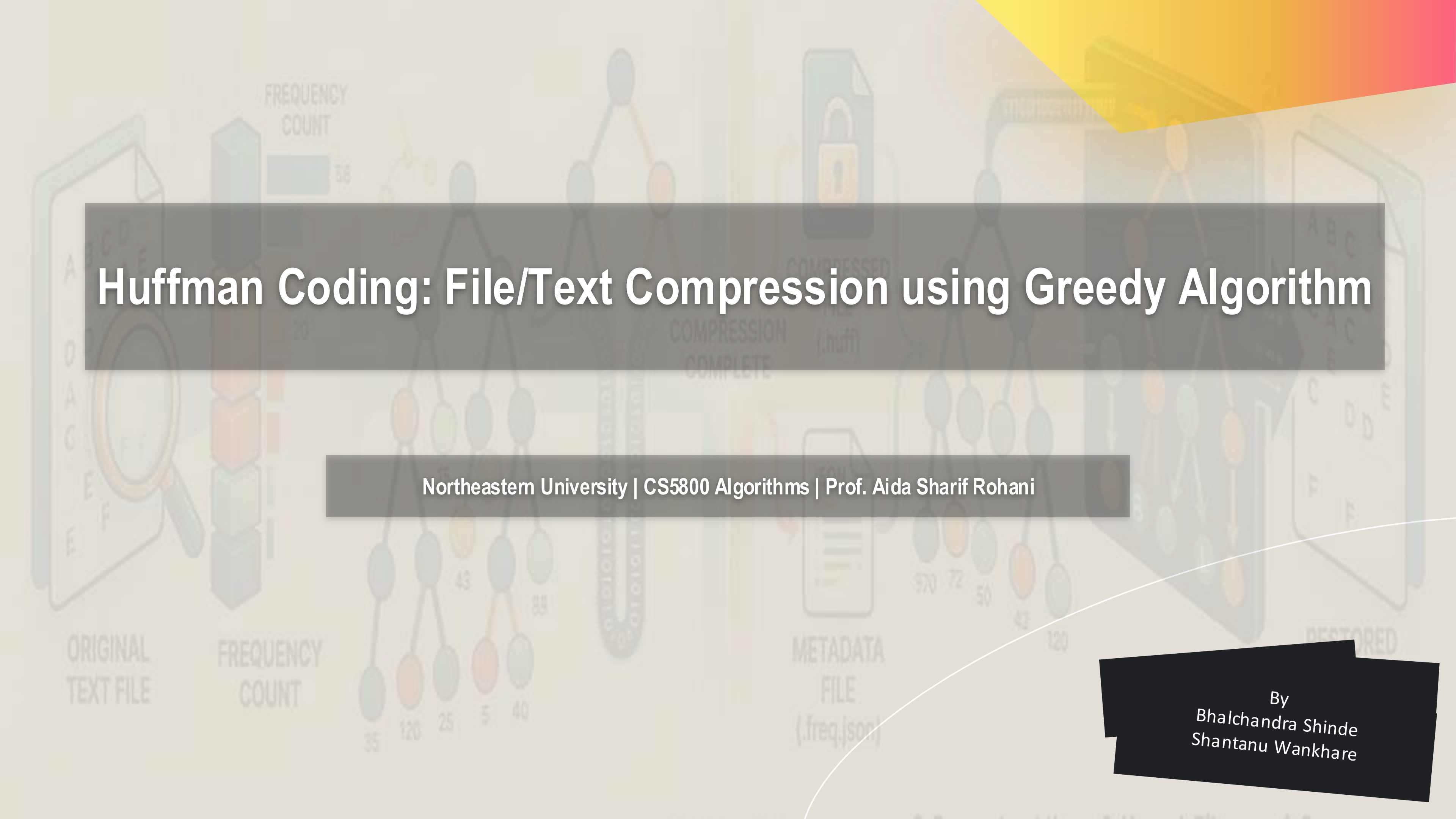


Huffman Coding: File/Text Compression using Greedy Algorithm

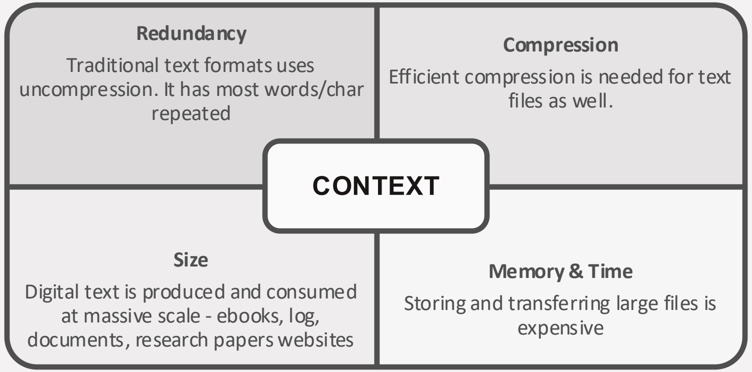
Northeastern University | CS5800 Algorithms | Prof. Aida Sharif Rohani

By
Bhalchandra Shinde
Shantanu Wankhare

Introduction

Key Question.

