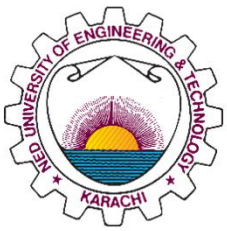

NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DATA STRUCTURES AND ALGORITHM COMPLEX ENGINEERING PROJECT:

File Compressor Algorithm And Analysis

Course Code: CS-218

Submitted to:

Dr. Urooj Ainuddin

Submitted by:

Usman Rasheed Siddiqui (CS-24038)

Table of Contents

1 Project Overview:

To develop a file compression tool that implements an appropriate compression algorithm, provides a graphical user interface (GUI), achieves effective compression, and allows analysis of time and space complexity while demonstrating the input and output of files.

2 Introduction:

2.1 Huffman Coding (Compression Algorithm):

Huffman coding is a **lossless data** compression algorithm. It assigns variable-length codes to input characters, with shorter codes for more frequent characters and longer codes for less frequent ones. Huffman compression is widely used in file compression formats due to its efficiency.

2.2 Importance of Compression:

Data compression is a critical process in modern computing for several reasons:

1. **Reduce Storage Requirements:**
   - Compressed files occupy less disk space, enabling more efficient storage of data.
   - Saves cost for servers, cloud storage, and personal devices
2. **Speed Up Data Transmission:**
   - Smaller files are faster to transmit over networks.
   - Reduces bandwidth usage and improves download/upload speeds.
3. **Supports Data Archiving and Sharing:**
   - Compressed files are easier to archive, store and share via email or cloud services.
4. **Energy Efficiency:**
   - Transmitting smaller files consumes less power, which is crucial for mobile and embedded devices.

3 Technologies Used:

This section explains the software tools and techlogies employed in the project:

1. **Django (Web Framework):**
   - Used to create the Graphical User Interface (GUI) of the compression tool.
   - Allows users to upload files, compress and decompress files, and download them easily.
   - Provides a web-based platform, making the tool accessible from any browser.
2. **Python Libraries for Compression:**
   - **Bitarray:** Efficiently stores and manipulates arrays of bits for Huffman coding.
   - **Collection (Counter):** Generates frequency tables of symbols in the input data.

- **Heapq:** Implements a min-heap to efficiently build the Huffman tree.

3. **Other Utilies:**
   - **os & time:** For handling files and measuring performance.

4 Compression Workflow

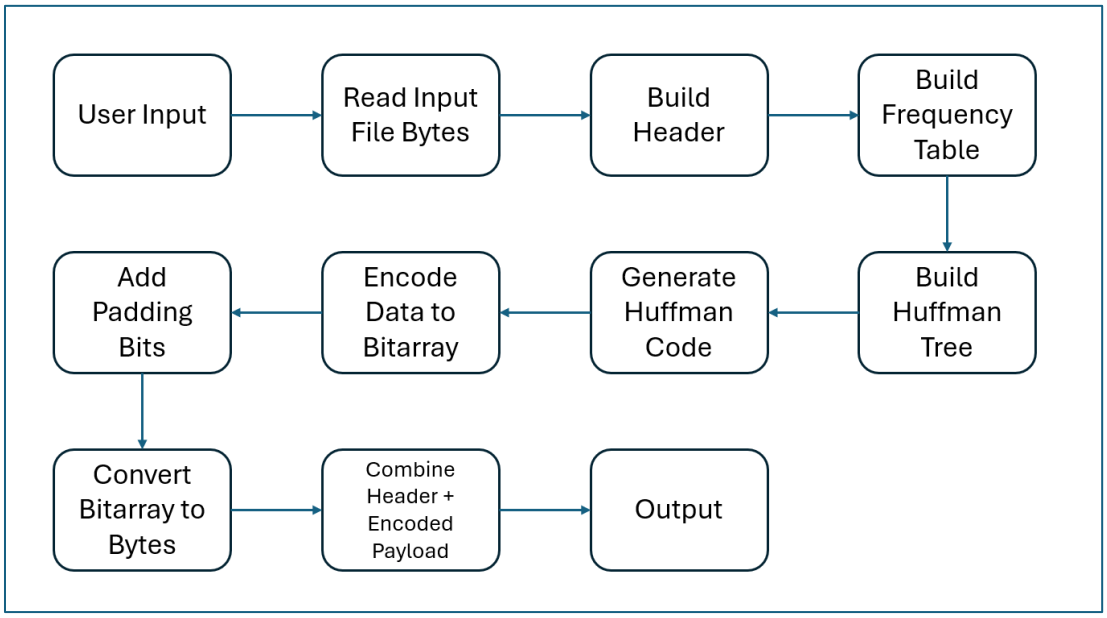

Figure 1: Compression Workflow

5 Code:

Github Link to Code:

https://github.com/Usman-Rasheed-Siddiqui/file-compressor/tree/main/filecompressor/compressor/algorithm

