

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**PROJECT REPORT**

Course Code: BCSE303L

Course Name: Operating System

Class Number:CH2024250502069

Slot: C1+TC1

Faculty: Dr. Sambath M

Student Name: Veditha. R

Registration Number: 23BCE1301

Date: 16th April 2025

*A project report on*

**ANALYSIS OF TEXT COMPRESSION ALGORITHMS**

*Submitted in partial fulfillment for the course of*

**BCSE303L Operating Systems**

**Faculty: Dr. SAMBATH M**

**Class Number: CH2024250502069**

**Slot: C1+TC1**

*by*

**VEDITHA R 23BCE1301**



**DECLARATION**

I hereby declare that the thesis entitled “ANALYSIS OF TEXT COMPRESSION ALGORITHMS” submitted by VEDITHA R (23BCE1301), for the course of Operating Systems is a record of Bonafide work carried out by me under the supervision of Dr. Sambath M.

I further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Chennai

Date:15/04/2025 Signature of the Candidate



**School of Computer Science and Engineering**

CERTIFICATE

This is to certify that the report entitled **“Analysis of Text Compression Algorithms”** is prepared and submitted by Veditha R (23BCE1301) to Vellore Institute of Technology, Chennai, in partial fulfillment of the requirement for the course Operating Systems is a bonafide record carried out under my guidance. The project fulfills the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

Signature of the Guide:

Name: Dr./Prof.

Date:

**ABSTRACT**

Efficient storage and transmission of textual data is a fundamental concern in modern computing, particularly within the domain of operating systems where resource management and performance optimization are critical. This project, titled "Analysis of Text Compression Algorithms," investigates and compares the effectiveness of several widely used text compression techniques, providing both theoretical insights and practical evaluations. The study is implemented as a web-based application, allowing users to upload text files and observe the performance of different algorithms in real time.

The project focuses on five prominent compression algorithms: Run-Length Encoding (RLE), Huffman Coding, Lempel-Ziv-Welch (LZW), Deflate, and Arithmetic Coding. Each algorithm is implemented in Python and integrated into a user-friendly web interface using the Flask framework. The application enables users to compress and decompress text files, and it systematically records key performance metrics such as compression ratio, compression time, and decompression time. These results are presented in a comparative table, offering a clear visualization of the strengths and weaknesses of each method.

Through extensive experimentation, the project highlights the trade-offs inherent in text compression. Simpler algorithms like RLE offer fast execution but are only effective for highly repetitive data, while more sophisticated methods such as Huffman and Arithmetic Coding achieve better compression ratios at the cost of increased computational complexity. The Deflate algorithm, widely used in real-world applications, demonstrates a balanced performance, combining speed and efficiency. LZW, known for its dictionary-based approach, provides robust results for a variety of text types.

The findings of this project underscore the importance of selecting an appropriate compression algorithm based on the specific requirements of the operating environment, such as the nature of the data, available computational resources, and the need for real-time processing. By providing both a practical tool and a detailed analysis, this project serves as a valuable resource for students and professionals seeking to understand the operational characteristics and practical implications of text compression algorithms in the context of operating systems.

**ACKNOWLEDGEMENT**

It is my pleasure to express with deep sense of gratitude to Dr. Sambath M, Assistant Professor, School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, for his constant guidance, continual encouragement, understanding; more than all, he taught me patience in my endeavor. My association with him is not confined to academics only, but it is a great opportunity on my part of work with an intellectual and expert in the field of Operating Systems.

It is with gratitude that I would like to extend my thanks to the visionary leader Dr. G. Viswanathan our Honorable Chancellor, Mr. Sankar Viswanathan, Dr. Sekar Viswanathan, Dr. G V Selvam Vice Presidents, Dr. Sandhya Pentareddy, Executive Director, Ms. Kadhambari S. Viswanathan, Assistant Vice-President, Dr. V. S. Kanchana Bhaaskaran Vice-Chancellor, Dr. T. Thyagarajan Pro-Vice Chancellor, VIT Chennai and Dr. P. K. Manoharan, Additional Registrar for providing an exceptional working environment and inspiring all of us during the tenure of the course.

Special mention to Dr. Ganesan R, Dean, Dr. Parvathi R, Associate Dean Academics, Dr. Geetha S, Associate Dean Research, School of Computer Science and Engineering, Vellore Institute of Technology, Chennai for spending their valuable time and efforts in sharing their knowledge and for helping us in every aspect.

My sincere thanks to all the faculties and staffs at Vellore Institute of Technology, Chennai who helped me acquire the requisite knowledge. I would like to thank my parents for their support. It is indeed a pleasure to thank my friends who encouraged me to take up and complete this task.

Place: Chennai

Date: **Veditha R**

**CONTENTS PAGE NO**

**INTRODUCTION 8**

**OBJECTIVE 9**

**COMPRESSION TECHNIQUES 10**

**METHODOLOGY 13**

**IMPLEMENTATION OVERVIEW 14**

**EVALUATION METRICS 16**

**CODES IMPLEMENTED 17**

**OUTPUT 24**

**GRAPHICAL ANALYSIS 26**

**CONCLUSION 29**

**REFERENCES 31**

![A black background with a black square

AI-generated content may be incorrect.](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADYAAABDCAYAAADXqvavAAAAj0lEQVR4Xt3IMQEAAAzDoPk33QmIAw4ebtuJEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhOIBGng+EU67+qYAAAAASUVORK5CYII=)

**Introduction**

The ever-increasing volume of digital text data has made efficient storage and transmission a central concern in modern computing. Text compression algorithms address this challenge by reducing the size of textual data, enabling more effective use of storage resources and faster data transfer across networks. In the context of operating systems, which are responsible for managing files and optimizing system performance, the choice of compression algorithm can have a significant impact on both resource utilization and user experience.

Text compression is achieved through lossless algorithms that eliminate redundancy and encode information in a more compact form, ensuring that the original data can be perfectly reconstructed. Several well-established algorithms are commonly used for this purpose, including Run-Length Encoding (RLE), Huffman Coding, Lempel-Ziv-Welch (LZW), Deflate, and Arithmetic Coding. Each of these algorithms employs distinct strategies, such as pattern substitution, statistical encoding, or dictionary-based methods, to achieve compression.