How can we design a lossless compression system that significantly reduces text file size while enabling fast, real-time decompression for reading?



Redundancy
Traditional text formats uses uncompression. It has most words/char repeated

Compression
Efficient compression is needed for text files as well.

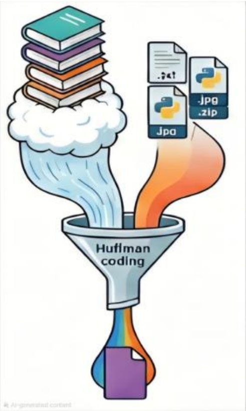
CONTEXT

Size
Digital text is produced and consumed at massive scale - ebooks, log, documents, research papers websites

Memory & Time
Storing and transferring large files is expensive

Huffman tool

Reduce storage size without losing information.



Reader Application

Decode compressed data efficiently enough to display pages of a book on demand.

So far Encodings

File Type	Extensions	Existing Compression / Encoding Method	Is File Already Compressed?	Expected Huffman Compression Ratio	Explanation
Plain Text	.txt, .log, .csv, .json, .xml	None	❌ No	40% – 80% smaller	Text has skewed character frequency; Huffman is ideal.
Documents (Office XML)	.docx, .xlsx, .pptx	ZIP (DEFLATE: LZ77 + Huffman)	✓ Already compressed	File becomes 20–200% larger	Office files are ZIP containers; content already compressed using Huffman + LZ.
PDF	.pdf	Flate/DEFLATE, LZW, JPEG, JP2	✓ Already compressed	+20% to +300% expansion	PDF streams already use entropy coding; random-like distribution.
Images (raw)	.bmp, .ppm, .tiff (uncompressed)	None (sometimes RLE for TIFF)	❌ No	30% – 70% smaller	Raw pixel data compresses fairly well.
Images (compressed)	.jpg, .jpeg	DCT + Quantization + Huffman	✓ Yes	Huge expansion: 300% – 800%	JPEG already uses Huffman coding inside. Compressing again makes it worse.
Images (compressed)	.png	DEFLATE (LZ77 + Huffman)	✓ Yes	Very large expansion: 200% – 600%	PNG uses entropy coding and filters; nearly incompressible.
Audio (raw)	.wav, .pcm, .aiff	None	❌ No	10% – 40% smaller	Raw amplitude distributions slightly skewed; small gains.
Audio (compressed)	.mp3, .aac, .flac	MP3: MDCT + Huffman / AAC: Huffman / FLAC: Rice/Huffman	✓ Yes	Massive expansion: 300% – 1000%	Audio codecs already use Huffman coding internally.
Video (raw)	.yuv	None	❌ No	10% – 30% smaller	Pixel values partly skewed; limited improvement.
Video (compressed)	.mp4, .mov, .mkv, .avi	H.264/HEVC/AV1 (CABAC, CAVLC, entropy coding)	✓ Yes	Very large expansion: 200% – 800%	These codecs use advanced entropy coding more efficient than Huffman.
Python/Source Code	.py, .java, .cpp, .html, .css, .js	None	❌ No	30% – 70% smaller	High redundancy and repeated keywords; good for Huffman.
Binary Executables	.exe, .dll, .bin	Often packed or randomized	⚠ Sometimes	Likely expansion: 20% – 500%	Binaries include many random bytes or pre-packed segments.
Archives	.zip, .7z, .rar, .gz, .whl	DEFLATE, LZMA, PPMD	✓ Fully compressed	Always expands	These formats already use Huffman, arithmetic coding, or LZ — cannot compress again.

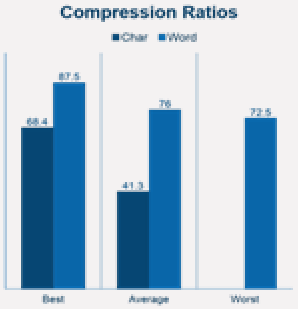
Rationale

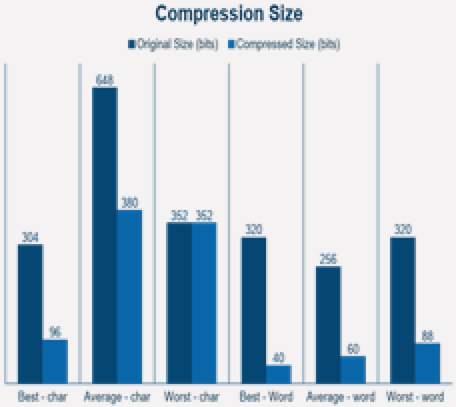
- Huffman Coding is a **classic Greedy Algorithm** that produces an **optimal prefix-free encoding** based on symbol frequencies.
- It is widely used in real systems (ZIP, JPEG, MP3), but classroom examples rarely show practical applications.
- This project extends the Huffman algorithm beyond theory, building a **complete working system**:
  - A compression tool
  - A decompression module
  - A chunk-based paging system
  - A Python Tkinter based book reader

