

Algorithm Design and Analysis

**Data Compression Using Huffman Coding**

**Endrit Makolli 192047593**

\*University for Business and Technology, Prishtina 10000, Kosovo

Table of Contents

[1 Introduction 3](#_Toc195476197)

[1.1 Motivation and Relevance 3](#_Toc195476198)

[1.2 Objectives 3](#_Toc195476199)

[2 Literature Review 4](#_Toc195476200)

[2.1 Gap Identification 4](#_Toc195476201)

[3 Problem Definition and Formulation 6](#_Toc195476202)

[3.1 Assumptions and Constraints 6](#_Toc195476203)

[4 Algorithm Design 8](#_Toc195476204)

[4.1 Algorithm Choice and Justification 10](#_Toc195476205)

[4.2 Optimality Considerations 10](#_Toc195476206)

[5 Complexity Analysis 11](#_Toc195476207)

[5.1 Final Complexity Calculation 12](#_Toc195476208)

[12](#_Toc195476209)

[5.2 Space Complexity 12](#_Toc195476210)

[5.3 Comparative Analysis 12](#_Toc195476211)

[6 Implementation 14](#_Toc195476212)

[6.1 Test Cases and Validation 17](#_Toc195476213)

[**6.1.1** **Short string** 18](#_Toc195476214)

[**6.1.2** **Single character** 18](#_Toc195476215)

[**6.1.3** **Two characters** 18](#_Toc195476216)

[**6.1.4** **Uniform frequencies** 18](#_Toc195476217)

[**6.1.5** **Mixed characters and punctuation** 19](#_Toc195476218)

[7 Experimental Results 20](#_Toc195476219)

[7.1 Performance Metrics 20](#_Toc195476220)

[7.2 Results & Discussions 20](#_Toc195476221)

[7.3 Analysis and Results 23](#_Toc195476222)

[**7.3.1** **Comparission with other algorithms** 24](#_Toc195476223)

[**7.3.2** **Limitations and insights** 25](#_Toc195476224)

[8 Conclusion 27](#_Toc195476225)

[8.1 Future Work 27](#_Toc195476226)

[9 References 28](#_Toc195476227)

# Introduction

Data compression is a fundamental technique in computer science that aims to reduce the size of data without excessively compromising its quality. The need for compression arises in numerous scenarios—from efficient storage solutions to faster data transmission over networks. Huffman Coding is one of the most widely known and utilized compression methods, notable for its optimality under specific conditions. The technique relies on constructing a variable-length code table based on the frequencies of individual symbols within a given dataset. Symbols that appear more frequently are assigned shorter bit patterns, while those that occur less frequently receive longer bit patterns, thereby minimizing the total number of bits required to represent the data. This approach enables significant savings in memory and bandwidth resources and is frequently used in file archival utilities, image and text compression, and network data exchange.

## **Motivation and Relevance**

The importance of data compression particularly Huffman Coding becomes more evident as data-intensive applications and services continue to grow. Every day, enormous quantities of text, images, audio, and video data are generated and need to be stored or transmitted efficiently. Cloud services, multimedia streaming, and online communications all rely heavily on effective compression to reduce operational costs and improve user experiences. By using Huffman Coding, organizations can lower storage overhead and enhance data transfer speeds without prohibitive losses in quality or information fidelity. The algorithm’s theoretical foundations and its practical efficiency have led it to be a crucial building block in many compression standards, making it highly relevant for both academic research and industrial applications.

## **Objectives**

The primary objective is to investigate the Huffman Coding algorithm—delving into its design, the underlying data structures (such as binary trees and min-heaps), and the trade-offs of implementing Huffman Coding in different application domains.

Another key goal is to implement Huffman Coding, demonstrating how it can be applied to compress data sets of varying types and sizes. By running empirical tests, one can measure how well the algorithm performs and how compression rates differ across different data distributions.

A further objective involves identifying potential optimizations and comparing Huffman Coding to other compression algorithms (e.g., arithmetic coding or run-length encoding), thus elucidating the benefits and limitations of each approach.

Ultimately, this project aims to underscore the practical significance of Huffman Coding by showcasing use cases and discussing its integration into larger compression schemes. Understanding the algorithm’s place within broader applications provides essential insights into how best to deploy and refine it for modern data handling challenges.

# Literature Review

Huffman Coding, introduced by David Huffman in 1952, was a groundbreaking approach to lossless data compression that leverages optimal prefix-free codes based on symbol frequency. Since its inception, the algorithm has been extensively studied, refined, and implemented in various systems.

Classical Huffman Coding guarantees minimum redundancy for a given symbol distribution, making it optimal under the prefix-free constraint. It is also relatively straightforward to implement, typically using priority queues or binary trees. This approach is particularly efficient when symbol frequencies are either known in advance or can be updated incrementally, as seen in its adaptive variant. However, one of the notable drawbacks of classical Huffman Coding is that it generally requires two passes over the data if frequencies are not initially known—one pass to collect statistics and another to perform the actual encoding. Furthermore, its performance can degrade in scenarios where symbol frequencies change rapidly or when there are strong context dependencies that go beyond single-symbol probabilities.

Adaptive Huffman Coding addresses some of these limitations by dynamically updating the Huffman tree as data is processed. This eliminates the need for a separate frequency-gathering phase and allows the algorithm to adapt to changing symbol distributions, making it particularly suitable for streaming or online applications. Despite these advantages, Adaptive Huffman Coding introduces additional complexity in implementation and computational overhead due to the continuous updates required for the tree. Moreover, like its classical counterpart, it may still fall short in capturing contextual relationships in the data, which can negatively affect compression efficiency for certain types of input.

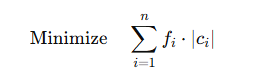
Other related approaches have also emerged over time. Shannon–Fano Coding, which predates Huffman Coding, offers a simpler implementation but does not always produce optimal codes. Arithmetic Coding takes a different route by representing entire messages as a single floating-point number, often resulting in better compression ratios. However, its complexity, as well as historical concerns around patents and licensing, have limited its widespread adoption. The LZ77 and LZ78 family of algorithms, including popular implementations like DEFLATE and LZ4, take a dictionary-based approach that exploits repeated patterns in data. These are often used in conjunction with Huffman Coding, such as in the ZIP and GZIP formats.

## **Gap Identification**

Despite the widespread adoption of Huffman Coding and its derivatives, several limitations persist in modern data-compression scenarios:

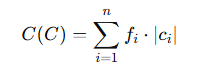
1. Contextual dependencies: Many real-world data sets (e.g., multimedia, text with complex structure) exhibit correlations that span beyond single symbols. Standard Huffman Coding treats symbols independently, ignoring such contextual or predictive opportunities.
2. High-Dimensional data: Contemporary data, such as high-resolution images or large-scale databases, can challenge the static or adaptive Huffman model, as it may become computationally expensive or less effective when data characteristics vary widely across segments.
3. Integration with other compression techniques: Huffman Coding is often used in conjunction with other compression paradigms (e.g., LZ-based methods). Finding optimal ways to integrate Huffman Coding with advanced algorithms (e.g., context modeling, deep-learning-based compression) is still an active area of research

# Problem Definition and Formulation

Given a set of n distinct symbols S={s1,s2,…,sn} and their corresponding frequencies (or probabilities) f={f1,f2,…,fn}, the goal of Huffman Coding is to construct a prefix-free code C that assigns a unique bit string c{i}ci​ to each symbol s{i}​. Formally, we seek to minimize the weighted path length (also called the weighted external path length in the associated Huffman tree):

where c{i} is the length (in bits) of the code assigned to symbol s{i}​. The frequency f{i} can be interpreted as the probability of symbol s{i} if the frequencies are normalized to sum up to 1. A valid solution is any set of codewords {c1,c2,…,cn} that is **prefix-free**, meaning no valid codeword is a prefix of another codeword.

