# The Effect of Audio Signal Patterns on Huffman Coding Compression Efficiency

Edbert Fernando - 13525111
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: edbertfernando09@gmail.com , 13525111@std.stei.itb.ac.id

***Abstract*—Audio signals can have different patterns, including repetitive periodic signals and random noise. In digital form, these patterns create different symbol frequency distributions, represented as samples or bytes. This paper analyzes how audio signal patterns affect the compression efficiency of Huffman Coding. Huffman Coding is a lossless compression method that assigns shorter binary codes to frequent symbols and longer codes to rare symbols. Several signal types are compared through their symbol distributions, average code lengths, Huffman encoded sizes ,compression ratios, and space saving values. The analysis focuses on how regularity or randomness influences the generated code. This study demonstrates the application of binary trees and prefix codes in lossless audio compression**

***Keywords—audio signal, Huffman coding, binary tree, prefix code, compression efficiency***

## I. INTRODUCTION

Digital audio can be viewed as a collection of numerical data that represents changes in sound amplitude over time [2]. Different audio signals may have different characteristics depending on their structure. Some signals have regular and repetitive patterns, such as periodic waves, while others have irregular or random variations, such as noise. These differences can affect the distribution of values within the audio data, which later influences how efficiently the data can be compressed.

Data compression is a technique used to represent data using fewer bits while preserving the required information [3]. One lossless compression technique that is closely related to discrete mathematics is Huffman Coding [1]. This method forms binary codes based on the frequency of each symbol in the data. Therefore, the performance of Huffman Coding depends on how the symbols are distributed in the input.

In the context of audio data, signal patterns can affect how frequently certain values appear. Repetitive signals may contain more recurring values, while random signals may produce values that are more evenly distributed. This makes audio signals an interesting object to analyze in relation to Huffman Coding compression efficiency.

This paper discusses how different audio signal patterns influence the compression efficiency of Huffman Coding. The analysis is carried out by comparing several types of audio signals based on their symbol frequency distributions, average code lengths, and compression ratios. Through this discussion, the paper aims to show the relationship between audio signal characteristics and discrete mathematics concepts, especially binary trees and prefix codes.

## II. THEORETICAL FRAMEWORK

### *A. Digital Audio Representation*

Digital audio is a representation of sound in the form of discrete numerical values. A sound signal that is originally continuous is sampled at certain time intervals and stored as a sequence of values [2]. Each value represents the amplitude of the sound signal at a specific point in time. In data compression, these values can be treated as symbols that form the input data. Therefore, an audio signal can be analyzed not only as a sound wave, but also as a sequence of discrete symbols.

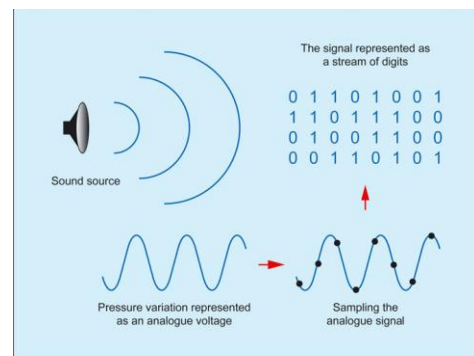


Fig. 1. Conversion of a continuous audio signal into discrete digital samples
(Source: https://www.soundonsound.com/techniques/digital-myth)

### *B. Symbol Frequency Distribution*

Symbol frequency distribution represents the number of occurrences of each symbol in a data sequence. In digital audio, a symbol may refer to a sample value or a byte value obtained from the audio data. The distribution of these symbols can vary depending on the pattern of the audio signal. A repetitive signal may cause some values to appear more frequently, while a random signal may produce values that are more evenly distributed. Since Huffman Coding uses symbol frequency as the basis for constructing its code, this distribution has an important role in determining the compression performance [3].

### C. *Binary Tree and Prefix Code*

A binary tree is a tree structure in which each node has at most two children. In coding theory, a binary tree can be used to represent binary codes by assigning a binary digit to each branch. A path from the root node to a leaf node then forms the codeword of a symbol. This structure is useful because different paths in the tree can represent different binary codes.

A prefix code is a code system in which no codeword is the prefix of another codeword. This property allows an encoded bit sequence to be decoded uniquely without ambiguity [3]. Huffman Coding produces prefix codes by placing symbols only at the leaf nodes of a binary tree [3]. As a result, the code of one symbol cannot become the beginning of another symbol's code.

