Project Milestone: Finite State Entropy (FSE)

Student: Aayush Gupta
Mentor: Pulkit
Code repo: https://github.com/aayushg55/stanford_compression_library
Milestone report: https://github.com/aayushg55/stanford_compression_library/blob/main/project_milestone.md

1. Introduction

Modern lossless compressors depend critically on the entropy coding stage: given an LZ-style or predictive front-end, the entropy coder must convert a stream of symbols to bits at rates close to the Shannon limit while remaining fast enough not to bottleneck the pipeline. Classical choices include Huffman coding, which is extremely fast but restricted to integer code lengths, and arithmetic or range coding, which achieves near-optimal rates but is traditionally slower and more complex.

Finite State Entropy (FSE) is a table-based realization of Asymmetric Numeral Systems (ANS). Instead of maintaining an interval (as in arithmetic coding) or emitting per-symbol codewords (as in Huffman), ANS keeps a single integer "state" that jointly encodes the past bitstream and the next symbol to be decoded. FSE, used in Zstandard, is a particular tANS design in which decoding reduces to a constant-time table lookup plus a few bit operations, giving compression ratios comparable to arithmetic coding and speeds closer to Huffman.

This project aims to reimplement and analyze FSE in a pedagogical setting. First, I implement a pure-Python FSE encoder/decoder inside the Stanford Compression Library (SCL) as a readable, well-documented reference. Then I port the implementation to C++ and incrementally add low-level optimizations inspired by Yann Collet's FiniteStateEntropy library and Zstandard with the goal to understand how ANS moves from theory to production-grade code and how each micro-optimizations contributes to compression ratio and speed.

---

2. Literature and Code Review

Jarek Duda's work on ANS shows how to match arithmetic coding's compression efficiency using a single integer "state" instead of maintaining an interval 1. In ANS, encoder and decoder share a mapping between symbols and ranges of states; the probability of a symbol is reflected in how many state values are assigned to it. There are two main practical realizations: range-based ANS (rANS), which behaves much like range coding and uses integer multiplications and reciprocal-based divisions in its inner loop, and table-based ANS (tANS). tANS, and FSE in particular, push that arithmetic into precomputed tables so that each encoding or decoding step reduces to a table lookup, a few bit operations, and an addition.

Yann Collet's Finite State Entropy (FSE) is a practical tANS variant engineered for speed to avoid costly multiplications and divisions, giving strong performance on both old and modern CPUs. Collet's blog posts 2 describe the core construction: symbol counts are normalized so their sum is a power of two, `2^tableLog`; symbols are "spread" across the state space using a co-prime step so their occupancy matches these normalized weights; and a decode table is built where each entry stores a symbol, a number of bits to read, and a base for the next state. Decoding a symbol consists of a single table lookup, outputting the symbol, reading `nbBits` from the bitstream, and adding those bits to the stored base to form the next state. Encoding runs this process in reverse: starting from a final state, it processes the message backwards, flushes low bits as needed, and uses a shared state table plus per-symbol transforms to update the state. The number of bits consumed for each symbol is tied to its probability: the decoder reads either `k` or `k+1` bits, arranged so that the average bits per symbol is a non-integer value consistent with the symbol's optimal Shannon code length.

The FiniteStateEntropy GitHub repository is Collet's optimized C implementation of FSE 3. Internally, it includes routines that normalize raw counts to signed "normalized counters," build encoder and decoder tables, and compress data using a precomputed table. These structures closely follow the blog descriptions and serve as a concrete target for my C++ port.

Zstandard (zstd) is a widely used lossless compressor that combines an LZ front end with a fast entropy stage based on Huffman (Huff0) and FSE 4. Within SCL, there are already several pure-Python entropy coders—Huffman, rANS, tANS, arithmetic—as well as wrappers around external libraries such as zlib and Zstandard 5. These provide both reference and baselines for compression ratio and throughput.

---

3. Methods

This project has three main threads. First, I will implement a clear FSE encoder/decoder in pure Python inside the Stanford Compression Library (SCL), aimed at being a readable, pedagogical reference rather than a maximally tuned codec, similar to the existing tANS or rANS implementations. Second, I will port this implementation to C++ and progressively add low-level optimizations—mirroring the structure of Collet's FSE and Zstandard's entropy stage—so that the C++ version approaches production-grade performance while staying faithful to the Python reference. Third, I will evaluate both versions quantitatively, comparing FSE against SCL's existing entropy coders (Huffman, rANS, tANS) and against zlib/zstd on standard benchmark corpora. For example, I will use the Canterbury and Silesia corpora spanning text, structured data, images, etc.