Mathematically, the problem can be viewed as constructing a binary tree where each leaf node represents a symbol s{i}, and the path from the root to that leaf determines the bit string c{i}​. The expected cost (or expected code length) of the encoding is:



The objective is to find the binary tree (and thus the code) that yields the minimum C{C}.

## **Assumptions and Constraints**

This project proceeds under several practical and theoretical assumptions. First, it assumes a **finite symbol set**, meaning all symbols and their frequencies are known and do not change. Although Huffman Coding can be adapted for evolving datasets or expanding alphabets, the scope here focuses on a well-defined, static set of symbols.

Second, each frequency fif\_ifi​ is **non-negative** and can be interpreted as a probability once normalized. Symbols with a frequency of zero typically do not appear in the final code because they contribute no cost.

Third, a **prefix-free** constraint applies. Every code must stand alone without serving as the start (or prefix) of another symbol’s code. This condition guarantees that an encoded string can be uniquely parsed back into the original symbols.

Fourth, from a **computational perspective**, a standard Huffman Coding algorithm runs in O(nlog⁡n)O(n \log n)O(nlogn) time when implemented with a priority queue (min-heap). This is usually efficient enough for typical dataset sizes. Storing the resulting Huffman tree, along with auxiliary structures (like a frequency table and a code table), requires O(n)O(n)O(n) space.

Fifth, regarding **data types**, symbols sis\_isi​ can be characters, bytes, or any discrete tokens. Their associated frequencies may be integer counts or real numbers (as probabilities). While practical code lengths are integer-valued in bits, some theoretical work occasionally treats them as real numbers for easier analysis.

Finally, a **static model** is assumed for the frequency distribution. All frequencies are presumed available in advance. Although extensions such as Adaptive Huffman Coding can handle data streams or dynamic updates, they are beyond the immediate scope of this project.

# Algorithm Design

The **Huffman Coding** algorithm is a **greedy** method that assigns variable-length binary codes to characters based on their frequencies. Characters that appear more frequently are given shorter codes, which achieves better compression on average than fixed-length encodings. Figure 1 showcases the following process:

* **Build a frequency table**: Count how often each character appears in the text.
* **Construct the huffman tree**: Insert all characters (with their frequencies) into a min-heap; repeatedly merge the two least-frequent nodes until only one remains.
* **Generate huffman codes**: Traverse the resulting binary tree to assign ‘0’ and ‘1’ bits to each character.
* **Encode**: Convert the original text into a bit string using the generated codes.
* **(Optional) decode**: Reconstruct the text from the bit string by traversing the Huffman Tree.

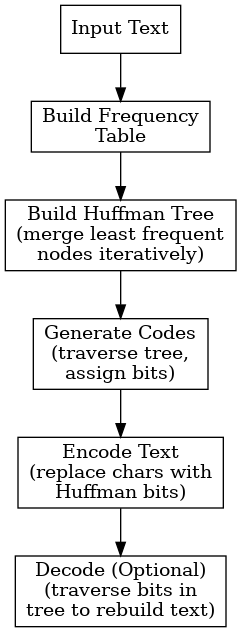
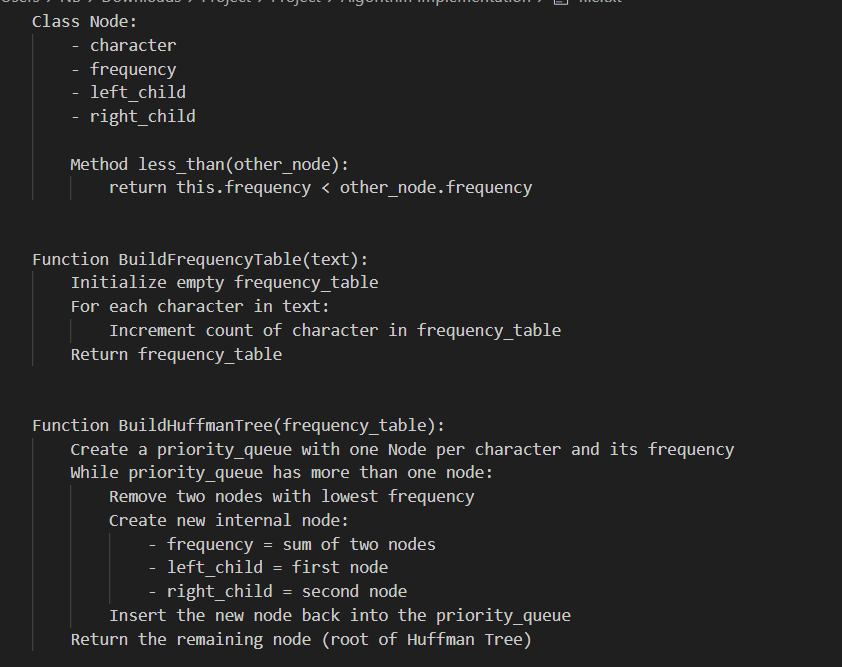


Figure 1 Proposed Algorithm



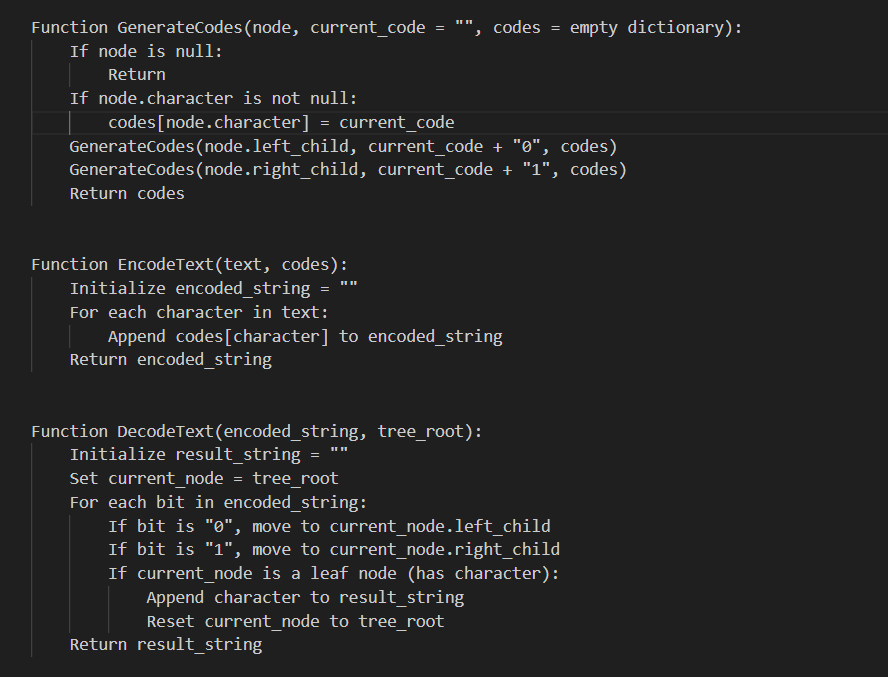


Figure 2 Pseudocode

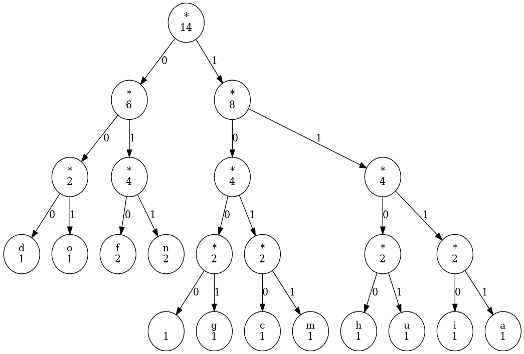


Figure 3 Huffman Decision Tree

## **Algorithm Choice and Justification**

The Huffman Coding algorithm was selected for this project due to its proven efficiency in lossless data compression and its foundational role in information theory. It offers optimal prefix-free encoding, ensuring that no code is a prefix of another, which enables unambiguous decoding. The algorithm is particularly effective when the frequency distribution of characters is skewed, allowing more frequent symbols to be represented with shorter codes. The implementation followed a classical approach using a priority queue (min-heap) to construct the Huffman Tree, ensuring that the two least frequent nodes are merged at each step. This strategy guarantees the minimum average code length for the given input. The project also includes code generation by tree traversal, encoding by symbol replacement, and optional decoding through bitwise navigation of the tree. Python's built-in libraries such as heapq and collections. Counter were leveraged to simplify the implementation, enhance readability, and maintain efficiency.