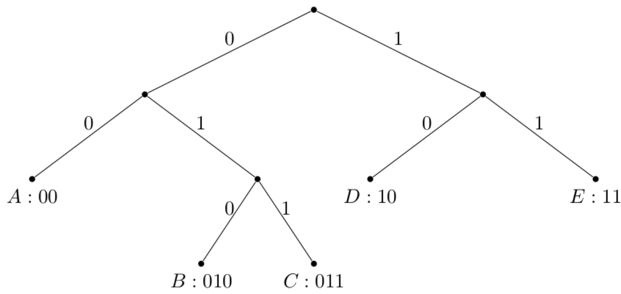


Fig. 2. Binary tree representation of prefix codes
(Source: https://leimao.github.io/blog/Huffman-Coding/)

### D. *Huffman Coding*

Huffman Coding is a lossless compression algorithm that produces variable-length binary codes based on symbol frequencies [1]. The main idea is to assign shorter codes to symbols that appear more frequently and longer codes to symbols that appear less frequently. Huffman introduced this method as a way to construct minimum-redundancy codes by minimizing the average number of coding digits needed to represent a set of messages [1].

Let $P(i)$ denote the probability of the $i$-th symbol in a set of $N$ symbols. The probabilities of all symbols satisfy [1]

$$\sum_{i=1}^{N} P(i) = 1$$

If $L(i)$ denotes the length of code assigned to the $i$-th symbol, then the average code length is given by [1]

$$L_{avg} = \sum_{i=1}^{N} P(i)L(i) \quad (2)$$

The construction of a Huffman tree begins by creating a node for each symbol and its frequency. The two nodes with the lowest frequencies are repeatedly combined into a new node whose frequency is the sum of the two original frequencies. This process is repeated until only one tree remains [3]. The binary code of each symbol is obtained by tracing the path from the root to the corresponding leaf node.

The efficiency of Huffman Coding depends on the symbol frequency distribution [3]. When some symbols appear much more frequently than others, Huffman Coding can produce a smaller average code length. However, when the symbols have nearly uniform frequencies, the generated codes tend to have similar lengths, which may reduce the compression efficiency.

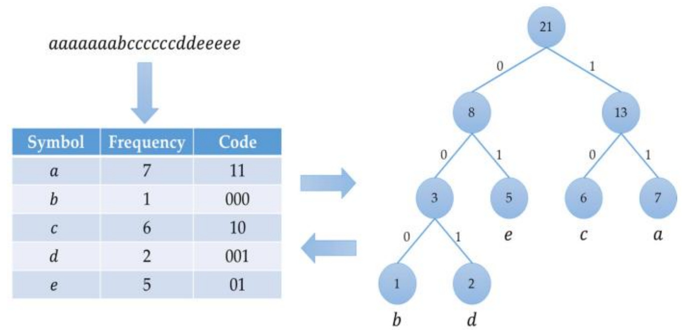


Fig. 3. Example of Huffman tree construction and the resulting binary codes.
(Source: https://www.mdpi.com/2073-8994/16/11/1419)

### E. *Compression Efficiency Metrics*

Compression efficiency is evaluated to determine how effectively Huffman coding reduces the number of bits needed to represent the input data. In data compression, common evaluation measures include average codeword length and compression ratio [3]. In this paper, the compression result is evaluated using average code length, original size, compressed size, compression ratio, and space saving.

Average code length represents the expected number of bits required to encode one symbol after Huffman Coding is applied. This value is obtained by weighting the length of each codeword by the probability of the corresponding symbol [3]. A smaller average code length indicates that the generated code is more efficient, because fewer bits are needed on average to represent each symbol.

The original size represents the number of bits required to store the data before compression. If the audio data is treated as a sequence of bytes, the original size is calculated as

$$Original\ Size = n \times 8 \quad (3)$$

where n is the number of symbols or bytes in the input data.

In this paper, compressed size refers to Huffman encoded size, which represents the total number of bits required after each symbol is replaced by its Huffman code. It can be calculated by multiplying the frequency of each symbol by the length of its codeword and summing the result

$$Huffman\ Encoded\ Size = \sum_{i=1}^{N} f_i\, L(i) \quad (4)$$

where $f_i$ is the frequency of the $i$-th symbol and L($i$) is the length of the Huffman Code assigned to that symbol.