This project presents a comparative analysis of these text compression algorithms, focusing on three key performance parameters:

* **Compression Ratio:** The effectiveness of each algorithm in reducing the size of the input text, which directly translates to storage and bandwidth savings.
* **Compression Time:** The amount of time required to compress the data, reflecting the computational efficiency of the algorithm.
* **Decompression Time:** The speed at which the original data can be restored from its compressed form, which is crucial for applications requiring rapid data retrieval.

To provide a clear and comprehensive comparison, the results for each algorithm are visualized using graphs, allowing for intuitive analysis of their relative strengths and weaknesses across different input files2. This approach highlights the trade-offs between compression efficiency and processing speed, which are essential considerations for practical deployment in operating systems.

By systematically evaluating these algorithms using quantitative metrics and graphical representations, the project aims to offer insights into their suitability for various real-world scenarios.



**Objective**

The primary objective of this project is to explore, implement, and evaluate the effectiveness of five widely used text compression algorithms: **Run-Length Encoding (RLE)**, **Huffman Coding**, **Lempel-Ziv-Welch (LZW)**, **DEFLATE**, and **Arithmetic Coding**. These algorithms were selected for their foundational roles in data compression and their diverse approaches to reducing file sizes. The focus lies not only on their theoretical strengths but also on how they perform in practical scenarios involving real-world text data.

To facilitate an in-depth analysis, the project examines three core performance metrics for each algorithm:

1. **Compression Ratio** – The size of the compressed file relative to the original file, highlighting the algorithm’s space-saving efficiency.
2. **Compression Time** – The time taken to compress the input data, which is crucial in time-sensitive applications.
3. **Decompression Time** – The time required to restore the original data, reflecting the algorithm’s responsiveness during data retrieval.

In addition to algorithmic implementation, a user-friendly **web interface** has been developed. This interface allows users to upload files, choose among compression methods, preview compressed and decompressed outputs, and download the results. It also visually presents the comparison between the algorithms in the form of tables and graphs, enabling an intuitive understanding of their strengths and trade-offs.

By combining backend processing with frontend interactivity, the project aims to serve as both an educational tool and a prototype for compression-based applications. Ultimately, the objective is to identify the most efficient algorithm based on context, while offering users an interactive experience to witness the entire compression-decompression workflow.

![A black background with a black square

AI-generated content may be incorrect.](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADYAAABDCAYAAADXqvavAAAAj0lEQVR4Xt3IMQEAAAzDoPk33QmIAw4ebtuJEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhOIBGng+EU67+qYAAAAASUVORK5CYII=)

**Compression Techniques Used**

**Run-Length Encoding (RLE)**

Run-Length Encoding (RLE) is a straightforward lossless data compression algorithm that encodes consecutive repeating elements in data into a single value and a count. It is particularly suited for compressing data that contains many repeated characters. The core concept is to replace sequences of the same data value occurring in consecutive elements with a single data value and a count that specifies how many times the value is repeated.

For example, consider the string AAAABBBCCDAA. RLE would compress this into 4A3B2C1D2A, indicating four occurrences of 'A', followed by three 'B's, two 'C's, one 'D', and two 'A's. The decoder can easily reconstruct the original data using these counts and characters. RLE is applicable to various data types, including text and image data, particularly those with many repeating patterns such as black-and-white images or simple geometric drawings.

The algorithm involves scanning the input data sequentially, tracking runs of characters, and recording their length along with the character itself. It requires minimal logic to implement and is easy to decode since it operates in a linear fashion. The simplicity of RLE makes it a foundational concept in data compression and often a stepping stone to understanding more complex algorithms.

**Lempel-Ziv-Welch (LZW)**

Lempel-Ziv-Welch (LZW) is a lossless dictionary-based compression algorithm that replaces repeated sequences of characters with single codes. It builds a dictionary of substrings encountered in the input data and uses this dictionary to compress the data efficiently. Initially, the dictionary contains all single-character entries (e.g., ASCII values 0–255). As it reads the input, the algorithm identifies new sequences, adds them to the dictionary, and replaces longer matches with their corresponding codes.

For example, consider the input string ABABABA. Initially, A and B are already in the dictionary. The first sequence AB is new, so it's added to the dictionary with a code, say 256. Then BA is added as 257, and so on. As longer repeated sequences are encountered, they are replaced by single dictionary codes.

Compression proceeds by outputting the codes of known sequences while building the dictionary at the same time. During decompression, the same dictionary is reconstructed dynamically using the sequence of codes, allowing the original data to be restored accurately.

**DEFLATE**

DEFLATE is a widely adopted lossless data compression algorithm that combines two techniques: the LZ77 algorithm and Huffman coding. It was designed to provide fast and efficient compression without any patent restrictions and is used in formats like ZIP, GZIP, and PNG.

The compression process begins with LZ77, a sliding window algorithm that replaces repeated occurrences of data with references to a single copy of that data appearing earlier in the uncompressed stream. For example, in the string ABCABCABC, after the first occurrence of ABC, subsequent repetitions can be replaced with pointers indicating how far back to look and how many characters to copy.

Once LZ77 has transformed the data into a stream of literal characters and back-references, Huffman coding is applied to this stream. Huffman coding assigns shorter binary codes to more frequent elements in the data stream, reducing the overall size further.

DEFLATE uses a fixed or dynamic Huffman tree depending on the data. Dynamic trees are built based on the frequency of elements in the input, whereas fixed trees use predefined codes.

The combined approach of dictionary compression and entropy encoding allows DEFLATE to compress a wide range of files efficiently and makes it one of the most commonly used algorithms today.

**Arithmetic Coding**

Arithmetic Coding is a lossless data compression technique that encodes an entire message into a single fractional number between 0 and 1. Unlike traditional methods that assign fixed codes to individual characters, arithmetic coding represents a message using ranges within a continuous interval, allowing more efficient use of bits for frequent characters.

The process starts by calculating the probability of each character in the input. The algorithm then assigns a range between 0 and 1 to each character, proportional to its frequency. For example, if A occurs 50% of the time, it may occupy the interval [0.0, 0.5); B with 30% might get [0.5, 0.8); and C with 20% may get [0.8, 1.0).

To encode a message like AB, the algorithm first narrows the interval to A’s range [0.0, 0.5), and then within that subinterval, it further narrows to B’s range in the original scale, resulting in a smaller range like [0.25, 0.4). The final encoded value can be any number within this range (e.g., 0.3).