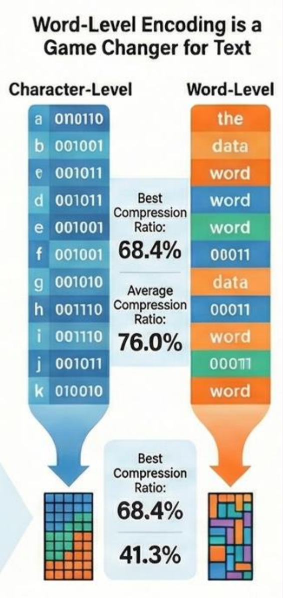
Why this matters:

- Demonstrates how algorithmic theory can be used to build a real application.
- Highlights trade-offs in designing usable compression systems (speed, memory, chunking, file formats).

Analysis - Word level?

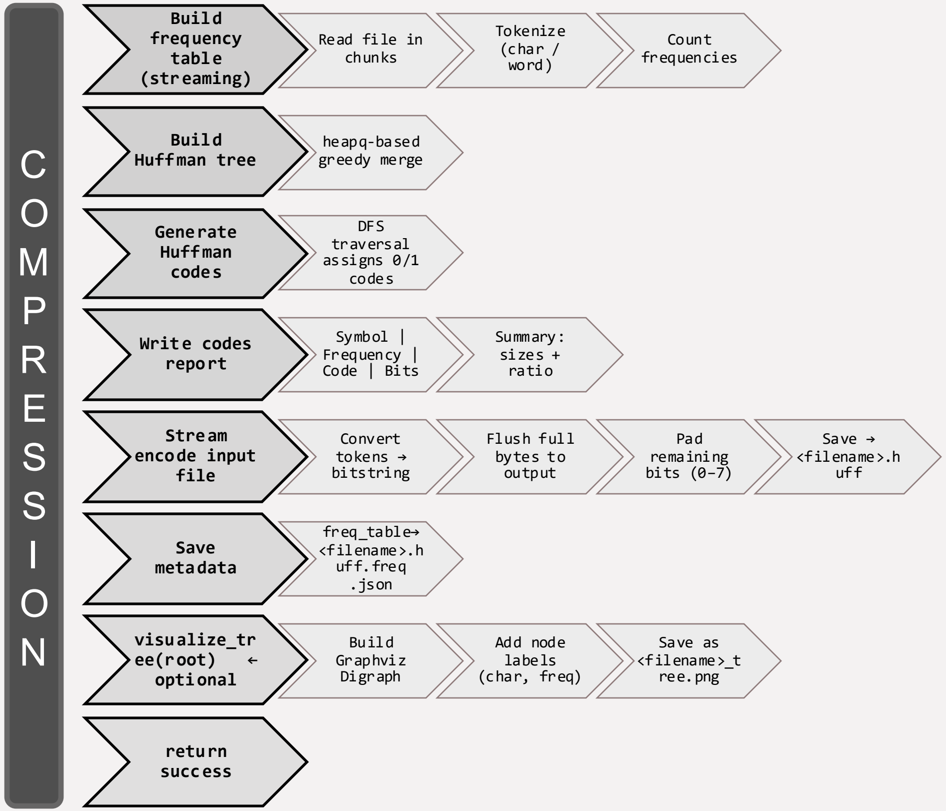






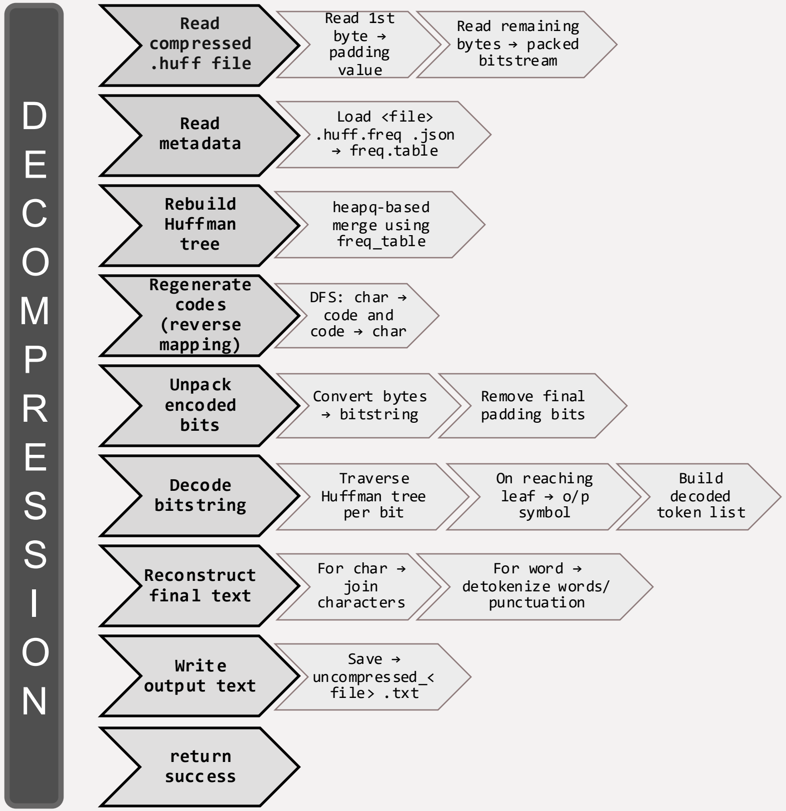
- Character-level Huffman coding compresses but leaves redundancy:
  "the", "and", "to", "of" appear thousands of times.
- Word-level encoding:
  - Dramatically reduces redundancy
  - Produces shorter average codewords
  - Improves readability after decoding
  - Works perfectly for book-style text
- Compression improvement observed:
- Word-level coding achieved up to 70% space savings vs raw .txt.

Pseudocode



COMPRESSION

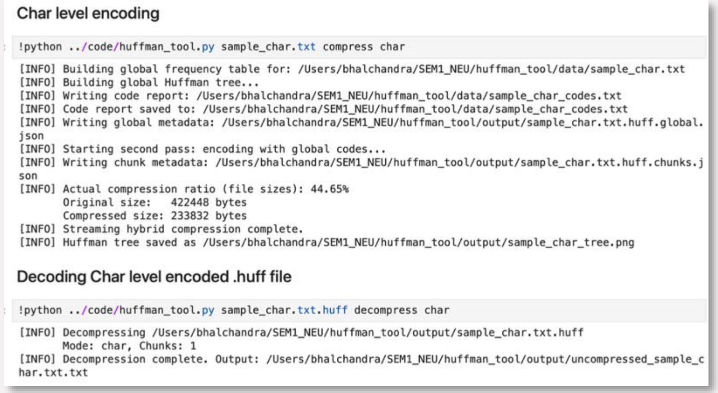
- Build frequency table (streaming) → Read file in chunks → Tokenize (char / word) → Count frequencies
- Build Huffman tree → heapq-based greedy merge
- Generate Huffman codes → DFS traversal assigns 0/1 codes
- Write codes report → Symbol | Frequency | Code | Bits → Summary: sizes + ratio
- Stream encode input file → Convert tokens → bitstring → Flush full bytes to output → Pad remaining bits (0-7) → Save → <filename>.huff
- Save metadata → freq_table→ <filename>.huff.freq .json
- visualize_tree(root) ← optional → Build Graphviz Digraph → Add node labels (char, freq) → Save as <filename>_tree.png
- return success



DECOMPRESSION

- Read compressed .huff file → Read 1st byte → padding value → Read remaining bytes → packed bitstream
- Read metadata → Load <file> .huff.freq .json → freq.table
- Rebuild Huffman tree → heapq-based merge using freq_table
- Regenerate codes (reverse mapping) → DFS: char → code and code → char
- Unpack encoded bits → Convert bytes → bitstring → Remove final padding bits
- Decode bitstring → Traverse Huffman tree per bit → On reaching leaf → o/p symbol → Build decoded token list
- Reconstruct final text → For char → join characters → For word → detokenize words/ punctuation
- Write output text → Save → uncompressed_<file> .txt
- return success

