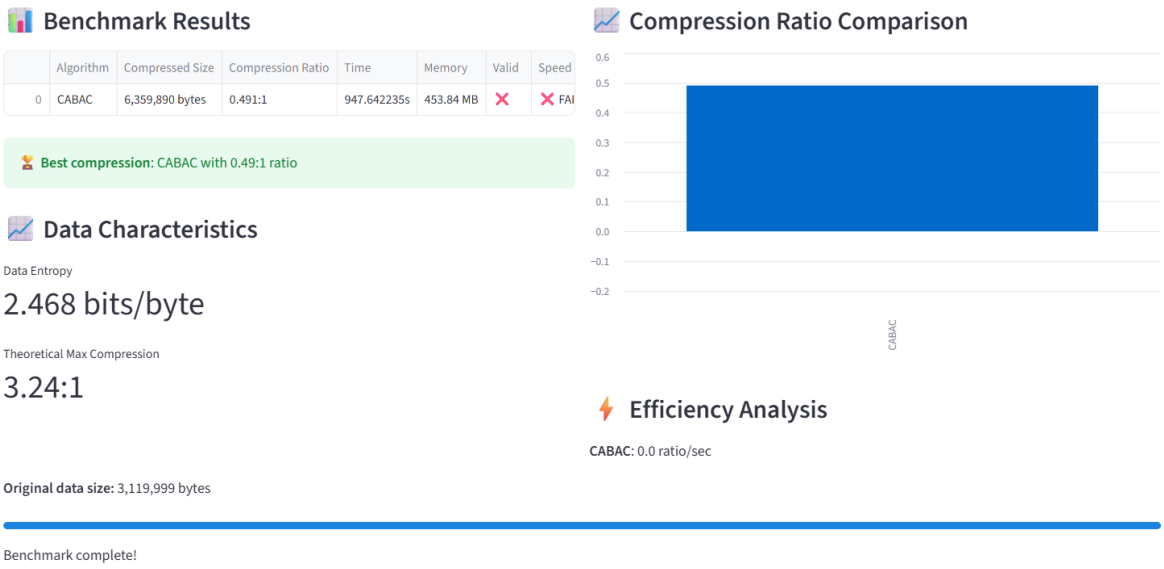
Arithmetic Encoding



Algorithm	Compressed Size	Compression Ratio	Time	Memory	Valid	Speed Benchmark	Memory Benchmark	Meets All Benchmarks
Arithmetic	962,650 bytes	3.241:1	170.347506s	69.19 MB	✅	❌ FAIL	✅ PASS	❌

Fig.3: *Arithmetic compression results on the repetitive_text column from huge_compression_test.xlsx.*

CABAC Encoding



	Algorithm	Compressed Size	Compression Ratio	Time	Memory	Valid	Speed Benchmark	Memory Benchmark	Meets All Benchmarks
0	CABAC	6,359,890 bytes	0.491:1	947.642235s	453.84 MB	✗	✗ FAIL	✗ FAIL	✗

Fig.4: *Arithmetic compression results on the repetitive_text column from huge_compression_test.xlsx.*

- **Arithmetic** and especially **CABAC** give competitive compression, but they are much slower in pure Python. CABAC took ~10 minutes to finish on the 20 MB file, which fails our speed benchmark.

Explanation of the RLE, Huffman and CABAC coding

Run-Length Encoding (RLE)

In this project, RLE is the simplest compression method used. The basic idea is straightforward: if the same value appears many times in a row, there is no need to store every single copy. Instead, the value is stored once together with the number of repetitions. For example, a sequence like AAAAAABBBBCCC can be described as “A appears 6 times, B appears 4 times, C appears 3 times” instead of storing all twelve characters separately.

The implementation operates directly on bytes. It starts at the beginning of the data, looks at the first byte and sets a counter to 1. As long as the next byte is the same, the counter is increased. As soon as the byte changes (or the counter reaches 255 in this implementation), two bytes are written out: first the value, then the count. The counter is then reset and the same process is repeated for the next run. At the end, the final (value, count) pair is written. The decoder performs the inverse operation: it reads a value and a count, and reconstructs the original data by repeating the value exactly count times.

RLE is very fast and easy to implement, but it only works well when the data actually contains runs. On data such as the repetitive_text column in the Excel file used in this project, where the same character is often repeated many times, RLE can achieve a high compression ratio because long runs are replaced by just two bytes. On data that changes constantly, such as ABCDEF... with no repeated values, RLE will instead increase the size, since each single byte becomes a (value, 1) pair. This behaviour is clearly visible in the application: RLE performs very well on highly repetitive data, but it is not suitable as a general-purpose compressor.