5.1 Library Imports

```
from collections import Counter  # For building frequency table of
symbols
from bitarray import bitarray     # Efficient bit array manipulation
import heapq                      # min-heap for Huffman tree
construction
```

5.2 Huffman Node Class

```
class Node:
    """
    Node class for Huffman Tree.
    symbol: Byte value for leaf nodes, None for internal nodes
```

```
    freq: Frequency of symbol or sum of child frequencies
    left, right: Child nodes
    """

    def __init__(self, symbol, freq, left=None, right=None):
        self.symbol = symbol
        self.freq = freq
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.freq < other.freq  # Comparison for heap operations
```

5.3 Build Huffman Tree

```
def build_huffman_tree(freq_table):
    """
    Builds a Huffman tree from a frequency table using a min-heap.
    Returns: Root node of Huffman tree
    """
    if not freq_table:
        return None
    heap = []
    for symbol, frequency in freq_table.items():
        heapq.heappush(heap, Node(symbol, frequency))  # Push leaf nodes
to heap

    while len(heap) > 1:
       node1 = heapq.heappop(heap)  # Pop two nodes with smallest
frequency
       node2 = heapq.heappop(heap)
       merged = Node(symbol=None, freq=node1.freq + node2.freq,
left=node1, right=node2)  # Merge them
       heapq.heappush(heap, merged)  # Push merged node back
    return heap[0]  # Root node
```

5.4 Generate Huffman Codes

```
def generate_huffman_code(node, current_node, code_map):
    """
    Recursively traverses Huffman tree to generate codes for each symbol.
    Stores symbol → code mapping in code_map dictionary.
    """
    if node is None:
        return
```

```
    if node.symbol is not None:
        if current_node == "":
            code_map[node.symbol] = "0"  # Edge case: only one symbol
        else:
            code_map[node.symbol] = current_node
        return

    # Recursion to the left of root node
    generate_huffman_code(node.left, current_node + "0", code_map)

    # Recursion to the right of root node
    generate_huffman_code(node.right, current_node + "1", code_map)
```

5.5 Encode Data to Bitarray

```
def encode_data_to_bitarray(data: bytes, code_map: dict) -> bytes:
    """
    Encodes input data using Huffman code map into a bitarray and packs
it into bytes.
    Returns: bytes containing the compressed data.
    """
    bits = bitarray()
    for b in data:
        bits.extend(code_map[b])  # Append Huffman code for each byte

    pad_len = (8 - len(bits) % 8) % 8  # Pad to make multiple of 8
    bits.extend("0" * pad_len)

    return bytes([pad_len]) + bits.tobytes()  # Prepend padding length
```

5.6 Compress Huffman

```
def compress_huffman(data: bytes, ext: str = "txt") -> bytes:
    """
    Compresses input data using Huffman coding.
    Returns: Bytes containing header + compressed payload.
    """
    if not data:
        return b''

    # Preparing Header
    header = bytearray(b"HUF1")

    # Building frequency table
```

```
    freq_table = Counter(data)

    # Encode Extension length and bytes
    ext_bytes = ext.encode("utf-8")
    if len(ext_bytes) > 10:
        raise ValueError("Extension too long")

    header += bytes([len(ext_bytes)]) + ext_bytes

    # Building header

    k = len(freq_table)
    header += k.to_bytes(2, 'big')
    for symbol, freq in freq_table.items():
        header += bytes([symbol]) + freq.to_bytes(8, 'big')

    # Build Payload
    root = build_huffman_tree(freq_table)
    code_map = {}
    generate_huffman_code(root, "", code_map)
    payload = encode_data_to_bitarray(data, code_map)

    return bytes(header + payload)
```

6 Libraries Used:

6.1 Bitarray Library:

The bitarray library provides an efficient way to store and manipulate arrays of bits (boolean values). Unlike standard Python lists or strings, bitarray stores bits in a compact form, using only 1 bit per boolean value instead of 8 bits (1 byte).