Tool Execution



Char level encoding

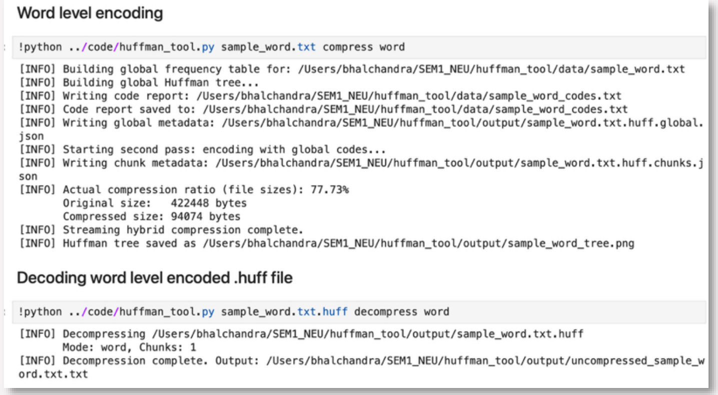
```
!python ../code/huffman_tool.py sample_char.txt compress char
```

```
[INFO] Building global frequency table for: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_char.txt
[INFO] Building global Huffman tree...
[INFO] Writing code report: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_char_codes.txt
[INFO] Code report saved to: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_char_codes.txt
[INFO] Writing global metadata: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_char.txt.huff.global.
json
[INFO] Starting second pass: encoding with global codes...
[INFO] Writing chunk metadata: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_char.txt.huff.chunks.j
son
[INFO] Actual compression ratio (file sizes): 44.65%
       Original size:   422448 bytes
       Compressed size: 233832 bytes
[INFO] Streaming hybrid compression complete.
[INFO] Huffman tree saved as /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_char_tree.png
```

Decoding Char level encoded .huff file

```
!python ../code/huffman_tool.py sample_char.txt.huff decompress char
```

```
[INFO] Decompressing /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_char.txt.huff
       Mode: char, Chunks: 1
[INFO] Decompression complete. Output: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/uncompressed_sample_c
har.txt.txt
```



Word level encoding

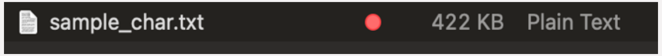
```
!python ../code/huffman_tool.py sample_word.txt compress word
```

```
[INFO] Building global frequency table for: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_word.txt
[INFO] Building global Huffman tree...
[INFO] Writing code report: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_word_codes.txt
[INFO] Code report saved to: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_word_codes.txt
[INFO] Writing global metadata: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_word.txt.huff.global.
json
[INFO] Starting second pass: encoding with global codes...
[INFO] Writing chunk metadata: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_word.txt.huff.chunks.j
son
[INFO] Actual compression ratio (file sizes): 77.73%
       Original size:   422448 bytes
       Compressed size: 94074 bytes
[INFO] Streaming hybrid compression complete.
[INFO] Huffman tree saved as /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_word_tree.png
```

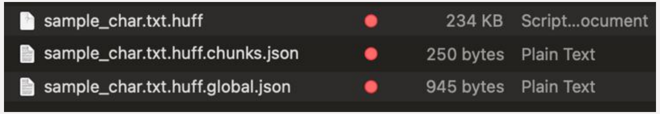
Decoding word level encoded .huff file

```
!python ../code/huffman_tool.py sample_word.txt.huff decompress word
```

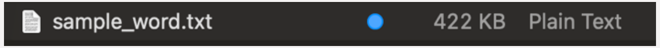
```
[INFO] Decompressing /Users/bhalchandra/SEM1_NEU/huffman_tool/output/sample_word.txt.huff
       Mode: word, Chunks: 1
[INFO] Decompression complete. Output: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/uncompressed_sample_w
ord.txt.txt
```



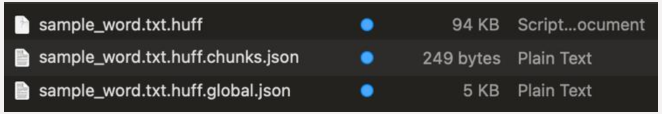
sample_char.txt	●	422 KB	Plain Text



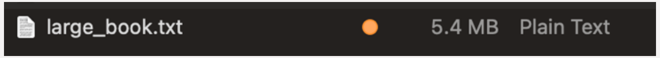
sample_char.txt.huff	●	234 KB	Script...ocument
sample_char.txt.huff.chunks.json	●	250 bytes	Plain Text
sample_char.txt.huff.global.json	●	945 bytes	Plain Text



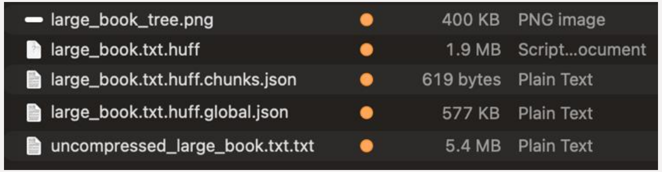
sample_word.txt	●	422 KB	Plain Text



sample_word.txt.huff	●	94 KB	Script...ocument
sample_word.txt.huff.chunks.json	●	249 bytes	Plain Text
sample_word.txt.huff.global.json	●	5 KB	Plain Text