Compression ratio is commonly used to compare the size of the compressed data with the size of original data, although its definition may vary depending on the convention used. In this paper, compression ratio is defined as the compressed size divided by the original size.

$$Compression\ Ratio = \frac{Huffman\ Encoded\ Size}{Original\ Size} \quad (5)$$

With this definition, a smaller compression ratio means better compression performance because the compressed data occupies a smaller portion of the original size. Space saving is then used to express the reduction in data size as a percentage.

$$Space\ Saving = (1 - \frac{Huffman\ Encoded\ Size}{Original\ Size}) \times 100\% \quad (6)$$

A higher space saving value indicates that more storage space has been reduced. These metrics are used to compare the compression performance of Huffman Coding across different audio signal patterns.

## III. Methodology

### A. Research Design

This study uses a comparative experimental approach to examine the relationship between audio signal patterns and Huffman Coding compression efficiency. Several synthetic audio signals with different structural characteristics are prepared and compressed using the same procedure. The purpose of using synthetic signals is to control the duration, sampling rate, and data representation so that the comparison between signal types can be done consistently.

The main factor observed in this study is the pattern of the audio signal. Different signal patterns are expected to produce different symbol frequency distributions, which may affect the average code length and final comparison result. Each signal is compressed using Huffman Coding and evaluated using the compression efficiency metrics described in the theoretical framework.

### B. Audio Signal Samples

The audio samples used in this study are generated using Python. The signals are selected to represent different levels of regularity and randomness. Regular signals are represented by sine wave, square wave, and sawtooth wave. A silence or low-variation signal is used to represent audio data with repeated or nearly identical values, while white noise is used to represent random audio data with more evenly distributed values.

These signal types are chosen because each of them has a different pattern. A sine wave has a smooth periodic structure, a square wave has a repetitive structure with limited amplitude values, and a sawtooth wave has a periodic structure with gradual changes in value. In contrast, the white noise does not have a clear repetitive pattern. By comparing these signals , this study observes how different audio patterns influence the performance of Huffman Coding

TABLE I. AUDIO SIGNAL SAMPLES USED IN THE EXPERIMENT

| Signal Type | Pattern Characteristic |
|---|---|
| Silence / Low Variation | Repeated or nearly identical values |
| Sine Wave | Smooth periodic pattern |
| Square Wave | Repetitive pattern with limited amplitude values |
| Sawtooth Wave | Periodic pattern with gradual value change |
| White Noise | Random pattern with more evenly distributed values |

### C. Data Representation

Before compression, each generated audio signal is converted into a sequence of byte values. In this representation, each byte is treated as one symbol in the Huffman Coding process. Byte representation is used because it provides a fixed symbol range from 0 to 255, making it easier to count symbol frequencies and compare the results across different signal types.

After the audio signal is represented as byte values, the frequency of each symbol is calculated. This frequency distribution is then used as the input for the Huffman Coding process. Since all signals are processed using the same representation, the differences in compression results are expected to come mainly from the differences of signal patterns.

### D. Evaluation Procedure

The same evaluation procedure is applied to every audio signal sample. First, the original size of each signal is calculated based on the number of byte values in the data. Next, Huffman Coding is applied to generate a binary code for each symbol based on its frequency. The Huffman encoded size is then calculated from the frequency of each symbol and the length of its Huffman code.

The compression results are compared using average code length, Huffman encoded size, compression ratio, and space saving. These metrics are used to determine how efficiently each signal pattern can be compressed. Signals with less uniform symbol distribution are expected to produce shorter average code lengths and better compression results. Meanwhile, signals with more uniform distributions are expected to produce lower compression efficiency because their Huffman codes tend to have more similar lengths.

## IV. Implementation

### A. Program Environment

The experiment was implemented using Python. Python was used to generate synthetic audio signals, convert signal values into byte values, apply Huffman Coding, and calculate the compression results. The implementation uses numerical arrays

to represent audio samples and frequency counting to determine how often each byte value appear in the data.

The main stages of the program consist of signal generation, byte conversion, symbol frequency calculation, Huffman tree construction, Huffman code generation, and compression metric calculation. Each signal is processed using the same program so that the comparison between signal types remains consistent.