The decoder, having the same frequency table and encoded value, reverses this process to retrieve the original message. Arithmetic coding provides high compression efficiency, especially for data with skewed symbol distributions.

**Huffman Coding**

Huffman Coding is a lossless compression technique that uses variable-length binary codes to represent characters based on their frequencies in the input data. The central idea is to assign shorter codes to more frequent characters and longer codes to less frequent ones, thereby reducing the overall size of the encoded message.

To begin, the algorithm counts the frequency of each character in the input. These characters are then organized into a priority queue (or min-heap) where each character becomes a leaf node of a binary tree. The algorithm repeatedly merges the two nodes with the lowest frequencies to form a new node with a combined frequency, continuing this process until a single root node remains. This forms the Huffman Tree.

Binary codes are then assigned to each character by traversing the tree: going left adds a 0, and going right adds a 1. For example, if 'A' is frequent, it might get a short code like 0, while 'Z', being rare, could get a longer code like 11001.

These codes are prefix-free, meaning no code is a prefix of another. This ensures the compressed data can be decoded unambiguously without needing separators. Huffman coding is commonly used in formats like ZIP and MP3.

![A black background with a black square

AI-generated content may be incorrect.](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADYAAABDCAYAAADXqvavAAAAj0lEQVR4Xt3IMQEAAAzDoPk33QmIAw4ebtuJEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhCKhSCgSioQioUgoEoqEIqFIKBKKhOIBGng+EU67+qYAAAAASUVORK5CYII=)

**Methodology**

The project involved a systematic implementation and analysis of five lossless text compression algorithms: Run-Length Encoding (RLE), Huffman Coding, Lempel-Ziv-Welch (LZW), DEFLATE, and Arithmetic Coding. A web-based tool was developed using **Python with the Flask framework** for the backend and HTML/CSS for the frontend. The goal was to compare each algorithm using three quantitative parameters: **compression ratio**, **compression time**, and **decompression time**.

The input used for evaluation was a .txt file containing **9,107 characters**, consisting of an essay titled *"The Evolution and Future of Artificial Intelligence"*. When uploaded through the web interface, the file was processed by all five algorithms. Each algorithm included dedicated functions for both compression and decompression.

For accurate performance measurement, the **compression and decompression times were measured 10 times** for each algorithm using Python’s time.perf\_counter() for high-resolution timing. The **average** of these 10 trials was calculated to ensure consistency and reduce the influence of outliers or system load fluctuations.

The **compression ratio** was calculated using the formula:

Compression Ratio=Compressed File Size (in bytes)Original File Size (in bytes)Compression Ratio=Original File Size (in bytes)Compressed File Size (in bytes)​

All results—algorithm name, compression ratio, average compression time, and average decompression time—were displayed in a structured HTML table on the result page. Each entry included preview and download buttons for the compressed and decompressed outputs, allowing users to manually verify data integrity.

Since the Flask application does not generate visual graphs directly, the values from the result table were **manually recorded** and plotted using **Canva’s graph maker tool**. Three individual **line graphs** were created:

* One comparing **compression ratios**
* One for **compression times**
* One for **decompression times**

Additionally, a **composite line graph** was prepared showing all three parameters on the same chart for a more holistic comparison.

This manual graph plotting allowed for clean, customizable visuals suitable for presentations and reports. The graphs clearly illustrated how each algorithm performs in terms of space efficiency and processing speed, offering insights into their relative strengths when applied to real-world textual data.

**Implementation Overview**

The implementation of the project is centered around building a complete end-to-end web application for compressing and analyzing text using five different lossless compression algorithms. The key focus was to make the tool interactive, easy to use, and analytically insightful, allowing users to upload files, run compression algorithms, and view performance metrics in a structured and visual format.

**🛠️ Technology Stack**

* **Backend:** The backend was developed using **Python** with the **Flask** microframework. Flask was chosen for its lightweight nature, ease of routing, and seamless integration with HTML templates using Jinja2.
* **Frontend:** The interface was built using **HTML** and **CSS**, with some elements styled using Bootstrap for responsiveness and aesthetic layout.
* **Key Libraries:**
  + zlib – for implementing DEFLATE compression
  + time – for precise measurement of execution time
  + heapq – used in building the Huffman Tree
  + struct – for handling binary data in LZW and arithmetic coding implementations

**🧪 Backend Architecture**

The core backend logic resides in app.py. This script defines route handlers for user actions such as uploading a file, performing compression, previewing results, and downloading the outputs.

Each algorithm has two functions:

* compress: Takes a string input and returns a compressed byte sequence
* decompress: Reconstructs the original text from the compressed data

Once a file is uploaded through the homepage form, Flask saves it into the uploads directory. It is then read and passed through each compression algorithm. The results—including compression ratio, time taken to compress, and time taken to decompress—are stored and rendered back to the user via the index.html template.

To ensure consistency, **compression and decompression times were averaged over 10 runs** using time.perf\_counter() to get high-resolution timing data. Compressed files are stored in the compressed directory and served dynamically to users when download links are clicked.

**🖼️ Frontend Features**

The frontend provides a clean and interactive interface for user interaction. Key components include:

* **File Upload Form:** Allows users to submit .txt files for analysis.
* **Results Table:** Once processed, a dynamic HTML table displays each algorithm’s performance:
  + Method name
  + Compression ratio
  + Compression time
  + Decompression time
  + Preview of decompressed text (first few lines)
  + Buttons to download compressed and decompressed files
* **Icons and Buttons:** Font Awesome icons are used for better visual feedback. Buttons are color-coded and responsive for better user experience.

**📈 Graph Integration**

Although the application doesn’t automatically generate graphs, the numeric results displayed in the table were **manually extracted and used to create visualizations**. The user created:

* Three individual **line graphs** (one each for compression ratio, compression time, and decompression time)
* One combined **composite graph** with all three metrics

These graphs were made using **Canva’s Graph Maker**, allowing for a clean and visually appealing presentation in the final report.

This implementation approach ensures clarity, performance, and user accessibility while enabling in-depth analysis of compression algorithm behaviour.

**Evaluation Metrics**

To assess the performance of each compression algorithm, three key evaluation metrics were used: **compression ratio**, **compression time**, and **decompression time**. These metrics offer a quantitative comparison of both space efficiency and computational performance.