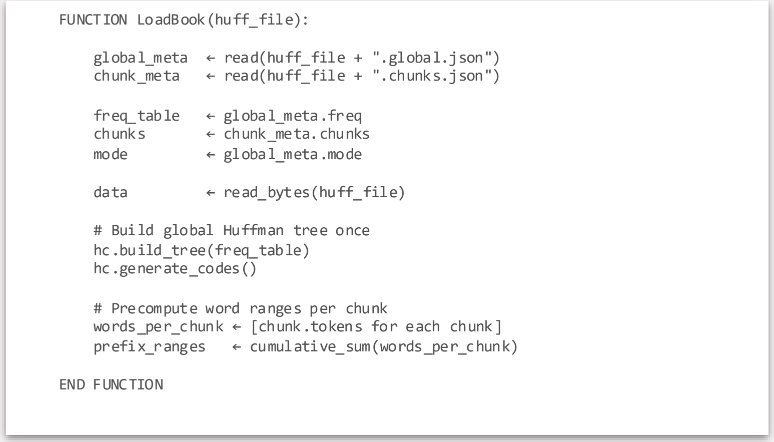


large_book.txt	●	5.4 MB	Plain Text

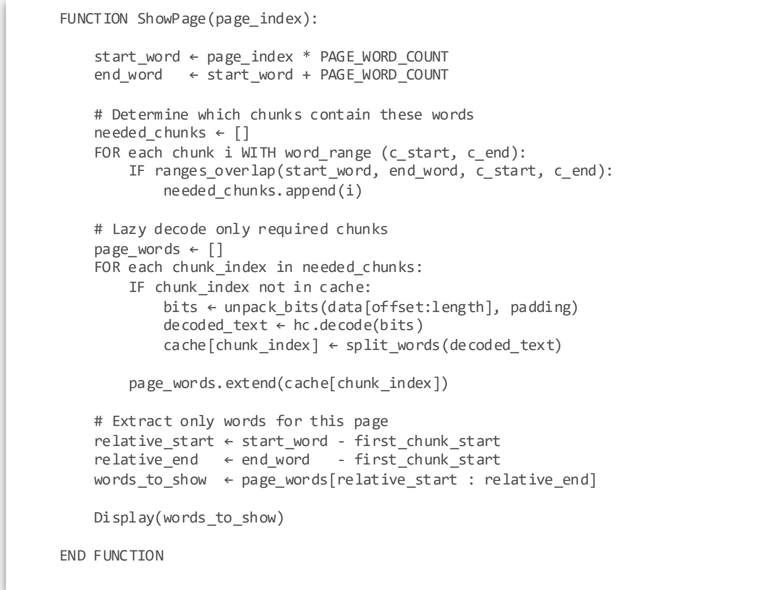


large_book_tree.png	●	400 KB	PNG image
large_book.txt.huff	●	1.9 MB	Script...ocument
large_book.txt.huff.chunks.json	●	619 bytes	Plain Text
large_book.txt.huff.global.json	●	577 KB	Plain Text
uncompressed_large_book.txt.txt	●	5.4 MB	Plain Text

Reader - pseudocode



```
FUNCTION LoadBook(huff_file):

    global_meta  ← read(huff_file + ".global.json")
    chunk_meta   ← read(huff_file + ".chunks.json")

    freq_table   ← global_meta.freq
    chunks       ← chunk_meta.chunks
    mode         ← global_meta.mode

    data         ← read_bytes(huff_file)

    # Build global Huffman tree once
    hc.build_tree(freq_table)
    hc.generate_codes()

    # Precompute word ranges per chunk
    words_per_chunk ← [chunk.tokens for each chunk]
    prefix_ranges   ← cumulative_sum(words_per_chunk)

END FUNCTION
```



```
FUNCTION ShowPage(page_index):

    start_word ← page_index * PAGE_WORD_COUNT
    end_word   ← start_word + PAGE_WORD_COUNT

    # Determine which chunks contain these words
    needed_chunks ← []
    FOR each chunk i WITH word_range (c_start, c_end):
        IF ranges_overlap(start_word, end_word, c_start, c_end):
            needed_chunks.append(i)

    # Lazy decode only required chunks
    page_words ← []
    FOR each chunk_index in needed_chunks:
        IF chunk_index not in cache:
            bits ← unpack_bits(data[offset:length], padding)
            decoded_text ← hc.decode(bits)
            cache[chunk_index] ← split_words(decoded_text)

        page_words.extend(cache[chunk_index])

    # Extract only words for this page
    relative_start ← start_word - first_chunk_start
    relative_end   ← end_word   - first_chunk_start
    words_to_show  ← page_words[relative_start : relative_end]

    Display(words_to_show)

END FUNCTION
```