## B. *Signal Generation and Byte Conversion*

The first part of the program generates five synthetic audio signals: silence or low-variation signal, sine wave, square wave, sawtooth wave, and white noise. All signals are generated using the same sampling rate of 8000 Hz and duration of 2 s, resulting in 16000 samples for each signal. The periodic signals are generated using a frequency of 440 Hz. This ensures that each signal has the same number of data points before compression.

```
#Generating 5 DIfferent Signals
def generate_signals(fs=8000, duration=2.0, frequency=440):
    t = np.linspace(0, duration, int(fs * duration), endpoint=False)
    rng = np.random.default_rng(42)

    signals = {
        "Silence / Low Variation": np.zeros_like(t),

        "Sine Wave": np.sin(2 * np.pi * frequency * t),

        "Square Wave": np.where(np.sin(2 * np.pi * frequency * t) >= 0,1,-1),

        "Sawtooth Wave": (2 * (frequency * t - np.floor(0.5 + frequency * t))),

        "White Noise": rng.uniform(-1, 1, len(t))
    }
    return signals, t
```

Fig. 4. Signal Generation
(Source: Author)

After the signals are generated, their amplitude values are normalized into the range from -1 to 1. The normalized values are then converted into byte values ranging from 0 to 255. In this implementation, each byte value is treated as one symbol in the Huffman Coding process. This conversion allows the program to count symbol frequencies directly and use them as the basis for constructing the Huffman code.

```
def convert_to_byte_values(signal):

    clipped_signal = np.clip(signal, -1, 1)
    byte_values = np.round((clipped_signal + 1) * 127.5).astype(np.uint8)
    return byte_values
```

Fig. 5. Signal Conversion to Bytes
(Source: Author)

## C. *Huffman Coding Program*

In this implementation, the Huffman Coding process is divided into three main functions: frequency counting, Huffman tree construction, and Huffman code generation. The program first receives the byte sequence produced from the audio signal conversion step. The byte sequence is then converted into integer values and counted using a frequency counter. The resulting frequency table is used as the input for building the Huffman tree.

The Huffman tree is constructed using a priority queue. At the beginning, each symbol is represented as a node with its corresponding frequency. The program repeatedly selects the two nodes with the smallest frequencies and merges them into a new node. The frequency of the new node is the sum of the two selected nodes. This process continues until only one root node remains.

```
def build_huffman_tree(frequencies):

    counter = itertools.count()
    heap = []

    for symbol, frequency in frequencies.items():
        heapq.heappush(heap, (frequency, next(counter), symbol))

    if len(heap) == 1:
        return heap[0][2]

    while len(heap) > 1:
        freq1, _, node1 = heapq.heappop(heap)
        freq2, _, node2 = heapq.heappop(heap)

        merged_node = (node1, node2)
        merged_frequency = freq1 + freq2

        heapq.heappush(heap, (merged_frequency, next(counter), merged_node))

    return heap[0][2]
```

Fig. 6. Building the Huffman Tree
(Source: Author)

After the tree is formed, the Huffman code for each symbol is obtained by traversing the tree from the root node to each leaf node. A left branch is represented by 0, while a right branch is represented by 1. The path from the root to the symbol determines the binary code assigned to that symbol.

```
def generate_huffman_codes(tree):

    codes = {}

    def traverse(node, current_code):
        if not isinstance(node, tuple):
            codes[node] = current_code if current_code != "" else "0"
            return

        left, right = node
        traverse(left, current_code + "0")
        traverse(right, current_code + "1")

    traverse(tree, "")
    return codes
```

Fig. 7. Generating the Huffman Code
(Source: Author)

The main Huffman Coding function connects all previous steps. It counts the frequency of each byte value, constructs the Huffman tree from the frequency table, generates the Huffman code for each symbol, and returns both the frequency table and the generated codes.

```python
def huffman_coding(byte_data):

    data_list = [int(x) for x in byte_data]
    frequencies = Counter(data_list)

    tree = build_huffman_tree(frequencies)
    codes = generate_huffman_codes(tree)

    return frequencies, codes
```

Fig. 8. Main Huffman Coding
(Source: Author)

### D. Compression Metric Calculation

After the Huffman codes are generated, the program uses the frequency table and the code length of each symbol to calculate the compression metrics. This step is implemented in a separate function so that the same calculation can be applied consistently to all generated signals. The program produces several output values, including the number of unique symbols, average code length, original size, Huffman encoded size, compression ratio, and space saving percentage.