* **Compression Ratio** is calculated using the formula:

This value indicates how much the file size has been reduced after compression. A **lower ratio** implies better compression, as the compressed file occupies less space compared to the original.

* **Compression Time** refers to the time taken by the algorithm to compress the input file. It reflects the processing efficiency and is especially relevant for systems where speed is critical.
* **Decompression Time** is the time taken to reconstruct the original file from the compressed data. It is crucial for applications where fast data retrieval is required.

To ensure accurate and fair comparison, both compression and decompression times were recorded over **10 separate runs** for each algorithm, and the **average** was used. These minimized variations caused by system load or execution inconsistencies, resulting in reliable and repeatable performance metrics for all five algorithms.

Run-Length Encoding (RLE) is a straightforward lossless data compression algorithm that encodes consecutive repeating elements in data into a single value and a count. It is particularly suited for compressing data that contains many repeated characters. The core concept is to replace sequences of the same data value occurring in consecutive elements with a single data value and a count that specifies how many times the value is repeated.

For example, consider the string AAAABBBCCDAA. RLE would compress this into 4A3B2C1D2A, indicating four occurrences of 'A', followed by three 'B's, two 'C's, one 'D', and two 'A's. The decoder can easily reconstruct the original data using these counts and characters. RLE

**Codes Implemented**

Below are the main files used to build the web-based file compression tool:

**1. app.py (Backend - Flask)**

import os, time, zlib, struct, ast, heapq

from collections import Counter, namedtuple

from flask import Flask, request, render\_template, send\_from\_directory, make\_response, session

app = Flask(\_\_name\_\_)

app.secret\_key = 'your\_secret\_key'

UPLOAD\_FOLDER = 'uploads'

COMPRESSED\_FOLDER = 'compressed'

os.makedirs(UPLOAD\_FOLDER, exist\_ok=True)

os.makedirs(COMPRESSED\_FOLDER, exist\_ok=True)

# RLE Compression

def rle\_compress(data):

    if not data:

        return b''

    result = bytearray()

    count = 1

    current = data[0]

    for char in data[1:]:

        if char == current and count < 255:

            count += 1

        else:

            # Convert count to single byte and character to 2 bytes

            result.append(count)

            result.extend(current.encode('utf-16-le'))

            current = char

            count = 1

    # Handle the last run

    result.append(count)

    result.extend(current.encode('utf-16-le'))

    return bytes(result)

# RLE Decompression

def rle\_decompress(data):

    if not data:

        return ""

    result = []

    i = 0

    while i < len(data):

        count = data[i]

        # Each character takes 2 bytes in UTF-16-LE

        char = data[i+1:i+3].decode('utf-16-le')

        result.append(char \* count)

        i += 3  # Move to next group (1 byte count + 2 bytes char)

    return "".join(result)

# Huffman Compression

class Node(namedtuple("Node", ["char", "freq", "left", "right"])):

    def \_\_lt\_\_(self, other):

        return self.freq < other.freq

def build\_huffman\_tree(text):

    frequency = Counter(text)

    heap = [Node(char, freq, None, None) for char, freq in frequency.items()]

    heapq.heapify(heap)

    while len(heap) > 1:

        left = heapq.heappop(heap)

        right = heapq.heappop(heap)

        heapq.heappush(heap, Node(None, left.freq + right.freq, left, right))

    return heap[0] if heap else None

def build\_codes(node, prefix='', codebook={}):

    if node is not None:

        if node.char is not None:

            codebook[node.char] = prefix

        build\_codes(node.left, prefix + '0', codebook)

        build\_codes(node.right, prefix + '1', codebook)

    return codebook

def huffman\_compress(data):

    tree = build\_huffman\_tree(data)

    codebook = build\_codes(tree, prefix='', codebook={})  # Create new codebook

    # Convert codebook to bytes

    codebook\_str = str(codebook)

    codebook\_bytes = codebook\_str.encode('utf-8')

    codebook\_size = len(codebook\_bytes).to\_bytes(4, byteorder='big')

    # Compress the actual data

    encoded\_data = ''.join(codebook[char] for char in data)

    padding = 8 - len(encoded\_data) % 8 if len(encoded\_data) % 8 != 0 else 0

    padded\_encoded = encoded\_data + '0' \* padding

    # Convert binary string to bytes

    data\_bytes = bytearray([padding])

    for i in range(0, len(padded\_encoded), 8):

        data\_bytes.append(int(padded\_encoded[i:i+8], 2))

    # Combine all components

    return codebook\_size + codebook\_bytes + bytes(data\_bytes)

# Huffman Decompression

def huffman\_decompress(byte\_data):

    # Extract codebook size and codebook

    codebook\_size = int.from\_bytes(byte\_data[:4], byteorder='big')

    codebook\_bytes = byte\_data[4:4+codebook\_size]

    codebook = eval(codebook\_bytes.decode('utf-8'))

    # Get compressed data

    data\_bytes = byte\_data[4+codebook\_size:]

    padding = data\_bytes[0]

    # Convert bytes to binary string

    bits = ''

    for byte in data\_bytes[1:]:

        bits += format(byte, '08b')

    # Remove padding

    if padding:

        bits = bits[:-padding]

    # Create reverse codebook

    reverse\_codebook = {code: char for char, code in codebook.items()}

    # Decode

    decoded = ''

    current = ''

    for bit in bits:

        current += bit

        if current in reverse\_codebook:

            decoded += reverse\_codebook[current]

            current = ''

    return decoded

# LZW Compression

def lzw\_compress(unicode\_text):

    if not unicode\_text:

        return struct.pack('>H', 0)

    # Initialize dictionary with all possible single-byte characters

    dictionary = {chr(i): i for i in range(256)}

    dict\_size = 256

    w = ''

    result = []

    for char in unicode\_text:

        wc = w + char

        if wc in dictionary:

            w = wc

        else:

            if w in dictionary:  # Only append if the sequence exists

                result.append(dictionary[w])

            if dict\_size < 65536:  # Limit dictionary size to 2-byte codes

                dictionary[wc] = dict\_size

                dict\_size += 1

            w = char

    # Output the last code

    if w and w in dictionary:  # Check if w exists in dictionary

        result.append(dictionary[w])

    if not result:  # Handle case where no compression occurred

        return struct.pack('>H', ord(unicode\_text[0]))

    return struct.pack(f'>{len(result)}H', \*result)