Functions Used:

- .***bitarray***() – Creates an empty bitarray object
  - **Time:** $O(1)$, **Space:** $O(1)$
- .***extend***() - Appends it to the array of bits
  - **Time:** $O(1)$, **Space:** $O(1)$
- .***frombytes***() - Appends all bits from a bytes object
  - **Time:** $O(m)$ for m bytes, **Space:** $O(m)$ (8m bits) ($\boldsymbol{m \approx n}$)
- .***tobytes***() - Converts the bitarray to a bytes object

- **Time:** $O(n)$, **Space:** $O(n)$ (n/8 bytes)

- $.\boldsymbol{to01}()$ - Returns a string of '0's and '1's
  - **Time:** $O(n)$, **Space:** $O(n)$ (n characters)

Advantage: Bitarray stores 1 bit per value, compared to Python bool objects which take much more memory, giving much more space savings for binary data.

6.2 Collections Library – Counter

The Counter class from collections is a dictionary subclass for counting hashable objects, useful for frequency analysis.

Function Used:

- $\boldsymbol{Counter(data)}$ **-** Creates a frequency table
  - **Time:** $\boldsymbol{O(n)}$ **-** where n = length of the data
    Iterate through each element:
    Hash the element: $\boldsymbol{O(1)}$ + Dictionary lookup: $\boldsymbol{O(1)}$ + Increment count: $\boldsymbol{O(1)}$

    Total: n iterations × $O(1)$ per iteration = $O(n)$
  - **Space:** $\boldsymbol{O(k)}$ where k = number of unique elements
    Stores one dictionary entry per unique element (for bytes, $k \leq 256$)

6.3 Heapq Library

The heapq library implements a **min-heap** using a Python list. A min-heap is a binary tree where each parent smaller than its children, allowing efficient access to the minimum element.

- **Heap Properties:**
  - Smallest element at `heap[0]`
  - **Height with k elements:** $\boldsymbol{h = (logk)}$

Functions Used:

- $.\boldsymbol{heappush(heap, item)}$ **-** Inserts an item while maintaining heap property
  - **Time:** $\boldsymbol{O(log\ k)}$ – bubble up through $\log_2$ k levels (1 comparison/swap per level)
  - **Space:** $\boldsymbol{O(1)}$ – Only stores one new element
- $.\boldsymbol{heappop(heap)}$ **-** Removes and returns the smallest element (root) from heap
  - **Time:** $\boldsymbol{O(log\ k)}$**:** bubble down through $\log_2$ k levels (compare with 2 children per level)

- **Space: $O(1)$:** Modifies heap in-place, no additional memory needed

Advantage: Heap reduces insertion/removal from O(k²) (list approach) to **O(log k)** per operation.

NOTE: In our code the list structure behaves exactly like a binary tree for the purpose of locating parents and children. But the final Huffman tree obtained is not a heap as each parent node's frequency equals to sum of children's nodes' frequencies.

7 User-Built Functions

7.1 build_huffman_tree(freq_table)

Builds a binary Huffman tree from the frequency table.

7.1.1 Analysis

```
if not freq_table:
      return None
```

- **Time:** $O(1)$ – checks if dictionary is empty
- **Space:** $O(1)$

```
heap = []
for symbol, frequency in freq_table.items():
    heapq.heappush(heap, Node(symbol, frequency))  # Push leaf nodes to
 heap
```

- **Time:** $O(k\ log\ k)$ – k pushes, each $O(log\ k)$
- **Space:** $O(k)$ – stores k leaf nodes

```
while len(heap) > 1:
    node1 = heapq.heappop(heap)  # Pop two nodes with smallest frequency
    node2 = heapq.heappop(heap)
    merged = Node(symbol=None, freq=node1.freq + node2.freq, left=node1,
       right=node2)  # Merge them
    heapq.heappush(heap, merged)  # Push merged node back
```

- **Time:** $O(k\ log\ k)$ – runs $k-1$ times, each iteration $O(log\ k)$
- **Space:** $O(k)$ – $total\ nodes\ =\ k\ leaves\ +\ (k-1)\ internal\ nodes\ =\ 2k-1$

```
return heap[0]  # Root node
```

- **Time:** $O(1)$
- **Space:** $O(1)$

7.1.2 Total Complexity

- **Time:** $O(k\ log\ k)$ (best & worst)
- **Space:** $O(k)$ (best & worst)