The end result I aim for is a pair of implementations (Python and C++) with identical behavior and similar compression ratios, plus a benchmark harness that reports, for each codec and dataset, the average bits per byte, compression ratio, and encode/decode throughput. Final results will be summarized in tables and plots showing the compression–speed trade-off and how performance varies across data types (text, structured data, images, executables), and how the various low-level optimizations contribute to performance. The project will be considered successful if the optimized C++ FSE significantly outperforms the Python baseline in throughput while maintaining correctness and equal or better compression ratios, while also narrowing the gap to mature libraries like zstd's FSE.

---

4. Progress Report

4.1 Completed So Far

So far, I have completed the theoretical groundwork, the Python implementation, and an initial round of testing and benchmarking. On the theory side, I have read and summarized Duda's ANS work and Collet's FSE blog posts, focusing on tANS/FSE's table construction, symbol spreading, and encode/decode loops. I have also inspected the FiniteStateEntropy C code to see how these ideas are realized in practice and how Zstandard wires FSE into a full compressor.

On the implementation side, I have added a pure-Python FSE encoder and decoder to SCL that integrate with its existing `DataEncoder/DataDecoder` interfaces and use `Frequencies` and `DataBlock` for modeling (FSE implementation). The implementation includes: normalization of symbol counts to a power-of-two table size; an FSE-style spreading routine driven by a co-prime step; construction of the decode table with (symbol, nbBits, newStateBase) entries and state-dependent nbBits; and construction of the encode tables, including the shared next-state table and per-symbol (deltaNbBits, deltaFindState) transforms. These components together implement a full tANS/FSE pipeline inside SCL. There is also a suite of unit tests to verify each component's correctness.

I have also run preliminary benchmarks. On synthetic small-alphabet distributions, preliminary results show that FSE matches the other ANS variants (rANS, tANS) in compression ratio and decodes faster than the existing pure-Python rANS and tANS implementations, while remaining slower than the pure-Python Huffman coder. On files from the datasets, with files treated as byte streams, the zlib and zstd wrappers are orders of magnitude faster than any pure-Python implementation and often achieve slightly better compression ratios, as expected. Overall, the basic Python goal has been reached: FSE matches other ANS variants in compression behavior, is faster than the existing pure-Python rANS/tANS implementations, is slower than Huffman, and still far from the performance of optimized C-based coders.

4.2 Plan for Remaining Weeks

In the remaining weeks, I plan to (i) finish and validate the C++ FSE port, (ii) add micro-optimizations one at a time and measure their impact, and (iii) run a more comprehensive benchmark suite on standard corpora such as Canterbury and Silesia. For the C++ work, I will start with a direct translation of the Python logic, then introduce FSE-style optimizations such as tighter table packing, branchless `deltaNbBits` computations, tuned `tableLog` selection, and, if time allows, multiple interleaved FSE states similar to loop unrolling. For the evaluation, I will refine the benchmarking setup to reduce Python overhead (or move the driver closer to C++) and run all codecs on the full corpus, collecting compression ratio and throughput numbers suitable for tables and plots in the final report. The final analysis will focus on which FSE-specific optimizations actually move the needle, how close the C++ implementation can get to the behavior of zstd's FSE stage, and what trade-offs emerge.

References

1 J. Duda, *Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding*, arXiv, 2013.
https://arxiv.org/abs/1311.2540

2 Y. Collet, "Finite State Entropy – a new breed of entropy coder" and follow-up posts, *Fast Compression Blog.*
https://fastcompression.blogspot.com

3 Y. Collet, **FiniteStateEntropy** library (FSE), GitHub.
https://github.com/Cyan4973/FiniteStateEntropy

4 Y. Collet et al., **Zstandard** compression format and source code, GitHub.
https://github.com/facebook/zstd

5 Stanford Compression Library (SCL), documentation and entropy coder implementations, GitHub.
https://github.com/stanfordcompression/stanford_compression_library