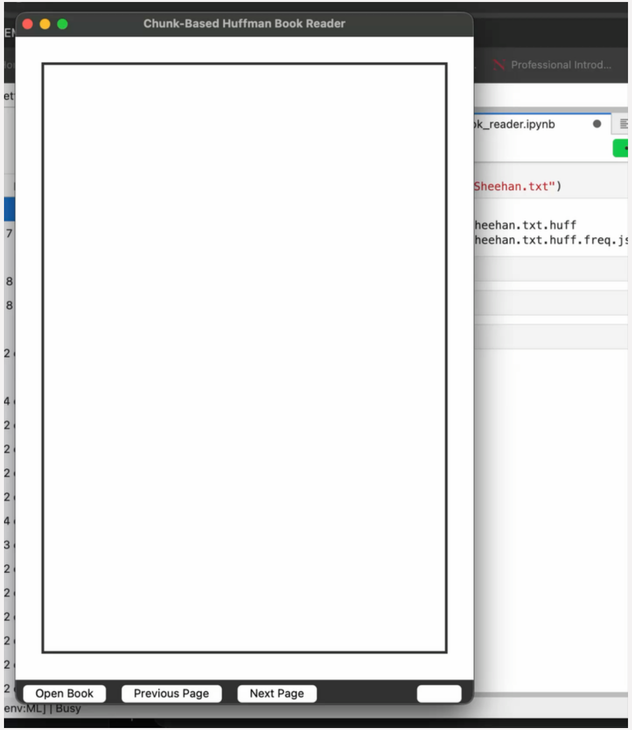
Inputs

- book.txt — raw book
- book.txt.huff — compressed binary
- book.txt.huff.freq.json — chunk metadata
- Chunk sizes, padding values, frequency table

Outputs

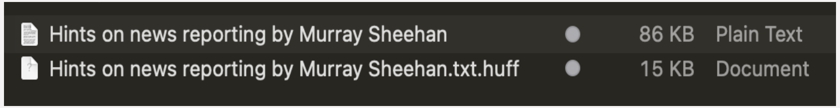
- Decoded individual pages
- Page count
- Compression ratio
- Memory-efficient reading of entire text

Reader application



```
Sheehan.txt")
heehan.txt.huff
heehan.txt.huff.freq.js
```

- Python Tkinter GUI
- Paper like reading experience
- Chunks of 250 words decoded per page
- Page Navigation controls
- Load books with .huff and .json metadata file



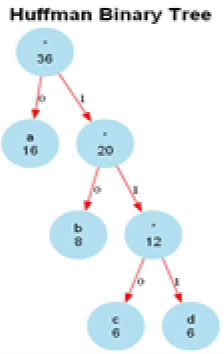
Hints on news reporting by Murray Sheehan	●	86 KB	Plain Text
Hints on news reporting by Murray Sheehan.txt.huff	●	15 KB	Document

Observations

- More repetition → better compression. Larger books shows higher compression ratio
- Huffman coding efficiency depends on frequency distribution
- Worst-case for char: no gain, overhead may slightly increase size. Word level encoding outperforms in any case.
- Visual tree helps understand code assignment
- Reader works in memory, no physical/temporary files created
- Page by Page/chunk wise decompression handles large files efficiently
- Optional files can be skipped to save memory – png tree, codes etc.

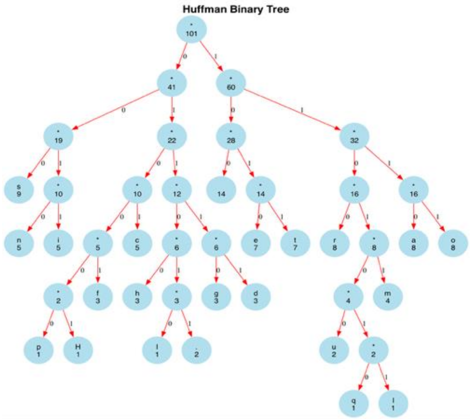
Test Cases - Char

Best case Example:
"aaaaaaaaaaaaaaaaabbbbbbbbccccccdddddd"



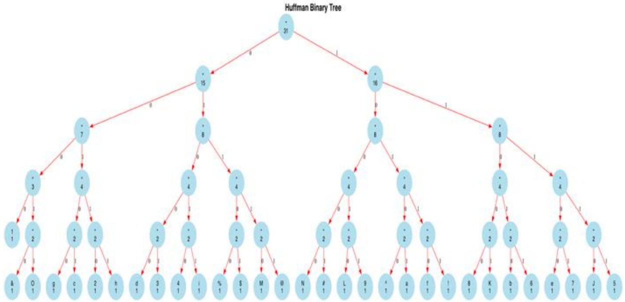
```
Huffman Encoding Report (Character-Level)
Original Text Length: 36 tokens
Symbol Table:
Symbol      | Frequency   | Huffman Code | Bits Used
===================================================
a           | 16          | 0            | 16
b           | 8           | 10           | 16
c           | 6           | 110          | 18
d           | 6           | 111          | 18
===================================================
Original size (bits): 288
Compressed size (bits): 68
Compression ratio: 76.39%
Unique symbols: 4
Most frequent symbol: 'a' (16 times)
Least frequent symbol: 'c' (6 times)
```