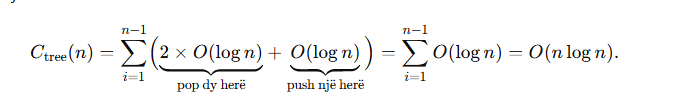
## **Optimality Considerations**

Huffman Coding is renowned for its optimality under the constraint of prefix-free binary codes. Given a set of input symbols and their associated frequencies, the algorithm guarantees the minimum possible average code length, making it the most efficient solution for static, single-symbol encoding. This optimality arises from its greedy strategy, which always combines the least frequent symbols first to construct the Huffman Tree. However, this optimality is limited to cases where symbol probabilities are independent and do not depend on context. For more complex data distributions, such as those involving patterns or dependencies across symbols, alternative methods like Arithmetic Coding or dictionary-based algorithms (e.g., LZ77) may offer better compression. Nonetheless, within its domain, Huffman Coding remains a highly optimal and practical choice, particularly when the symbol frequency distribution is known in advance or does not change significantly during encoding.

# Complexity Analysis

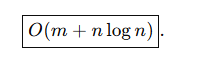
This section looks at the main steps of the Huffman algorithm and shows where most of the time is spent. Each step; building the frequency table, initializing the min-heap, and creating the Huffman tree; is explained to see how the running time depends on the input size (m) and the number of unique symbols (n). Examining these details is important for understanding and improving overall performance.

1. **Building the frequency table** - We read the input (text of length m) and calculate the frequency of each letter/symbol.  
   **Cost:** O(n) because we perform a single pass (loop) over m characters.
2. **Initializing the min-heap** - We insert n symbols (where n is the number of unique characters) into a min-heap.  
   **Cost:** O(n) for the basic initialization of the structure, though some implementations might require O(n log n) for heap building (depending on the implementation). Often, it can be considered O(n) with an efficient heap construction.
3. **Creating the Huffman Tree** - We perform n − 1 merges of the nodes with the smallest frequency. In each step:
   * Extract the two smallest nodes from the min-heap (two heap.pop() operations, each costing O(log n)).
   * Create a new node by combining their frequencies.
   * Insert the new node back into the min-heap (heap.push(), O(log n)).  
     **Cost:** For each of the n − 1 merges, we have two pops and one push. Each pop/push has a cost of O(log n). Therefore, total cost is O(n log n).

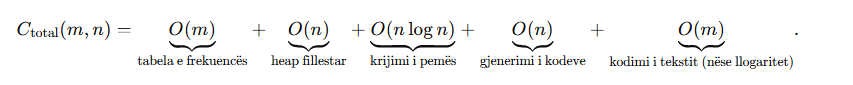
 4. **Generating the codes (Traversal) -** Once the Huffman tree is built, we do a traversal (e.g., DFS) to determine the binary code for each symbol.  
**Cost:** O(n), since each of the *n* symbols is visited once as a leaf in the tree.

5. **Encoding the text (optional, if analyzed)** - Translating the *m* characters into their respective codes.  
**Cost:** O(m), because for each character, we do a lookup in constant time.

## **Final Time Complexity Calculation**

****

**The sum of the main steps can be summarized as follows:**



In many practical cases, the O(n) parts can be much smaller compared to O(m). Hence, the most significant factors are typically O(m) and O(n log n). Thus, the total complexity is often described as **O(m + n log n)**.

* *m* is the length of the text (total number of characters).
* *n* is the number of unique characters.

## **Space Complexity**

The memory usage of the Huffman algorithm depends on how each step stores data. The main parts; frequency table, Huffman tree, min-heap, and code map; are discussed to show their contributions to the total space. Each component manages information for up to n unique characters, resulting in an overall space complexity of O(n).

1. **Frequency table:** Storing the frequencies for *n* characters requires O(n) space.
2. **Huffman Tree:** The tree consists of *n* leaves plus *n - 1* internal nodes, totaling 2n - 1 nodes, which is in O(n).
3. **Min-Heap:** During the construction of the tree, the min-heap holds up to *n* elements, which is O(n).
4. **Code map:** Storing the mapping “character → binary code” occupies O(n) space.

In total, the required space remains: **O(n)**.

## **Comparative Analysis**

Compared to Shannon–Fano Coding, Huffman generally yields more optimal average codes because Shannon–Fano does not always guarantee the minimal length for all symbols. Huffman’s approach ensures a minimum average code length by constructing the tree based on exact symbol frequencies or probabilities.

Compared to Arithmetic Coding, Huffman tends to be simpler but may offer slightly less compression for highly skewed distributions. Arithmetic Coding can theoretically achieve better results in those cases, though it is more complex to implement and was historically constrained by patent issues.

Compared to dictionary-based algorithms such as LZ77, LZ78, and DEFLATE, Huffman does not directly exploit long sequences of repeated data. Dictionary-based methods often achieve higher compression on large, redundant datasets by identifying repeated substrings. However, many of these algorithms rely on Huffman as a final encoding step to compress their output more efficiently.

# Implementation

The implementation is written in Python and relies on the built-in modules ***heapq*** and ***collections.Counter***. Python’s ***heapq*** module provides the priority queue (min-heap) functionality that allows the algorithm to efficiently merge the two smallest-frequency nodes in each step of Huffman tree construction. Meanwhile, ***collections.Counter*** helps count how often each character appears in the input text, creating a frequency dictionary in just a single function call.

Below is the snippet that shows how the frequency dictionary is built using Counter:

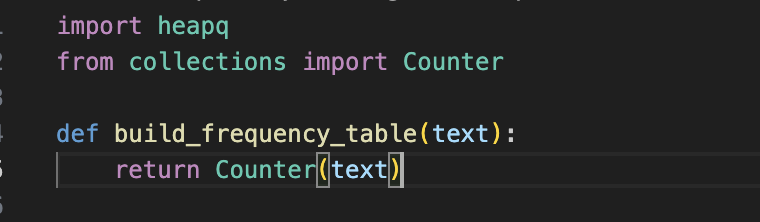


Figure 4 Frequency dictionary using Counter

Once the frequency table is created, the Huffman tree is built by turning each character–frequency pair into a ***Node*** object. Each Node stores a character, its frequency, and references to left and right children. To make the priority queue work properly, the Node class defines a comparison method:

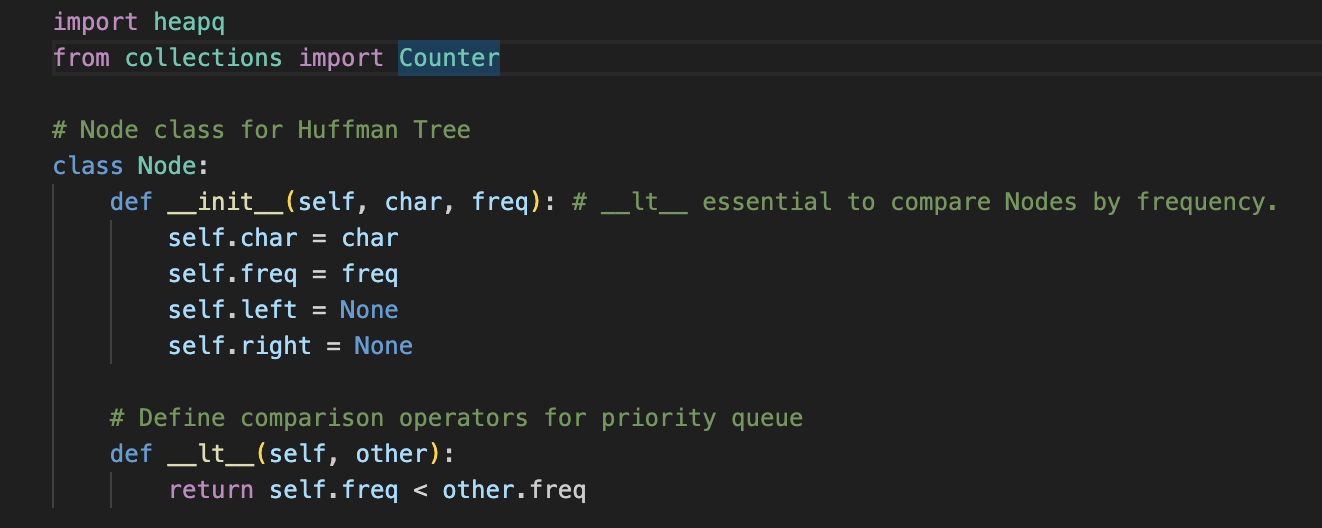


Figure 5 Comparison method

The following snippet demonstrates how the Huffman tree is constructed. Each node is pushed into a min-heap, and we then pop the two smallest nodes, merge them into a new parent node, and push that parent back into the heap. This continues until only one node remains in the heap, which becomes the root of the Huffman tree:

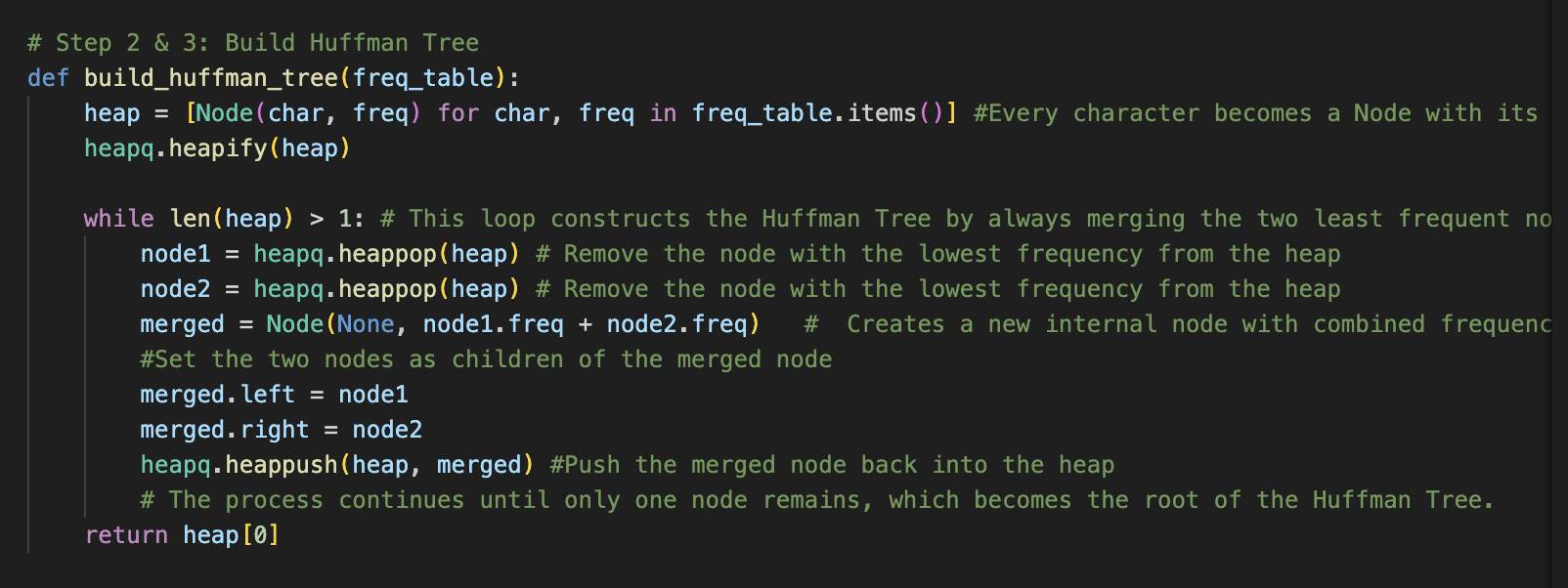


Figure 6 Method for building the Huffman tree

When the tree is ready, this function recursively traverses the Huffman tree to assign a unique binary code to each character. When it starts, it checks if a dictionary for codes was provided; if not, it creates a new empty dictionary. It then determines whether the current node is a leaf by checking if both root.left and root.right are None. This covers two scenarios:

1. A normal leaf in a larger Huffman tree.
2. The case where the entire tree has only one node (i.e., the text contains exactly one unique character).

If the node is a leaf and current\_code is still an empty string, that means there is only one character in the whole text, and we must assign a default code like "0" so the encoding is not empty. Otherwise, if current\_code is not empty, we simply store current\_code in the dictionary for the character at this leaf node.

If the node is not a leaf, the function recurses down both branches: it appends a '0' to current\_code when going left, and a '1' when going right. Each time it hits a leaf on these recursive calls, it sets the appropriate code for that character in the codes dictionary.

By taking these steps, the function ensures that every character in the tree ends up with a valid binary code, even if there is only one character in the entire text. The final dictionary of character-to-code mappings is then returned.