7.2 generate_huffman_code(node, current_node, code_map)

Traverses the Huffman tree recursively and generates binary codes for each symbol

7.2.1 Analysis

1. **Base Condition:**

```
if node.symbol is not None:
    if current_node == "":
        code_map[node.symbol] = "0"  # Edge case: only one symbol
```

- **Time:** $O(1)$ per node
- **Space:** $O(1)$ for single assignment (special case: "0" if only one symbol)

```
else:
    code_map[node.symbol] = current_node
    return
```

- **Time:** $O(1)$ per dictionary assignment
- **Space:**
    - **Best:** $\boldsymbol{O(log\ k)}$ - stores code string of length log k
    - **Worst:** $\boldsymbol{O(k)}$ - stores code string of length k

Overall space also includes recursion stack and code_map.

2. **Recursive Condition:**

```
# Recursion to the left of root node
generate_huffman_code(node.left, current_node + "0", code_map)

# Recursion to the right of root node
generate_huffman_code(node.right, current_node + "1", code_map)
```

Recursions visit all $2k\ -\ 1$ nodes (k is the number of symbols).

- **Time:** $O(1)\ x\ k\ =\ O(k)$
- **Space:** $O(h)$ (Depth of Runtime Stack) $-\ h\ =\ height\ of\ the\ tree$

7.2.2 Total Time Complexity:

Each node processed once = $\boldsymbol{O(k)}$

7.2.3 Total Space Complexity:

- **Recursion Stack:** $O(h)$
  - **Best (balanced tree):** $O(logk)$ – Perfect Binary or Almost Perfect Binary Tree
  - **Worst (skewed tree):** $O(k)$ – All nodes arranged in the form of linked list
- **Code map:** Stores k codes, each code is $\leq h : O(k \,.\, h)$
  - **Best:** $O(k\ logk)$
  - **Worst** $O(k^2)$
- **Final Space Complexity:**
  - **Best:** $O(k\ logk)$
  - **Worst** $O(k^2)$

NOTE: For the special case of a single-symbol tree, assigning "0" requires $O(1)$ time and space. However, for a general Huffman tree, total space includes recursion stack and code_map storage, giving $O(k\ log\ k)$ best case and $O(k^2)$ worst case.

7.3 encode_data_to_bitarray()

It converts the input data (bytes) into a compressed bit representation using the Huffman code_map, then packs those bits into full bytes for storage.

7.3.1 Analysis

1. **Initialize bitarray:**

```
bits = bitarray()
```

   - **Total time:** $O(1)$
   - **Space:** $O(1)$

2. **Encode each symbol:**

```
for b in data:
    bits.extend(code_map[b])  # Append Huffman code for each byte
```

   - **Time per symbol:** $O(1)$ lookup + $O(1)$ append
   - **Total time:** $O(n)$, where n = number of symbols in input

- **Space:** $O(n)$, as bitarray grows with input

3. **Padding to full bytes:**

```
pad_len = (8 - len(bits) % 8) % 8  # Pad to make multiple of 8
bits.extend("0" * pad_len)
```

- **Time:** $O(1)$
- **Space:** $O(1)$

4. **Convert to bytes:**

- **Time:** $O(n)$, traverses all bits in the bitarray
- **Space:** $O(n)$, stores final byte sequence

7.3.2 Total Complexity:

- **Time:** $O(n)$
- **Space:** $O(n)$

8 Main Compression Algorithm:

The function compresses a byte sequence using Huffman coding. It builds a frequency table, constructs a Huffman tree, generates codes, and encodes the input into a compressed byte stream with a header containing metadata.

8.1 Defining Function

```
def compress_huffman(data: bytes, ext: str = "txt") -> bytes:
```

- Time: $O(1)$ to initialize a function
- Space: $O(1)$

8.2 Condition Check

To check if data is present or not.

```
if not data:
        return b''
```

- Time: $O(1)$
- Space: $O(1)$

8.3 Header Preparation

Creates a header starting with `"HUF1"` to identify the file as a Huffman-compressed file. Encodes the file extension (`ext`) and stores its length. Constructs a **frequency table** using `Counter(data)` that counts occurrences of each symbol. Stores the number of unique symbols (`k`) and each symbol's frequency as metadata.

```
header = bytearray(b"HUF1")
```

- **Time:** $O(1)$ → creating a 4-byte array.
- **Space:** $O(1)$ → stores 4 bytes.

```
freq_table = Counter(data)
```