In this study, the Huffman encoded size represents the number of bits required to encode the data using the generated Huffman codes. The program does not create a complete compressed audio file, because the additional overhead required to store the Huffman tree, codebook, padding, and metadata is not included. This approach is used because the main objective of this study is to compare how different audio signal patterns affect Huffman Coding efficiency under the same encoding procedure.

```python
def calculate_compression_metrics(byte_data):

    frequencies, codes = huffman_coding(byte_data)

    number_of_symbols = len(byte_data)
    unique_symbols = len(frequencies)

    original_size = number_of_symbols * 8

    compressed_size = sum(
        frequencies[symbol] * len(codes[symbol])
        for symbol in frequencies
    )

    average_code_length = compressed_size / number_of_symbols
    compression_ratio = compressed_size / original_size
    space_saving = (1 - compression_ratio) * 100

    return {
        "Unique Symbols": unique_symbols,
        "Average Code Length": average_code_length,
        "Original Size (bits)": original_size,
        "Huffman Encoded Size (bits)": compressed_size,
        "Compression Ratio": compression_ratio,
        "Space Saving (%)": space_saving
    }, frequencies, codes
```

Fig. 9. Calculating the Compression Metrics
(Source: Author)

In addition to the numerical compression metrics, the program also generates supporting output files. These outputs include waveform visualizations, symbol frequency distribution graphs, a table image of the compression results, and a Huffman codebook in CSV format. The visualizations are used to support the analysis of signal patterns and symbol distributions, while the codebook file records the generated Huffman code, frequency, and code length for each symbol.

```python
#Visualization
def make_safe_filename(signal_name):
    return signal_name.replace("/", "-").replace(" ", "_")


def plot_waveform(signal_name, signal, t, output_folder):
    plt.figure(figsize=(8, 3))

    plt.plot(t[:500], signal[:500])

    plt.title(f"Waveform of {signal_name}")
    plt.xlabel("Time (seconds)")
    plt.ylabel("Amplitude")
    plt.tight_layout()

    safe_name = make_safe_filename(signal_name)
    file_path = output_folder / f"{safe_name}_waveform.png"

    plt.savefig(file_path, dpi=300)
    plt.close()


def plot_symbol_distribution(signal_name, frequencies, output_folder):
    symbols = sorted(frequencies.keys())
    counts = [frequencies[symbol] for symbol in symbols]

    plt.figure(figsize=(8, 3))
    plt.bar(symbols, counts, width=0.8)

    plt.title(f"Symbol Frequency Distribution of {signal_name}")
    plt.xlabel("Byte Value")
    plt.ylabel("Frequency")

    plt.xlim(-5, 260)
    plt.xticks([0, 64, 128, 192, 255])

    plt.tight_layout()

    safe_name = make_safe_filename(signal_name)
    file_path = output_folder / f"{safe_name}_distribution.png"

    plt.savefig(file_path, dpi=300)
    plt.close()
```

Fig. 10. Snippet code for the visualization
(Source: Author)

## V. RESULTS AND DISCUSSION

### A. Compression Result

After the Huffman Coding Program was applied to each generated audio signal, the compression metrics were obtained. The metrics include the number of unique symbols, average code length, original size, Huffman encoded size, compression ratio, and space saving percentage. Since all signals were generated using the same sampling rate of 8000 Hz and duration of 2 s, each signal consists of 16000 samples and has the same original size of 128000 bits.

```
=== Huffman Coding Compression Results ===

Signal Type                  : Silence / Low Variation
Unique Symbols               : 1
Average Code Length          : 1.000000
Original Size (bits)         : 128000
Huffman Encoded Size (bits)  : 16000
Compression Ratio            : 0.125000
Space Saving (%)             : 87.500000
-------------------------------------------------------
Signal Type                  : Sine Wave
Unique Symbols               : 94
Average Code Length          : 6.580000
Original Size (bits)         : 128000
Huffman Encoded Size (bits)  : 105280
Compression Ratio            : 0.822500
Space Saving (%)             : 17.750000
-------------------------------------------------------
Signal Type                  : Square Wave
Unique Symbols               : 2
Average Code Length          : 1.000000
Original Size (bits)         : 128000
Huffman Encoded Size (bits)  : 16000
Compression Ratio            : 0.125000
Space Saving (%)             : 87.500000
-------------------------------------------------------
```