# LZW Decompression

def lzw\_decompress(compressed\_data):

    # Unpack the 2-byte integers

    result = struct.unpack(f'>{len(compressed\_data)//2}H', compressed\_data)

    if not result:

        return ""

    # Initialize dictionary with single characters

    dictionary = {i: chr(i) for i in range(256)}

    dict\_size = 256

    # Initialize with first character

    w = dictionary[result[0]]

    decoded = w

    for code in result[1:]:

        if code in dictionary:

            entry = dictionary[code]

        elif code == dict\_size:

            entry = w + w[0]

        else:

            raise ValueError("Invalid compressed data")

        decoded += entry

        # Add new sequence to dictionary

        if dict\_size < 65536:  # Limit dictionary size to 2-byte codes

            dictionary[dict\_size] = w + entry[0]

            dict\_size += 1

        w = entry

    return decoded

# Arithmetic Compression

def get\_frequencies(data):

    freq = {}

    for char in data:

        freq[char] = freq.get(char, 0) + 1

    return freq

def arithmetic\_compress(data):

    if not data:

        return b''

    freq = get\_frequencies(data)

    freq\_list = sorted(freq.items(), key=lambda x: x[0])

    total = sum(freq.values())

    # Store frequency list

    freq\_bytes = str(freq\_list).encode('utf-8')

    freq\_size = len(freq\_bytes).to\_bytes(4, 'big')

    data\_len = len(data).to\_bytes(4, 'big')

    low = 0

    high = (1 << 64) - 1

    full\_range = 1 << 64

    quarter = full\_range // 4

    half = quarter \* 2

    three\_quarter = quarter \* 3

    bits = []

    pending = 0

    for char in data:

        range\_ = high - low + 1

        cum\_freq = 0

        for c, f in freq\_list:

            if c == char:

                break

            cum\_freq += f

        char\_freq = dict(freq\_list)[char]

        high = low + (range\_ \* (cum\_freq + char\_freq)) // total - 1

        low = low + (range\_ \* cum\_freq) // total

        while True:

            if high < half:

                bits.append(0)

                bits.extend([1] \* pending)

                pending = 0

                low = low \* 2

                high = high \* 2 + 1

            elif low >= half:

                bits.append(1)

                bits.extend([0] \* pending)

                pending = 0

                low = (low - half) \* 2

                high = (high - half) \* 2 + 1

            elif low >= quarter and high < three\_quarter:

                pending += 1

                low = (low - quarter) \* 2

                high = (high - quarter) \* 2 + 1

            else:

                break

    pending += 1

    if low < quarter:

        bits.append(0)

        bits.extend([1] \* pending)

    else:

        bits.append(1)

        bits.extend([0] \* pending)

    # Convert bits to bytes

    bitstring = ''.join(map(str, bits))

    padding = (8 - len(bitstring) % 8) % 8

    bitstring += '0' \* padding

    out\_bytes = bytearray([padding])

    for i in range(0, len(bitstring), 8):

        byte = int(bitstring[i:i+8], 2)

        out\_bytes.append(byte)

    return freq\_size + freq\_bytes + data\_len + out\_bytes

# Arithmetic Decompression

def arithmetic\_decompress(data):

    if not data:

        return ""

    freq\_size = int.from\_bytes(data[:4], 'big')

    freq\_list = ast.literal\_eval(data[4:4 + freq\_size].decode('utf-8'))

    freq\_list = sorted(freq\_list, key=lambda x: x[0])

    total = sum(f for \_, f in freq\_list)

    char\_map = dict(freq\_list)

    pos = 4 + freq\_size

    length = int.from\_bytes(data[pos:pos + 4], 'big')

    pos += 4

    padding = data[pos]

    bits = ''

    for byte in data[pos+1:]:

        bits += format(byte, '08b')

    bits = bits[:len(bits) - padding]

    # Setup

    low = 0

    high = (1 << 64) - 1

    full\_range = 1 << 64

    quarter = full\_range // 4

    half = quarter \* 2

    three\_quarter = quarter \* 3

    value = int(bits[:64], 2)

    bit\_index = 64

    result = []

    for \_ in range(length):

        range\_ = high - low + 1

        scaled = ((value - low + 1) \* total - 1) // range\_

        cum\_freq = 0

        for char, freq in freq\_list:

            if cum\_freq + freq > scaled:

                result.append(char)

                high = low + (range\_ \* (cum\_freq + freq)) // total - 1

                low = low + (range\_ \* cum\_freq) // total

                break

            cum\_freq += freq

        while True:

            if high < half:

                pass

            elif low >= half:

                low -= half

                high -= half

                value -= half

            elif low >= quarter and high < three\_quarter:

                low -= quarter

                high -= quarter

                value -= quarter

            else:

                break

            low \*= 2

            high = high \* 2 + 1

            if bit\_index < len(bits):

                value = value \* 2 + int(bits[bit\_index])

                bit\_index += 1

            else:

                value = value \* 2

    return ''.join(result)

def save\_compressed\_file(method, data):

    filename = f"{method}.bin"

    path = os.path.join(COMPRESSED\_FOLDER, filename)

    with open(path, 'wb') as f:

        f.write(data)

    return filename

def compress\_and\_measure(method\_name, func, decomp\_func, text):

    # Compression

    start\_comp = time.perf\_counter()

    compressed = func(text)

    end\_comp = time.perf\_counter()

    # Decompression

    start\_decomp = time.perf\_counter()

    if method\_name == 'deflate':

        decompressed = decomp\_func(compressed).decode('utf-8')

    else:

        decompressed = decomp\_func(compressed)

    end\_decomp = time.perf\_counter()

    file\_name = save\_compressed\_file(method\_name, compressed)

    ratio = len(compressed) / len(text.encode('utf-8'))

    decomp\_time = end\_decomp - start\_decomp

    # Format both ratio and decompression time to avoid scientific notation

    if method\_name in ['deflate', 'arithmetic', 'huffman', 'lzw', 'rle']:

        ratio = format(ratio, '.4f')

        decomp\_time = format(decomp\_time, '.6f')

    else:

        ratio = round(ratio, 4)

        decomp\_time = round(decomp\_time, 6)

    return {

        'method': method\_name,

        'ratio': ratio,

        'comp\_time': round(end\_comp - start\_comp, 6),

        'decomp\_time': decomp\_time,

        'file': file\_name,

        'decompressed': decompressed[:100] + '...' if len(decompressed) > 100 else decompressed