- Time: $\boldsymbol{O(n)}$ - iterate entire input
- Space:
  - **Best Case:** $\boldsymbol{O(1)}$ when $k = 1$ (all same byte)
  - **Worst Case:** $\boldsymbol{O(k)}$ when many unique bytes

```
ext_bytes = ext.encode("utf-8")
if len(ext_bytes) > 10:
   raise ValueError("Extension too long")

header += bytes([len(ext_bytes)]) + ext_bytes
```

- **Time:** $O(1)$ – Only encodes extension to binary and concatenate to header file
- **Space:** $O(1)$ – Stores only extension in byte form

```
k = len(freq_table)
```

- **Time:** $O(1)$ – Length is stored as metadata
- **Space:** $O(1)$ – just stores an integer

```
header += k.to_bytes(2, 'big')
for symbol, freq in freq_table.items():
    header += bytes([symbol]) + freq.to_bytes(8, 'big')
```

- **Time:** $O(k)$ – The loop runs k times to add symbols and their respective frequencies in bytes to the header
- **Space:** $O(k)$ – As we are concatenating k times.

Total Complexity:

- **Time:** $O(n)$ – iterate entire input in frequency counter
- **Space:** $O(k)$ – store frequency table + extension info

8.4 Huffman Tree Construction

Builds the Huffman tree from the frequency table using a min-heap (heapq).

```
root = build_huffman_tree(freq_table)
```

- **Time:**
  - **Best Case & Worst Case:** $O(klog_2k)$ – loop runs $k-1$ times, each iteration $O(log\ k)$ (within build_huffman_tree)
- **Space:**
  - **Best Case & Worst Case:** $O(k)$- $k\ leaves\ +\ (k-1)$ $internal\ nodes\ =\ 2k-1$

8.5 Generating Huffman Codes:

Recursively traverses the tree to assign a unique binary code to each symbol.

```
code_map = {}
generate_huffman_code(root, "", code_map)
```

- **Time:** The Huffman tree is traversed here
  - **Best Case & Worst Case:** $O(k)$ – Recursion occurs for nodes
- **Space:** Stores k codes, each code is $\leq h : O(k\,.h)$
  - **For best Case:** $O(klog_2k)$
  - **For worst Case:** $O(k^2)$

8.6 Data Encoding:

Converts input bytes into a `bitarray` using the Huffman codes. Pads the bitarray to a multiple of 8 bits and converts to bytes.

```
payload = encode_data_to_bitarray(data, code_map)
```

- **Time:**
  - **Best Case & Worst Case**: $O(n)$ – where n is the input size in bytes
- **Space:**
  - **Best Case & Worst Case**: $O(n)$ – stores encoded payload

8.7 Combining Header and Payload

Concatenates the header and the encoded payload to produce the final compressed byte stream.

```
return bytes(header + payload)
```

- Time: Copies all bytes – $\boldsymbol{O(n + k)}$
- Stores all bytes – $\boldsymbol{O(n + k)}$

8.8 Final Time Complexity

Operation	**Best Case**	**Worst Case**
Defining function	$O(1)$	$O(1)$
Build frequency table	$O(n)$	$O(n)$
Empty check	$O(1)$	$O(1)$
Build header	$O(k)$	$O(k)$
Build Huffman tree	$O(k \log k)$	$O(k \log k)$
Generate codes	$O(k)$	$O(k)$
Encode data	$O(n)$	$O(n)$
Final combination	$O(n + k)$	$O(n + k)$

Adding them up: $O(1) + O(n) + O(k) + O(k \log k) + O(k) + O(n) + O(n + k)$

Result: $\boldsymbol{O(n + k \log k)}$

- **Best Case**: $O(n + k \log k)$
- **Worst Case**: $O(n + k \log k)$

Since in typical files—whether text, images (BMP), CSV, program files, PDFs, DOCX, or others—the file size n is much larger than the number of unique symbols k, the complexity can be simplified to O(n) in practice.