Fig. 11. Program output for silence/low variation, sine wave, and square wave (Source: Author)

```
Signal Type                  : Sawtooth Wave
Unique Symbols               : 206
Average Code Length          : 7.738625
Original Size (bits)         : 128000
Huffman Encoded Size (bits)  : 123818
Compression Ratio            : 0.967328
Space Saving (%)             : 3.267188
-------------------------------------------------------
Signal Type                  : White Noise
Unique Symbols               : 256
Average Code Length          : 7.999125
Original Size (bits)         : 128000
Huffman Encoded Size (bits)  : 127986
Compression Ratio            : 0.999891
Space Saving (%)             : 0.010937
-------------------------------------------------------
```

Fig. 12. Program output for sawtooth wave and white noise (Source: Author)

### B. *Symbol Distribution Analysis*

The result shows that the compression efficiency of Huffman Coding is strongly influenced by the number of unique symbols and their frequency distribution. The silence or low-variation signal has only one unique symbol, while the square wave has only two unique symbols. Because the number of symbols is very small, the generated Huffman codes have an average code length of 1 bit per symbol. As a result, both signals achieved a high space saving value of 87.50%

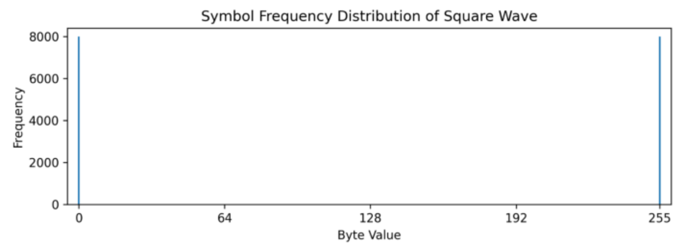


Fig. 13. Symbol frequency distribution of square wave (Source: Author)

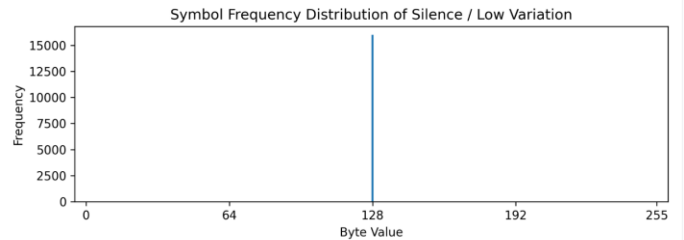


Fig. 14. Symbol frequency distribution of silence/low-variation (Source: Author)

The sine wave produces 94 unique symbols with an average code length of 6.580 bits. Although the sine wave is a periodic signal, its amplitude changes smoothly, causing more byte values to appear after the conversion. However, the distribution is still not completely uniform, so Huffman Coding can still reduce the encoded size by assigning shorter codes to more frequent symbols.

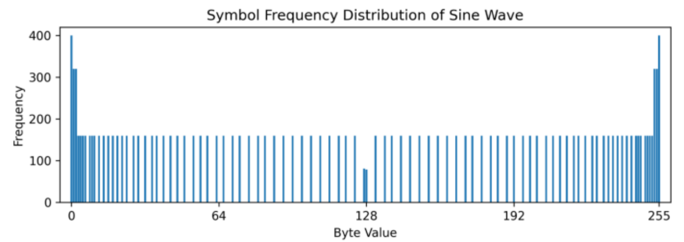


Fig. 15. Symbol frequency distribution of sine wave (Source: Author)

The sawtooth wave produces 206 unique symbols with an average code length of 7.739 bits. This value is close to the original 8-bit representation, which means that the compression efficiency is relatively low. This occurs because the sawtooth wave changes gradually through many byte values, resulting in a more evenly distributed set of symbols.

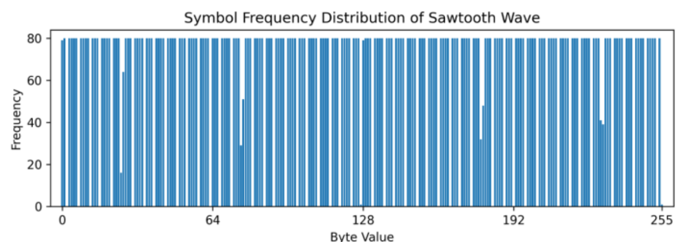


Fig. 16. Symbol frequency distribution of sawtooth wave (Source: Author)

White noise produces 256 unique symbols with an average code length of 7.999 bits. Similar to the sawtooth wave, the symbol distribution of white noise is spread across many byte values. Therefore, the generated Huffman codes tend to have similar lengths, making the Huffman encoded size close to the original data size.