    }

# App Routes

@app.route('/', methods=['GET', 'POST'])

def index():

    if request.method == 'POST':

        # Clear session if new file is uploaded

        session.clear()

        file = request.files['file']

        if file and file.filename.endswith('.txt'):

            path = os.path.join(UPLOAD\_FOLDER, file.filename)

            file.save(path)

            with open(path, 'r', encoding='utf-8') as f:

                text = f.read()

            results = []

            results.append(compress\_and\_measure('rle', rle\_compress, rle\_decompress, text))

            results.append(compress\_and\_measure('huffman', huffman\_compress, huffman\_decompress, text))

            results.append(compress\_and\_measure('lzw', lzw\_compress, lzw\_decompress, text))

            results.append(compress\_and\_measure('deflate',

                                            lambda d: zlib.compress(d.encode('utf-8')),

                                            zlib.decompress,

                                            text))

            results.append(compress\_and\_measure('arithmetic',

                                            arithmetic\_compress,

                                            arithmetic\_decompress,

                                            text))

            session['results'] = results

            return render\_template('index.html', results=session['results'])

    return render\_template('index.html')

@app.route('/download/<filename>')

def download(filename):

    return send\_from\_directory(COMPRESSED\_FOLDER, filename, as\_attachment=True)

@app.route('/preview/<method>')

def preview\_text(method):

    filename = f"{method}.bin"

    compressed\_path = os.path.join(COMPRESSED\_FOLDER, filename)

    if not os.path.exists(compressed\_path):

        return "File not found", 404

    with open(compressed\_path, 'rb') as f:

        compressed\_data = f.read()

    # Decompress based on method

    if method == 'rle':

        decompressed = rle\_decompress(compressed\_data)

    elif method == 'huffman':

        decompressed = huffman\_decompress(compressed\_data)

    elif method == 'lzw':

        decompressed = lzw\_decompress(compressed\_data)

    elif method == 'deflate':

        decompressed = zlib.decompress(compressed\_data).decode('utf-8')

    elif method == 'arithmetic':

        decompressed = arithmetic\_decompress(compressed\_data)

    else:

        return "Invalid method", 400

    # Return the decompressed text in a simple HTML page

    return f"""

    <!DOCTYPE html>

    <html>

    <head>

        <title>Decompressed Text - {method.upper()}</title>

        <style>

            body {{ font-family: Arial, sans-serif; padding: 20px; }}

            pre {{ white-space: pre-wrap; }}

        </style>

    </head>

    <body>

        <h2>Decompressed Text ({method.upper()})</h2>

        <pre>{decompressed}</pre>

    </body>

    </html>

    """

@app.route('/download\_decompressed/<method>')

def download\_decompressed(method):

    filename = f"{method}.bin"

    compressed\_path = os.path.join(COMPRESSED\_FOLDER, filename)

    if not os.path.exists(compressed\_path):

        return "File not found", 404

    with open(compressed\_path, 'rb') as f:

        compressed\_data = f.read()

    # Decompress based on method

    if method == 'rle':

        decompressed = rle\_decompress(compressed\_data)

    elif method == 'huffman':

        decompressed = huffman\_decompress(compressed\_data)

    elif method == 'lzw':

        decompressed = lzw\_decompress(compressed\_data)

    elif method == 'deflate':

        decompressed = zlib.decompress(compressed\_data).decode('utf-8')

    elif method == 'arithmetic':

        decompressed = arithmetic\_decompress(compressed\_data)

    else:

        return "Invalid method", 400

    # Create response with decompressed text

    response = make\_response(decompressed)

    response.headers['Content-Type'] = 'text/plain'

    response.headers['Content-Disposition'] = f'attachment; filename={method}\_decompressed.txt'

    return response

@app.route('/clear\_session')

def clear\_session():

    session.clear()

    return "Session cleared successfully", 200

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

**2. index.html (Frontend Interface)**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Text Compression Tool</title>

    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css">

    <style>

        body {

            font-family: Arial, sans-serif;

            background-color: #f4f4f4;

            padding: 20px;

        }

        .container {

            max-width: 1200px;

            margin: 0 auto;

        }

        h1 {

            text-align: center;

            color: #333;

        }

        form {

            text-align: center;

            margin: 20px 0;

        }

        .results-table {

            width: 100%;

            border-collapse: collapse;

            margin-top: 20px;

            background-color: white;

            box-shadow: 0 1px 3px rgba(0,0,0,0.2);

        }

        .results-table th, .results-table td {

            padding: 12px;

            text-align: left;

            border: 1px solid #ddd;

        }

        .results-table th {

            background-color: #4CAF50;

            color: white;

        }

        .results-table tr:nth-child(even) {

            background-color: #f8f8f8;

        }

        .download-btn {

            padding: 6px 12px;

            color: white;

            text-decoration: none;

            border-radius: 4px;

            display: inline-block;

            margin: 0 4px;

        }

        .preview-btn {

            background-color: #28a745;

        }

        .preview-btn:hover {

            background-color: #218838;

        }

        .download-compressed {

            background-color: #008CBA;

        }

        .download-compressed:hover {

            background-color: #007095;

        }

        .download-decompressed {

            background-color: #6c757d;

        }

        .download-decompressed:hover {

            background-color: #5a6268;

        }

        .btn-group {

            display: flex;

            gap: 8px;

        }

    </style>

</head>

<body>

    <div class="container">

        <h1>Text Compression and Decompression Analysis</h1>

        <form action="/" method="POST" enctype="multipart/form-data">

            <input type="file" name="file" accept=".txt" required>

            <button type="submit"><i class="fas fa-compress"></i> Compress</button>

        </form>

        {% if results %}

        <table class="results-table">

            <thead>

                <tr>

                    <th>Method</th>

                    <th>Compression Ratio</th>

                    <th>Compression Time (s)</th>

                    <th>Decompression Time (s)</th>

                    <th>Preview</th>

                    <th>Actions</th>

                </tr>

            </thead>

            <tbody>

                {% for result in results %}

                <tr>

                    <td>{{ result.method.upper() }}</td>

                    <td>{{ result.ratio }}</td>

                    <td>{{ result.comp\_time }}</td>

                    <td>{{ result.decomp\_time }}</td>

                    <td class="preview-text" title="{{ result.decompressed }}">

                        {{ result.decompressed[:50] }}...