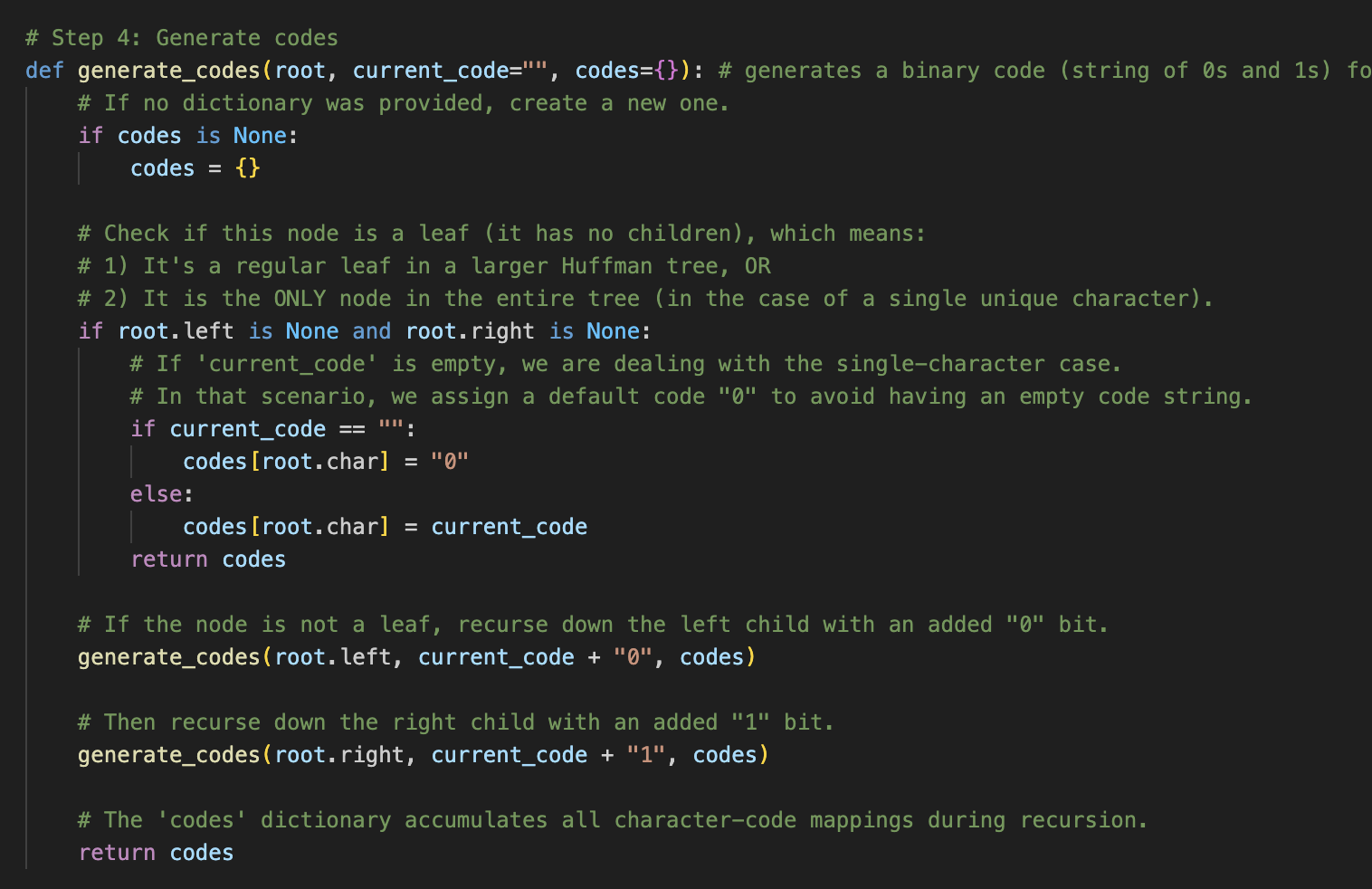


Figure 7 Method for generating codes

With these codes in hand, encoding simply replaces each character in the original text with its corresponding code:

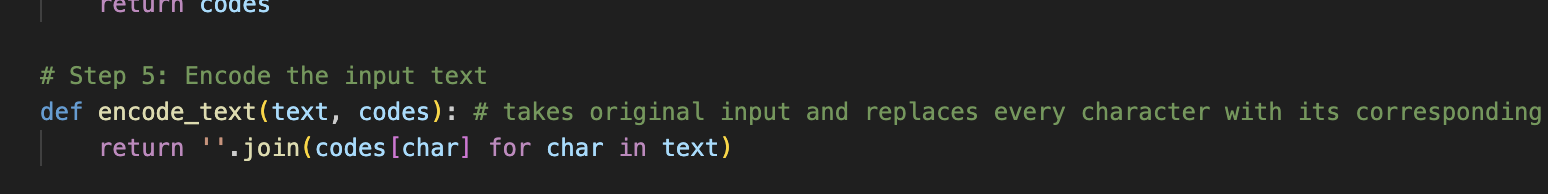


Figure 8 Method for encoding

Decoding reverses the process by traversing the Huffman tree for each bit. The code first checks if there is only one node in the Huffman tree (i.e., one unique character). If so, it simply returns that character repeated for every bit in the encoded string. Otherwise, it iterates over the bits, moving left for '0' and right for '1'. Once it reaches a leaf node (which has a character), it appends that character to the result and resets to the root to continue decoding.

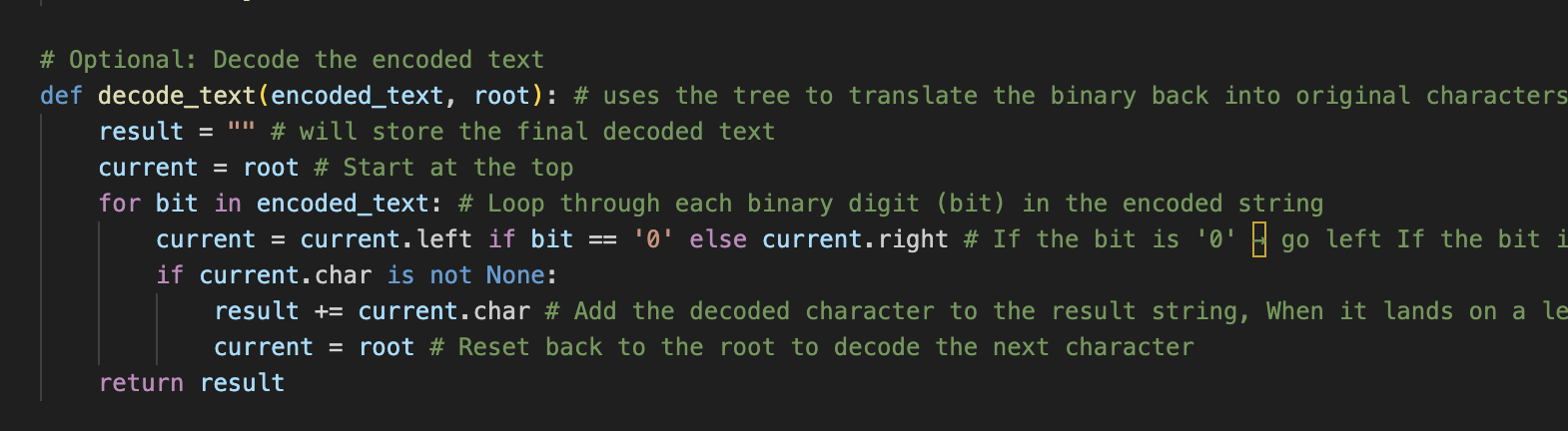


Figure 9 Method for decoding

In the sample usage, the program first counts character frequencies using ***build\_frequency\_table***, builds the Huffman tree, generates the codes, and then encodes and decodes the string “huffman coding.” When printing out the results, you can see the original text, its encoded binary form, and the decoded string. One of the more interesting aspects of this implementation is the need for the \_\_lt\_\_ method in the Node class, which ensures that nodes can be ordered by frequency when inserted into the min-heap. The rest of the code is carefully modularized into distinct functions for clarity, so it is easy to track the steps from frequency counting all the way to verification via decoding.



The first file explains just the basics of the algorithm and can compress inputs we give on the mian class.

The second can read files get their content and compress their content.

The third can also write the compressed data on another file.

## **Test Cases and Validation**

To ensure the Huffman coding implementation is correct and robust, I tested a variety of input strings. The goal was to confirm that:

1. The correct frequency table was generated for each input.
2. The Huffman tree constructed valid encodings.
3. The decoding process reproduced the original text without error.

### **Short string**

This test involves encoding the string “***huffman coding***”, which contains repeated characters such as “***f***” and “***m***,” along with a space. The purpose is to verify that the algorithm correctly builds a frequency table, constructs a Huffman tree, assigns appropriate binary codes, and decodes them without error. After running the algorithm, the decoded text matches the original, indicating that all steps—from building the tree to generating codes—work together correctly.