8.9 Final Space Complexity:

Component	**Best Case**	**Worst Case**
Frequency table	$O(1)$	$O(k)$
Header Preparation	$O(1)$	$O(k)$
Huffman tree (build)	$O(k)$	$O(k)$
Code map	$O(k \log k)$	$O(k^2)$
Recursion stack	$O(\log k)$	$O(k)$
Bitarray/Payload	$O(n)$	$O(n)$
Final output	$O(n + k)$	$O(n + k)$

Adding them up (Best Case): $O(1) + O(1) + O(k) + O(k \log k) + O(\log k) + O(n) + O(n + k)$

Adding them up (Worst Case): $O(k) + O(k) + O(k) + O(k^2) + O(k) + O(n) + O(n + k)$

- **Best Case:** $\boldsymbol{O(n + k \log k)}$ when few unique symbols, balanced tree, good compression
- **Worst Case:** $\boldsymbol{O(n + k^2)}$ when any unique symbols, skewed tree, poor compression

- *Full complexity is O(n + klogk) best case, but n dominates for typical files*
- *Full complexity is O(n + k²) worst case, but n dominates for typical files*

Where:

- $\boldsymbol{n}$ = size of input data in bytes
- $\boldsymbol{k}$ = number of unique symbols
- $\boldsymbol{h}$ = height of Huffman tree
  - Best case: $h = O(\log k)$
  - Worst case: $h = k$

For byte-based files, each symbol is a byte that can take a value from 0 to 255. Hence, the **maximum possible number of unique symbols is 256**.

- In practice, the actual number of unique symbols depends on the file content:
  - Simple text or grayscale $images \rightarrow k < 256$
  - Complex images, binary files, PDFs, DOCX → $k\ can\ approach\ 256$

For analysis purposes, we use $\boldsymbol{k \leq 256}$ as an upper bound for worst-case complexity calculations.

8.10 Dominance Analysis

In Big O notation, a term dominates if it grows significantly faster than others as input size increases.

This means:

- $\boldsymbol{k \log k \leq 2,048}$ (maximum possible value)
- $\boldsymbol{k^2 \leq 65,536}$ (maximum possible value)
- $\boldsymbol{n}$ **can be unlimited** (any file size)

File Size (n)	**k** (max)	**n**	**k log k**	**k²**	**Dominant Term**
100 bytes	100	**100**	664	10000	k^2
1 KB	256	**1,024**	2,048	65,536	k^2
64 KB	256	**65,536**	2,048	65,536	$n \approx k^2$

100 KB	256	**102,400**	2,048	65,536	n
1 MB	256	**1,048,576**	2,048	65,536	n
1 GB	256	**1,073,741,824**	2,048	65,536	n

From the Table,

Time (Best and Worst Case):

$(n\ vs\ k\ log\ k)$**:** n dominates when $n\ >\ 2$ KB

Space:

Best Case: $(n\ vs\ k^2)$**:** n dominates when $n\ >\ 64$ KB

Worst Case: $(n\ vs\ klogk)$**:** n dominates when $n\ >\ 2$ KB

8.11 Conclusion:

Best Case:

For typical files larger than 2 KB, the complexity simplifies to:

- **Time:** $O(n\ +\ k\ log\ k)\ \rightarrow\ O(n)$
- **Space:** $O(n\ +\ k\ log\ k)\ \rightarrow\ O(n)$

Worst Case:

For typical files larger than 64 KB, the complexity simplifies to:

- **Time:** $O(n\ +\ k\ log\ k)\ \rightarrow\ O(n)$
- **Space:** $O(n\ +\ k^2)\ \rightarrow\ O(n)$

9 Extent Of Compression

The extent of compression can be calculated using the following formula:

$$Compression\ Ratio\ (\%) = \frac{(Original\ Size - Compressed\ Size)}{Original\ Size} \times 100$$

The compression tool was tested on various file formats including text files, BMP images, CSV datasets, and code files.
The overall compression efficiency varies depending on data redundancy and file complexity.

9.1 Text File (.txt)

Text files generally contain high redundancy (repeated characters), making them highly compressible.

- **Best case:** ~87% reduction (highly repetitive text)
- **Typical case:** ~35–50% reduction (normal English text)
- **Range achieved: 37% – 87% compression**

9.2 BMP Images (.bmp)

BMP images with patterns or solid colors compress very well, but complex images compress less because pixel data has low redundancy.

- **Simple BMPs:** 80–83% reduction
- **Complex BMPs:** 24–27% reduction
- **Range achieved: 24% – 83% compression**

9.3 CSV Data Files (.csv)

CSV files contain structured numbers and repeated patterns (commas, column names), giving moderate compression.

- Achieved compression: **~30% reduction**

9.4 Program Files (.py, .exe, .circ)

These files contain mixed data and less redundancy than text but still show moderate compression.