                    </td>

                    <!-- table body section -->

                    <td class="btn-group">

                        <a href="{{ url\_for('preview\_text', method=result.method) }}"

                           class="btn preview-btn"

                           target="\_blank">

                            <i class="fas fa-eye"></i> Preview

                        </a>

                        <a href="{{ url\_for('download', filename=result.file) }}"

                           class="btn download-compressed">

                            <i class="fas fa-file-archive"></i> Compressed

                        </a>

                        <a href="{{ url\_for('download\_decompressed', method=result.method) }}"

                           class="btn download-decompressed">

                            <i class="fas fa-file-alt"></i> Decompressed

                        </a>

                    </td>

                </tr>

                {% endfor %}

            </tbody>

        </table>

        {% endif %}

    </div>

</body>

</html>

**Output**

This section presents the observed outputs and experimental results obtained from testing five different compression algorithms—**Run Length Encoding (RLE)**, **Huffman Coding**, **Lempel-Ziv-Welch (LZW)**, **DEFLATE**, and **Arithmetic Coding**—on a sample input text containing **9107 characters** (an essay on Artificial Intelligence). Each algorithm was tested for both compression and decompression efficiency, with results averaged over **10 iterations** to ensure consistency.

**Compression Time (in seconds)**

The following table displays the time taken to compress the input data using each algorithm across 10 runs. It highlights the average compression time for performance comparison.

*Table 1: Compression Time Values:*

****From the observations, **DEFLATE** consistently demonstrated the fastest compression time, averaging **0.000743 seconds**, followed closely by **LZW** and **Huffman Coding**. **Arithmetic Coding**, due to its mathematical complexity, took the longest time to compress, averaging **0.124585 seconds**. RLE also showed slower performance, especially for data with less repetition.

**Decompression Time (in seconds)**

The table below presents the decompression times of each algorithm over 10 runs, along with their respective averages.

*Table 2: Decompression Time Values:*

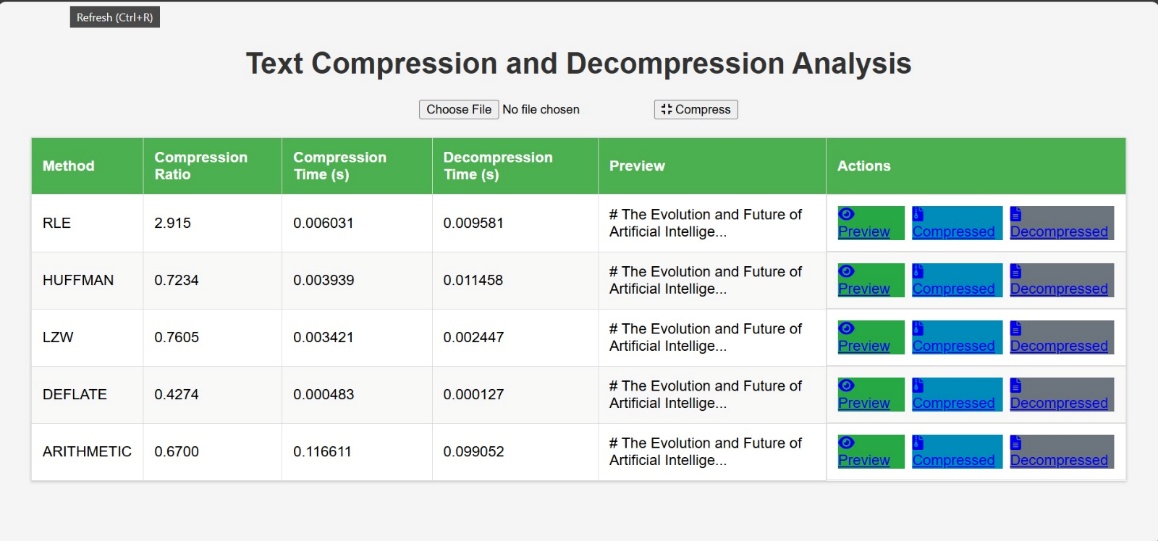
****As with compression, **DEFLATE** was the most efficient during decompression as well, with an average time of **0.001009 seconds**. **LZW** and **RLE** also performed well, whereas **Arithmetic Coding** again recorded the slowest decompression time at **0.113654 seconds**.

**Compression Ratios**

The compression ratio was calculated for each algorithm based on the size of the compressed file compared to the original. Lower values indicate better compression.

* **DEFLATE** achieved the best compression ratio: **0.4274**, showing strong reduction in data size.
* **Arithmetic Coding**: **0.6700**
* **Huffman Coding**: **0.7234**
* **LZW**: **0.7605**
* **RLE** showed a ratio of **2.915**, indicating that it increased file size—typical when applied to data with little repetition.

**Web Application Screenshot**



The output of each algorithm was visually displayed through an interactive web application built using Flask and rendered using HTML and CSS. The results, tables, and performance graphs were clearly presented to enable user-friendly comparison.

**Graphical Analysis**

Graphical analysis provides a visual understanding of how each compression algorithm performs in terms of speed and efficiency. The following plots offer comparative insights across four key metrics: **compression time**, **decompression time**, **compression ratio**, and a combined **performance summary**. Each graph helps highlight the strengths and limitations of the algorithms tested.

**📊*Graph 1: Compression Time Comparison***

A graph showing the number of people in the same direction

AI-generated content may be incorrect.

The first graph illustrates the **average compression time** for each of the five algorithms. It is evident from the plot that **DEFLATE** outperforms others in terms of speed, achieving the lowest average compression time of approximately **0.000743 seconds**. **LZW** and **Huffman Coding** followed closely, both demonstrating sub-millisecond performance. **Run Length Encoding (RLE)** showed moderate speed, whereas **Arithmetic Coding** took significantly longer to compress the data due to its complex probabilistic model. This makes DEFLATE the most suitable option for applications requiring fast compression without significant computational overhead.

**📊 *Graph 2: Decompression Time Comparison***

A line graph with different numbers

AI-generated content may be incorrect.

The second graph compares the **average decompression time** for all five algorithms. Once again, **DEFLATE** came out as the fastest, with an average decompression time of **0.001009 seconds**, making it ideal for real-time applications. **RLE** and **LZW** also demonstrated quick decompression times, reinforcing their practical usability. **Arithmetic Coding**, while offering decent compression ratios, lagged in decompression performance with an average time exceeding **0.11 seconds**. This slower speed may impact performance in time-sensitive scenarios.