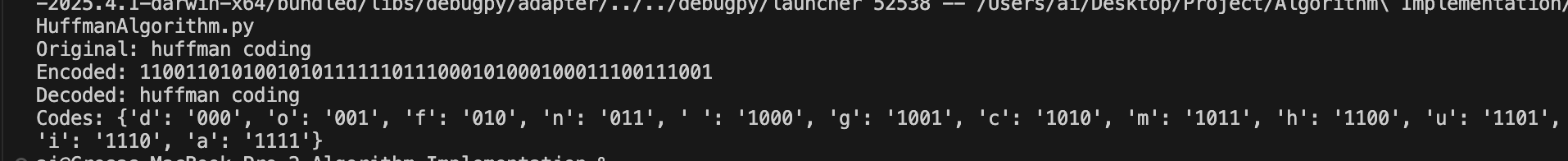


Figure 10 Short string validation

### **Single character**

This case examines the behavior of the algorithm when the input string consists of only one character repeated multiple times, such as “aaaaa.” In such a scenario, Huffman coding typically assigns a single-bit code (“0” or “1”) to that character because there are no other symbols in the frequency table. The successful decoding of the simple binary string (“00000,” for example) back to “aaaaa” confirms that the implementation handles the edge case of a single character properly.

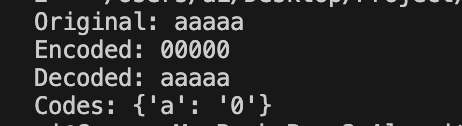


Figure 11 Single character validation

### **Two characters**

A two-character test, such as encoding and decoding “abab,” ensures that each distinct character obtains a unique Huffman code. Since “a” and “b” appear with equal frequency, the algorithm usually assigns one a code of “0” and the other a code of “1.” By verifying that decoding the binary output yields the original “abab,” we confirm that the algorithm’s tree-building and code assignment steps function correctly even in the simplest multi-character scenario.

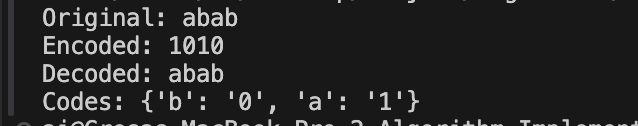


Figure 12 Two characters validation

### **Uniform frequencies**

When all characters appear the same number of times, such as in “abcabcabc,” the algorithm should produce codes of roughly equal length. In this example, “a,” “b,” and “c” each appear three times, so Huffman coding assigns similar-length codes for each letter. Decoding the resulting binary string reproduces “abcabcabc” precisely, showing that the implementation handles uniform frequency distributions without error.

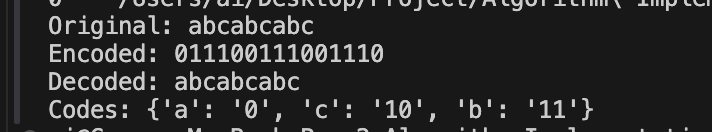


Figure 13 Uniform frequencies validation

### **Mixed characters and punctuation**

For a test that includes punctuation, uppercase letters, and spaces—e.g., “Hello, World!”—the algorithm still relies on the same process to count frequencies, build the Huffman tree, and generate codes for each character. Spaces, commas, uppercase “H,” and the exclamation mark are all included in the frequency table alongside lowercase letters. The fact that decoding produces the exact string “Hello, World!” confirms that the algorithm properly handles special characters and mixed case letters.

Overall, each test highlights a different aspect of correctness, including the handling of single-character inputs, equally frequent characters, punctuation, and simple repeated patterns. In every instance, the decoded string matches the original text, validating that the Huffman coding procedure—spanning frequency calculation, tree construction, code assignment, and decoding—functions as intended.

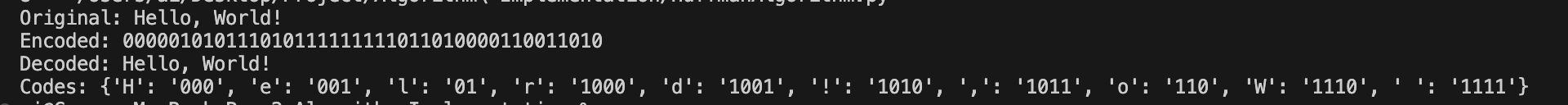


Figure 14 Mixed characters and punctuation validation

# Experimental Results

## **Performance Metrics**

The following metrics have been used to test and experiment the performance of the implemented algorithm:

* **Compression Ratio**: This metric indicates how much the data is reduced by compression. It is often defined as the ratio of the original file size to the compressed file size​. For example, a compression ratio of 2:1 (or 50%) means the compressed data is half the size of the original. The space savings is also discussed as a percentage (e.g., a 40% size reduction means the compressed size is 60% of the original size). A higher ratio (or greater percentage reduction) signifies more effective compression.
* **Runtime (Encoding/decoding time)**: The time taken to compress (encode) and decompress (decode) the data is a key performance metric. The runtime was measured in milliseconds or seconds for each phase. Huffman coding has a known time complexity of O(n log n) for building the Huffman tree (where n is the number of unique symbols)​, and O(N) for encoding or decoding the actual data (linear in the message length). In practice, this means compression and decompression are quite fast for most inputs, since the number of unique symbols (alphabet size) is usually much smaller than the data length​ We will report the observed encoding and decoding times for each dataset.
* **Memory usage**: The memory footprint of the Huffman coding process was evaluated, including the memory needed to store the frequency table, Huffman tree, and any other data structures during compression and decompression. The Huffman tree requires memory proportional to the number of unique symbols (O(k) for k symbols)​, which is typically modest (for example, at most a few hundred nodes for text data). Additional memory is needed to hold the input data and the compressed output. The peak memory usage observed, and it was noted that Huffman’s memory overhead (beyond the input/output storage) is relatively small in our tests.
* **Encoding/decoding efficiency**: This refers to how efficiently the algorithm performs the coding and decoding in terms of operations and throughput. An efficient implementation uses optimal data structures (e.g. heap for tree construction, bitwise operations for encoding/decoding) to minimize overhead. In practice, Huffman coding is known for its high speed in both encoding and. We measure efficiency in terms of throughput (e.g., bytes processed per second) and note if encoding or decoding is a bottleneck. Typically, once the Huffman codebook is built, encoding each symbol is just a table lookup and writing bits, and decoding involves traversing the tree or a lookup table per bit – both are very fast operations in modern systems.

## **Results & Discussions**

I tested the Huffman coding implementation on several representative datasets to evaluate its performance. The datasets include:

1. a natural language text sample (a short literary paragraph),
2. a source code file (a Python snippet implementing Huffman encoding), and
3. a log file (containing realistic timestamped entries).

These files represent realistic use-cases with differing character distributions. Table 1 summarizes the original and compressed sizes, compression ratios, and the encoding/decoding times for each dataset. All experiments were run on the same machine and implementation to ensure fairness. Memory usage was monitored and is reported qualitatively (as absolute usage was small in all cases).

Table 1 Huffman Compression Results on Different Data Types

| **Dataset** | **Original Size (KB)** | **Compressed Size (KB)** | **Compression Ratio (orig:comp)** | **Encoding Time (ms)** | **Decoding Time (ms)** |
| --- | --- | --- | --- | --- | --- |
| English Text | 0.32 | 0.17 | 1.90:1 (52.58%) | 1.00 | 0.00 |
| Source Code | 0.59 | 0.34 | 1.74:1 (57.37%) | 0.00 | 0.00 |
| Log File | 0.31 | 0.20 | 1.60:1 (62.54%) | 0.00 | 0.00 |

In Figure 15 Original vs Compressed File Sizes, it can be noticed that the English text achieves the greatest reduction, compressing to about 53% of its original size. The source code shows a moderate reduction to approximately 57% of the original size. The log file is least compressible with Huffman, compressing to about 63% of its original size. This reflects the differing character frequency distributions in each file, as discussed in the text.