Average case Example:
"Huffman coding is a data compression algorithm. It assigns shorter codes to more frequent characters."



```
Huffman Encoding Report (Character-Level)
Original Text Length: 101 tokens
========================================
Original size (bits): 808
Compressed size (bits): 417
Compression ratio: 48.39%
Unique symbols: 22
Most frequent symbol: ' ' (14 times)
Least frequent symbol: 'H' (1 times)
```

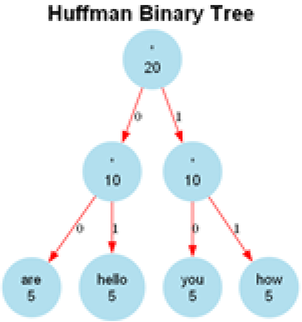
Worst case Example:
"abcdefghiJKLMNO123456789!@#$%^&"



```
Huffman Encoding Report (Character-Level)
Original Text Length: 31 tokens
============================================
Original size (bits): 248
Compressed size (bits): 154
Compression ratio: 37.90%
Unique symbols: 31
Most frequent symbol: 'a' (1 times)
Least frequent symbol: 'a' (1 times)
```

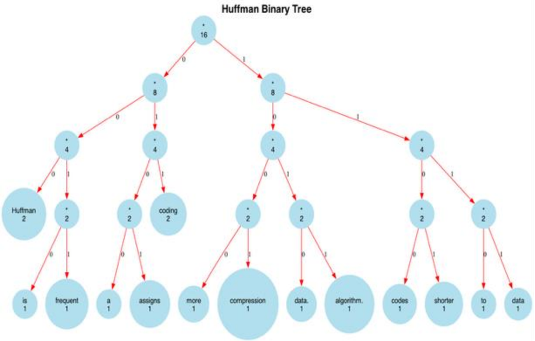
Test Cases - Word

Best case Example:
"hello how are you hello how are you hello how are you hello how are you hello how are you"



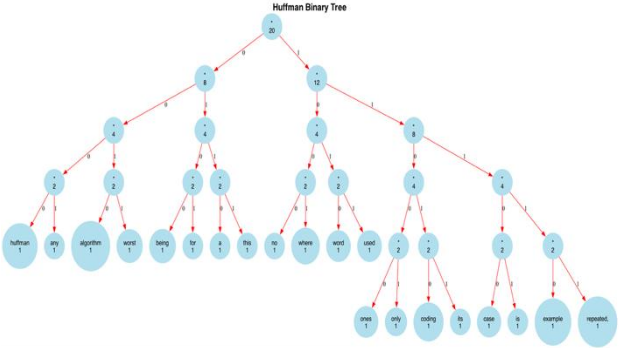
```
Huffman Encoding Report (Word-Level)
Original Text Length: 20 tokens
Symbol Table:
Symbol   | Frequency | Huffman Code | Bits Used
==============================================
are      | 5         | 00           | 10
hello    | 5         | 01           | 10
you      | 5         | 10           | 10
how      | 5         | 11           | 10
==============================================
Original size (bits): 320
Compressed size (bits): 40
Compression ratio: 87.50%
Unique symbols: 4
Most frequent symbol: 'hello' (5 times)
Least frequent symbol: 'hello' (5 times)
```

Best case Example:
"Huffman coding is a data compression algorithm. Huffman coding assigns shorter codes to more frequent data."



```
Huffman Encoding Report (Word-Level)
Original Text Length: 16 tokens
==========================================
Original size (bits): 256
Compressed size (bits): 60
Compression ratio: 76.56%
Unique symbols: 14
Most frequent symbol: 'Huffman' (2 times)
Least frequent symbol: 'is' (1 times)
```

Worst case Example:
"this is a worst case example for huffman coding algorithm where no any word being repeated, its used only ones"



```
Huffman Encoding Report (Word-Level)
Original Text Length: 20 tokens
============================================
Original size (bits): 320
Compressed size (bits): 88
Compression ratio: 72.50%
Unique symbols: 20
Most frequent symbol: 'this' (1 times)
Least frequent symbol: 'this' (1 times)
```

Conclusion

Implemented system

- Reduces storage usage substantially
- Correctly reconstructs text
- Ensures fast page loading
- Works well across multiple books

Huffman Coding is not just theoretical, it powers real systems

This project demonstrates complete integration from **algorithm → compression → metadata → decoding → UI**.

Shows how algorithm design, data structures and UX can combine to produce a working, practical application.

`GIT link - https://github.com/bshind87/Huffman_tool.git`

THANK YOU