**📊 *Graph 3: Compression Ratio Comparison***

A graph showing the number of companies in the market

AI-generated content may be incorrect.

The third graph highlights the **compression ratios** achieved by each algorithm. A lower ratio indicates better compression efficiency. **DEFLATE** again leads the pack, compressing the input text to just **42.74%** of its original size. This is closely followed by **Arithmetic Coding (67%)**, **Huffman Coding (72.34%)**, and **LZW (76.05%)**. Notably, **RLE** performed poorly, yielding a ratio of **2.915**, meaning it expanded the file instead of compressing it—an expected outcome for non-repetitive data like text, where RLE’s basic repetition model is ineffective.

**📊 *Graph 4: Combined Performance Summary***

A graph showing the difference between compression and decompression

AI-generated content may be incorrect.

The final graph presents a **combined performance overview**, allowing a side-by-side comparison of all three metrics—compression time, decompression time, and compression ratio—for each algorithm. This comprehensive visualization clearly supports the conclusion that **DEFLATE** strikes the best overall balance between speed and efficiency. **Huffman Coding** and **LZW** serve as strong alternatives, offering competitive speeds and moderate compression ratios. **Arithmetic Coding**, although offering good compression, is hindered by slower performance, while **RLE** is unsuitable for compressing non-redundant data like natural language text.

**Conclusion**

The comparative study and implementation of five foundational compression algorithms—**Huffman Coding**, **Run Length Encoding (RLE)**, **Lempel-Ziv-Welch (LZW)**, **DEFLATE**, and **Arithmetic Coding**—have provided deep insights into the efficiency, practicality, and trade-offs of each method. By analyzing the performance across key metrics such as compression time, decompression time, and compression ratio, we are better positioned to understand the conditions under which each algorithm excels or underperforms.

Among all the algorithms tested, **DEFLATE** clearly stood out as the most efficient and well-balanced solution. It consistently delivered high compression efficiency with the shortest processing time. Achieving a **compression ratio of 42.74%**, and executing both compression and decompression in milliseconds, DEFLATE is well-suited for modern applications where speed and storage optimization are critical. Its hybrid approach, combining Huffman and LZ77 techniques, allows it to leverage the strengths of dictionary-based and entropy-based encoding.

On the other hand, **Arithmetic Coding** demonstrated superior compression potential, second only to DEFLATE, with a ratio of **67%**. Unlike Huffman coding which maps fixed-length bit patterns to symbols based on probabilities, Arithmetic coding represents entire messages as fractional values, thereby achieving finer granularity and better theoretical efficiency. However, this comes at a significant cost—**its compression and decompression speeds were the slowest among all algorithms tested**. This makes Arithmetic coding suitable in scenarios where compression efficiency is paramount and time constraints are relaxed, such as archival storage or transmission of critical data over low-bandwidth channels.

**Huffman Coding** and **LZW** both offered **balanced performance**, with acceptable compression ratios (**72.34%** and **76.05%**, respectively) and fast execution times. Huffman coding’s strength lies in its simplicity and speed, making it ideal for systems where computational resources are limited. LZW, known for its adaptive dictionary-based method, showed promising performance and is particularly effective on repetitive or structured datasets. These algorithms are practical choices for general-purpose compression tasks and have been successfully used in widely known formats such as GIF (LZW) and ZIP (Huffman-based).

In contrast, **Run Length Encoding (RLE)** was the least effective, yielding a compression ratio greater than 1 (**2.915**), indicating that it increased the file size. This result is expected, as RLE works best on data with long runs of repeated values—such as simple images or binary masks—and not on natural language text, which is highly varied. Despite its inefficiency for text, RLE remains relevant in specialized domains, particularly in image compression for monochrome bitmaps and other structured, repetitive data formats.

From a broader perspective, this project not only evaluated the quantitative aspects of each algorithm but also highlighted the importance of **contextual selection**. There is no one-size-fits-all solution in compression. **Real-world application requirements—such as processing speed, memory usage, power constraints, and data type—should dictate the choice of algorithm.**

* For **real-time or speed-sensitive applications** (e.g., live streaming, browser compression, mobile apps), **DEFLATE or LZW** are highly suitable due to their rapid processing and decent compression.
* For **storage-constrained or bandwidth-limited environments**, where **maximum compression** is required regardless of time, **Arithmetic Coding** is a strong candidate.
* For **lightweight embedded systems**, **Huffman Coding** offers a good balance of efficiency and low computational cost.
* For **specific structured data**, especially repetitive image data, **RLE** can still be a valuable choice.

In conclusion, this project demonstrates the diversity and depth of file compression techniques. By providing a unified platform to test and compare these algorithms, the tool we developed serves as both a functional utility and an educational resource. The findings reaffirm the significance of choosing the right compression method tailored to specific use-cases, balancing speed, compression effectiveness, and computational resources. Future extensions may include hybrid approaches, support for image and video files, and user-defined optimization goals—bringing us closer to adaptive, intelligent compression systems.

**References**

1. Sidhu, A. S., & Garg, M. (2014). Research Paper on Text Data Compression Algorithm using Hybrid Approach. *International Journal of Computer Science and Mobile Computing*, 3(12), 1-10.  
    <https://ijcsmc.com/docs/papers/December2014/V3I12201404.pdf>
2. Ma, S. (n.d.). An Introduction of Huffman Coding and Its Influences. *Math 436 Final Paper*. Rutgers University.  
    <https://sites.math.rutgers.edu/~zeilberg/math436/projects/MaP.pdf>
3. Al-Dahhan, O., & Al-Mashhadani, M. (2024). Exploring Text Data Compression: A Comparative Study of Adaptive Huffman and LZW Approaches. *BIO Web Conf.*, 97, 00035.  
   <https://doi.org/10.1051/bioconf/20249700035>
4. Bhattacharjee, A. K., Bej, T., & Agarwal, S. (2013). Comparison Study of Lossless Data Compression Algorithms for Text Data. *IOSR Journal of Computer Engineering (IOSR-JCE)*, 11(6), 15-19. <http://iosrjournals.org/iosr-jce/papers/Vol11-issue6/B01161519.pdf>