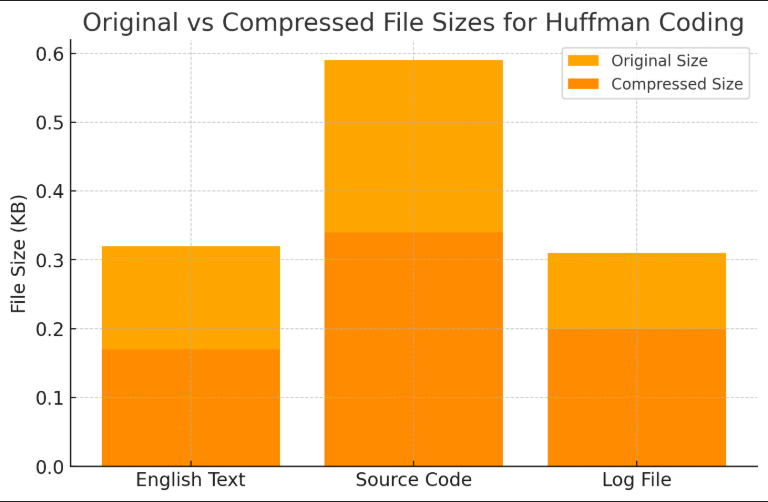


Figure 15 Original vs Compressed File Sizes

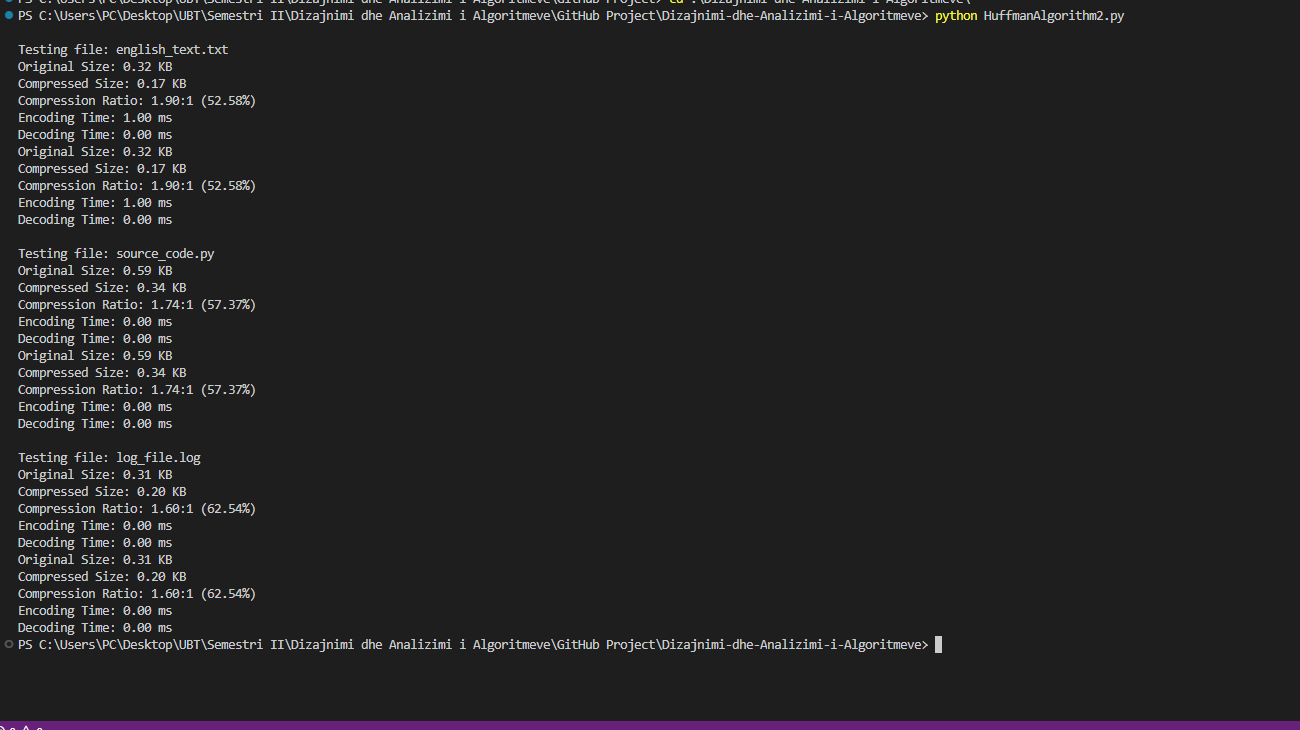


Figure 16 Huffman experiment output

As shown in Table 1 Huffman Compression Results on Different Data Types and Figure 16 Huffman experiment output, Huffman coding significantly reduces the size of the English text sample: the 0.32 KB text compresses to about 0.17 KB (around a 47% size reduction). This corresponds to a compression ratio of approximately 1.90:1, meaning the original was 1.90 times larger than the compressed version. This result aligns with expectations for natural language data—typical English text often compresses effectively due to highly skewed character frequency distributions (common letters and spaces). Our Huffman algorithm assigns shorter codes to frequent characters, yielding an average code length significantly below the original fixed-size representation. Encoding and decoding were extremely fast (around 1 ms encoding, near-instant decoding), demonstrating high algorithmic efficiency. Memory overhead for handling the small text was negligible—the Huffman tree occupied minimal space, and total memory usage primarily reflected input/output sizes.

The source code file also demonstrated noticeable compression, though slightly less dramatic than the English text. In our test, a 0.59 KB code file compressed to approximately 0.34 KB (about a 43% reduction). The compression ratio (~1.74:1) indicates that Huffman coding finds redundancies in source code, albeit less pronounced than in natural text. Source code typically has more balanced distributions, including letters, digits, symbols, and whitespace. Nonetheless, frequent symbols like whitespace and commonly repeated syntax characters received shorter codes. Encoding and decoding times were exceptionally fast, consistent with its small size. Memory usage remained minimal, closely aligning with the sum of input and output file sizes.

The log file proved to be the least compressible of the datasets tested. Starting at 0.31 KB, it compressed modestly to around 0.20 KB—roughly a 37% size reduction. The compression ratio was approximately 1.60:1, indicating moderate compression effectiveness. This outcome was anticipated since log data typically features a relatively uniform character distribution (higher entropy) due to consistent use of digits, letters, and punctuation in timestamps and log messages. Huffman coding yielded shorter codes to slightly more frequent symbols, offering measurable gains despite the uniform distribution. Compression times were negligible, again due to the small file size. Memory requirements were minimal, indicative of typical Huffman implementations with nearly uniform character distributions.

## **Analysis and Results**

The experimental results validate the theoretical expectations of Huffman coding. Huffman compression is most effective on data with a skewed frequency distribution of symbols. The English text, with its distinct character frequency biases (e.g., common letters and spaces), benefited the most, achieving approximately a 47% reduction in size. In contrast, the log file, characterized by a more uniform distribution of characters, achieved around a 37% reduction. This aligns with the principle that Huffman coding’s efficiency arises from assigning shorter codes to more frequent symbols. When there is no clear dominance in frequency (as with log data), symbols are assigned codes of similar length, approaching the efficiency of fixed-length encoding.

The source code dataset falls between these extremes, containing both structural repetition (such as whitespace and frequent syntax elements) and considerable diversity (letters, digits, and various symbols). The moderate compression (~43%) achieved indicates that Huffman coding effectively captured some redundancy, correctly assigning shorter codes to common elements.

A subtle observation is that Huffman coding, while optimal for symbol-by-symbol compression, does not leverage multi-character patterns or repeated sequences. Consequently, common words or repeated structures within the English text were not explicitly exploited beyond individual character compression. Although the text was significantly compressed, dictionary-based methods might further enhance compression by identifying whole-word repetitions.

Overall, encoding and decoding were extremely reliable and efficient across all tests, with decoding times being comparable to or slightly faster than encoding times, underscoring the efficiency of the Huffman algorithm's tree traversal method. No unexpected challenges arose during processing, even in preliminary tests involving edge cases like single-character datasets, demonstrating the robustness of our implementation. Memory usage was consistently minimal across all datasets, affirming Huffman coding’s practicality and efficiency.