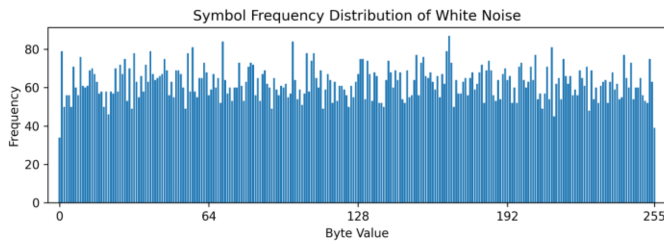


Fig. 17. Symbol frequency distribution of white noise
(Source: Author)

### C. Compression Efficiency Discussion

The highest compression efficiency is achieved by the silence or low-variation signal and the square wave. Both signals have a compression ratio of 0.125 and a space saving of 87.50%. This result occurs because their data contains very few unique symbols, allowing Huffman Coding to represent the data using much fewer bits than the original 8-bit representation.

The sine wave has moderate compression efficiency, with a compression ratio of 0.823 and a space saving of 17.75%. This shows that a periodic signal is not automatically compressed with very high efficiency. Although the sine wave has a regular pattern, it still produces many different byte values after conversion.

The sawtooth wave and white noise have low compression efficiency. Their compression ratios are 0.967 and approximately 1, respectively, with space saving values of 3.27% and 0.01%. These results indicate that signals with many unique symbols and relatively even symbol distributions are difficult to compress using Huffman Coding.

Overall, the experiment shows that compression efficiency is not determined only by whether a signal looks regular or random. A periodic signal can still have low compression efficiency if its byte values are spread relatively evenly. Therefore, the symbol frequency distribution after byte conversion is the main factor that determines the effectiveness of Huffman Coding in this experiment.

It should also be noted that the reported Huffman encoded size does not include the overhead required to store the Huffman tree, codebook, padding, or metadata. Therefore, the results are used to compare the relative compression efficiency between signal types rather than to represent the final size of a complete compressed audio file.

## VI. Conclusion

This paper analyzed the effect of audio signal patterns on Huffman Coding compression efficiency. Based on the experiment, signals with concentrated byte-value distributions, such as silence or low-variation signal and square wave, achieve higher compression efficiency than signals whose byte values are distributed more evenly, such as sawtooth wave and white noise.

The results also show that periodicity alone does not guarantee high compression efficiency. Although sine wave and sawtooth wave are both periodic signals, they produce different compression results because their byte-value distributions are different after conversion. Therefore, the symbol frequency distribution in the digital representation is the main factor that affects Huffman Coding efficiency.

This study demonstrates the application of discrete mathematics concepts, especially binary trees and prefix codes, in lossless compression analysis. The reported Huffman encoded size is used for relative comparison between signal types and does not include the overhead required to store the Huffman tree, codebook, padding, or metadata.

## Appendix

The source code and generated outputs used in this experiment are available in the following GitHub repository: https://github.com/Edbertf/Huffman-Audio-Compression-Analysis

## Acknowledgment

The author would like to express gratitude to God Almighty for His blessings and guidance throughout the process of writing this paper. The author would also like to express gratitude to Dr. Ir. Rinaldi, M.T., as the lecturer of the IF1220 Discrete Mathematics course, for the knowledge, guidance, and learning materials provided throughout the semester. In addition, the author appreciates the support and encouragement from both of the author's parents. Last but not least, the author would also like to thank Mr. David A. Huffman for inventing the Huffman code, which heavily inspired this paper.

## References

[1] D. A. Huffman, "A method for the construction of minimum-redundancy codes," Proceedings of the I.R.E., vol. 40, no. 9, pp. 1098-1101, September. 1952.

[2] L. M. Po, "Sampling, reconstruction, and quantization," EE4015 Multimedia Technologies, City University of Hong Kong, lecture slides, 2022. https://www.ee.cityu.edu.hk/~lmpo/ee4015/pdf/2022_EE4015_L03B_Sampling.pdf. Accessed on Jun 13, 2026.

[3] D. A. Lelewer and D. S. Hirschberg, "Data compression," ACM Computing Surveys, vol. 19, no. 3, pp. 261-296, September. 1987.

[4] R. Munir, "Pohon (Bag. 2)," IF1220 Matematika Diskrit, Program Studi Teknik Informatika, STEI-ITB, lecture slides, 2026. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf. Accessed on Jun 13, 2026.

## PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2025

Edbert Fernando, 13525111