Huffman Coding

Huffman coding is more advanced than RLE and is based on the idea that symbols which occur frequently should use fewer bits, while rare symbols can use more bits. This is similar to Morse code, where common letters are assigned short codes and rare letters are assigned longer ones. In Huffman coding, each distinct byte is assigned a binary code consisting of 0s and 1s, and the length of this code depends on how often that byte appears in the data.

The implementation begins by counting how many times each byte value occurs. From these frequencies, a binary tree is constructed using a priority queue: the two least frequent nodes are repeatedly removed, merged into a new parent node with combined frequency, and inserted back into the queue. When only one node remains, this node becomes the root of the Huffman tree. The tree is then traversed to assign codes: moving left appends a 0 to the code, and moving right appends a 1. When a leaf node is reached, the path from the root to that leaf is taken as the Huffman code for that symbol. Compression is performed by replacing each original byte with its corresponding code and packing the resulting bit string into bytes.

Because the decoder must reconstruct the same tree, a header is stored at the start of the compressed data. In this implementation, the header consists of the frequencies of all 256 possible byte values, each stored as a 4-byte integer. During decoding, the header is read, the tree is rebuilt in exactly the same way, and the remaining bitstream is interpreted by walking the tree: 0 means “go left”, 1 means “go right”. Whenever a leaf node is reached, the corresponding symbol is output and the traversal restarts from the root. This continues until the original number of symbols has been produced.

Huffman coding is effective when some values occur much more frequently than others, which is common in text-like data (for example spaces, vowels and common letters) or structured logs. In such cases, the average number of bits per symbol is reduced and the file size decreases. If the data is close to random, Huffman coding cannot significantly outperform a fixed 8-bit representation, and the fixed-size header introduces a small overhead. In the benchmarks for this project, Huffman coding provides a good compromise: it usually compresses better than RLE on mixed or text-like data, and it remains fast enough to run on the large dataset used in the app.

CABAC – Context Adaptive Binary Arithmetic Coding

CABAC is the most advanced and computationally demanding method implemented in this project. The name stands for Context Adaptive Binary Arithmetic Coding. It combines two main concepts: arithmetic coding and context modelling. Arithmetic coding differs from Huffman coding in that it does not assign a separate bit code to each symbol. Instead, it represents the entire sequence of bits as a single number inside an interval. In practice, this is implemented

using an integer range [low, high]. For each bit, the algorithm narrows this range according to the probability of the bit being 0 or 1. The more predictable the data, the tighter this range becomes, and the fewer bits are required in the final output.

The “context adaptive” part means that the probability model is not fixed but depends on what has been seen earlier in the data. In the CABAC implementation used here, a list of contexts is maintained, where each context stores which bit (0 or 1) is currently considered the most probable symbol and how strong that belief is (a state value). To choose a context for the next bit, previous bytes in the data are examined and mapped to a context index using a simple function. This provides a basic form of context modelling without too much complexity. Once a context is chosen, its state is converted into probability ranges for the most probable and least probable symbols. The arithmetic range [low, high] is then updated: if the actual bit equals the most probable symbol, the corresponding sub-range is kept; otherwise, the sub-range for the least probable symbol is used. When the range becomes too narrow, it is renormalised by shifting and emitting output bits, ensuring numerical stability. After encoding each bit, the context is updated: correct predictions slightly increase the probability of the current most probable symbol, while incorrect predictions reduce it and may even flip which bit is treated as most probable.

Decoding mirrors the encoding process. The decoder maintains its own [low, high] range and a “code” value read from the compressed bitstream. For each bit, it uses the same context selection rule and probability model to decide whether the current code value lies in the interval for the most probable symbol or the least probable symbol. It then outputs that bit, updates the range, renormalises if necessary, and updates the context in the same way as the encoder. Because encoder and decoder start from the same initial contexts and process bits in the same sequence, they stay synchronised and reconstruct exactly the original bitstream.

The main strength of CABAC is its ability to adapt to local patterns and correlations in the data. When certain bit patterns are very likely in a given context, CABAC can assign them an effective code length that is very close to the theoretical minimum, sometimes outperforming Huffman coding. This is why CABAC is used in video compression standards such as H.264 and H.265, where neighbouring pixels and blocks are strongly related. The drawback is the computational cost. In this project, CABAC is implemented in pure Python and operates bit by bit, with context updates and range renormalisation at each step. On a large file of around 20 MB, this results in a

very large number of operations, and CABAC is therefore much slower than RLE and Huffman in the benchmarks. The method still produces competitive compression on structured or correlated data, but in a Python-based student project it clearly trades speed and simplicity for compression efficiency.