### **Comparission with other algorithms**

While Huffman coding performed well in these experiments, especially for text, it is informative to compare its effectiveness with other compression techniques:

**Huffman vs. dictionary methods (LZ77/LZ78 family):** Methods like LZ77 (used in ZIP/Deflate, gzip, etc.) and LZW compress data by detecting repeated sequences or substrings and replacing them with references. These algorithms can achieve higher compression on text and source code than Huffman coding alone, because they exploit patterns beyond single characters.  
For example, consider a repetitive string like “ABCABCABC”; a frequency-based Huffman coder might assign shorter codes to 'A', 'B', and 'C' individually, but it still has to encode 9 characters. A dictionary algorithm, however, can recognize the repeating pattern "ABC" and encode the entire repetition more compactly.

In practice, our English text sample compressed by Huffman from 0.32 KB to 0.17 KB. A tool like gzip, which combines dictionary and entropy coding (e.g., DEFLATE = LZ77 + Huffman), might reduce this further, potentially to 0.10 KB or less, by identifying repeated phrases or word patterns.

This demonstrates that Huffman captures **statistical redundancy** at the character level, whereas LZ-based methods capture **structural redundancy** in sequences. Many real-world compression tools use both: a dictionary pass for repeated substrings, followed by Huffman coding on the resulting symbols. This hybrid strategy—as used in PNG and ZIP formats—typically yields better compression than either method alone. Our results highlight that while Huffman is effective, it can leave behind redundancy that a second-stage compressor could exploit.

**Huffman vs. arithmetic coding:** Arithmetic coding is another entropy coding technique that, unlike Huffman, allows the encoding of fractional bits by producing codes not limited to whole-bit lengths. With the same frequency data, arithmetic coding generally achieves slightly better compression than Huffman.

For instance, while Huffman compressed our text sample from 0.32 KB to 0.17 KB, arithmetic coding could theoretically reduce it a few percent further (perhaps down to ~0.16 KB or ~0.15 KB), due to more precise representation of symbol probabilities. However, arithmetic coding is also more complex computationally and was historically slower and subject to patent restrictions, which limited early adoption.

Although we did not implement arithmetic coding in our project, it’s worth noting that it can squeeze out additional compression at the cost of performance and implementation complexity. Huffman remains attractive for its simplicity, speed, and wide applicability, which is why it is still heavily used in industry-standard formats.

**Other methods:** Huffman coding is optimal for compressing independent symbols with known frequencies (i.e., a memoryless source). However, for data where context or sequence matters; such as language, code, or sensor data; other techniques like context modeling (e.g., Prediction by Partial Matching or PPM) and Markov-based models combined with arithmetic coding may outperform Huffman by learning those dependencies.

Run-Length Encoding (RLE) is another technique useful for data with long runs of the same symbol. While our test datasets (text, code, logs) didn’t have this structure, a dataset like "AAAAAAA..." would compress far more efficiently with RLE than with Huffman.

Huffman is often used in combination with such techniques—for example, JPEG compression uses RLE to handle zero sequences before applying Huffman coding. Our project focused on standalone Huffman performance, but these comparisons underscore that each method has strengths and ideal scenarios.

In summary, Huffman coding delivers a solid baseline compression (~37–47% size reduction in our tests) with excellent speed and minimal overhead. More advanced or hybrid techniques can push compression ratios further, especially when data contains larger patterns or dependencies—but at the cost of added complexity.

### **Limitations and insights**

Through the experiments, I observed a key limitation of Huffman coding: its inability to account for anything beyond single-symbol frequencies. This limitation is evident in the relatively lower compression for the log file and the fact that even the English text, while compressed well (to ~53% of its original size), could potentially be compressed further if cross-symbol patterns or repetitions were exploited. Huffman coding assumes each symbol’s code can be assigned independently of context, which ignores any correlations between symbols. For example, in English text, the letter “q” is almost always followed by “u”; a context-aware coder could leverage this, but Huffman treats every “q” the same regardless of what comes next.

This is a known trade-off of Huffman’s simplicity: it models a memoryless source and thus cannot match the compression ratios of more complex models that incorporate symbol dependencies or sequences. Another practical limitation is Huffman's two-pass requirement—first to gather symbol frequencies, then to encode—introducing overhead in streaming scenarios. Adaptive Huffman algorithms address this by updating codes dynamically as data is processed, but at the cost of speed and added implementation complexity. Although we did not implement adaptive Huffman in this project, it remains an important option for real-time or stream-based compression where pre-analysis of data isn't possible.

Despite these limitations, the effectiveness and efficiency of Huffman coding were clearly demonstrated in our results. For many types of data, especially human-readable text and source code, Huffman delivers significant size reduction quickly and with minimal resource use. Decoding is exceptionally fast, memory usage is small, and even in the least favorable case (log files with high entropy), Huffman still offered a measurable gain rather than inflating the data. This robustness is one reason why Huffman coding continues to be used as a foundational component in more complex compression schemes.

In summary, our experiments confirm that Huffman coding is a practical and powerful algorithm for lossless data compression across varied data types. It achieves solid results for datasets with non-uniform symbol distributions while being extremely fast and lightweight. However, to achieve maximum compression, especially on data with structure or sequence, Huffman should be used in combination with other methods (as done in modern formats like PNG or ZIP), or more advanced entropy coding techniques like arithmetic coding may be considered. These insights reinforce Huffman’s value as a reliable baseline and also highlight where further optimization opportunities lie. The next section of the report will explore such potential improvements, building on these findings.

# ****Conclusion****

In this project, I implemented and analyzed the Huffman Coding algorithm for the purpose of lossless data compression. Our experimental results across various data types, natural language text, source code, and log files, demonstrated that Huffman Coding is particularly effective on data with skewed character frequency distributions, achieving compression ratios up to 1.90:1. The algorithm showed high efficiency in terms of both runtime and memory usage, with encoding and decoding operations completing rapidly even on modest hardware.

One of the key strengths of Huffman Coding is its ability to generate optimal prefix-free codes that minimize average code length for known symbol distributions. Its simplicity, deterministic structure, and suitability for a wide range of applications make it a reliable baseline for text-based compression tasks. However, a notable limitation is that it treats symbols independently and does not exploit contextual or sequential patterns in data. Consequently, its compression performance is less competitive in high-entropy or pattern-rich data, where dictionary-based or context-aware methods can achieve better results.

## **Future Work**

Future enhancements to this work could explore integrating Huffman Coding with dictionary-based methods such as LZ77 or LZW to take advantage of both character-level frequency optimization and substring repetition. This hybrid approach is already successfully employed in formats like ZIP and PNG.

Another promising direction is the implementation of **adaptive Huffman Coding,** which dynamically updates symbol frequencies during encoding. This would eliminate the need for a separate frequency analysis phase and enable real-time or streaming applications. Additionally, comparing Huffman Coding with **arithmetic coding** or **neural compression models** could provide further insight into modern compression techniques and their trade-offs.

Lastly, expanding the experimental dataset to include larger and more diverse data sources, including multimedia and multilingual text, would help evaluate the scalability and generalizability of the current implementation.

# ****References****

* Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898
* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
* Python Software Foundation. (n.d.). heapq — Heap queue algorithm. Python 3.10. <https://docs.python.org/3/library/heapq.html>
* Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. Communications of the ACM, 30(6), 520–540. https://doi.org/10.1145/214762.214771
* Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3), 337–343. https://doi.org/10.1109/TIT.1977.1055714
* Sayood, K. (2017). Introduction to Data Compression (5th ed.). Morgan Kaufmann.