- **.py files:** 26–29% reduction
- **.exe file:** ~37% reduction
- **.circ file:** ~36% reduction

Range achieved: 26% – 37% compression

NOTE:

1. Extremely Small Files

Very small files (< 1 KB) may show no compression or even a slight size increase because the Huffman header (frequency table + metadata) becomes larger than the file itself.

Example:

- A 0.51 KB file became 0.66 KB (–31%)

This behavior is expected and is a known limitation of Huffman coding on tiny data.

2. Highly Compressed File Types (DOCX, PDF, PNG)

Formats like **DOCX, PDF, and PNG** already use strong internal compression (ZIP/DEFLATE). Because of this, applying Huffman coding on top provides **very limited reduction** and may sometimes **increase the file size due to compression overhead**.

DOCX (Word Files)

- Achieved in test: **50.71%** (text-heavy file)
- Normally: **10%–50%**

PDF (Documents)

- Achieved in test: **2.75%**
- Normally: **0%–10%**

PNG (Images)

- Typically: **0%–5%** (often no benefit)

10 Sample input and output files

The following tables presents sample files used during testing, along with their original size, compressed size, and compression achieved.

10.1 Text Files (.txt)

File Name	Original Size	Compressed Size	Saved
test_repetitive.txt	10.0 MB	1.25 MB	**87.5%**
lorem.txt	54.36 MB	27.3 MB	**49.78%**
sample.txt	107.02 KB	66.8 KB	**37.58%**

10.2 Bitmap Images (.bmp)

File Name	Original Size	Compressed Size	Saved
checker_pattern.bmp (simple)	12.05 KB	2.26 KB	**81.28%**
triangle.bmp (simple)	117.24 KB	19.65 KB	**83.24%**
image4.bmp (complex)	22.9 MB	17.26 MB	**24.63%**
bmp_13.bmp (complex)	14.64 MB	10.62 MB	**27.44%**

10.3 Program Files (.py, .exe, .circ)

File Name	Type	Original Size	Compressed Size	Saved
complication.py	Python	3.04 KB	2.23 KB	**26.49%**
DoublyLinkedList.py	Python	3.61 KB	2.55 KB	**29.3%**
Lab1_Q1.exe	Executable	59.22 KB	37.37 KB	**36.9%**
FINAL_CEP.circ	Logisim	31.53 KB	19.99 KB	**36.59%**

10.4 CSV File (.csv)

File Name	Original Size	Compressed Size	Saved
annual-enterprise-survey.csv	8.39 MB	5.46 MB	**34.87%**
customers-100.csv	16.86 KB	12.3 KB	**27.05%**

10.5 Document Files (DOCX, PDF)

File Name	Type	Original Size	Compressed Size	Saved
RLC_Energy_Storage_Report.docx	Word	1.46 MB	736.22 KB	**50.71%**
workbook.pdf	PDF	1.03 MB	1.00 MB	**2.75%**
DLD_CEP_REPORT.docx	Word	466.02 KB	465.41 KB	**0.13%**

Notes

- Simple or repetitive data compresses extremely well (80–90%).
- Complex images and already-compressed formats (PDF, DOCX, EXE) usually show minimal gain, rarely higher.
- Very small files may experience compression overhead.
- **All sample input and output files are included in the GitHub repository and Zipped File Attached.**
- **Output Files Name:** $< name > (< extension >).huff$
  GitHub Link:
  https://github.com/Usman-Rasheed-Siddiqui/file-compressor/tree/main/sample

11 Key Observations and Advantages

- Efficient compression for repetitive and large datasets (e.g., large text files, simple BMP images).
- Consistent performance across diverse file types like text, CSV, and program files.
- Compression ratios scale well with increasing file size, with minimal time overhead.
- Algorithm is simple, deterministic, and easy to implement using Python libraries.

12 Limitations of the Compression Algorithm

- **Already-compressed formats:** The algorithm achieves little or no compression for files that are already compressed, such as PDF, PNG, JPEG, and other deflated formats. In rare cases, compression overhead can even slightly increase the file size.
- **Very small files:** Files smaller than ~1 KB may not compress effectively, and can sometimes result in a larger compressed file due to header overhead.

13 Why Choose Huffman Coding

We opted for **Huffman coding** because:

- **LZ77**: Time and space complexities are harder to calculate and analyze.
- **BWT, DELTA, RLE**: Produced extremely large compression overhead in our tests.
- **Huffman**: Offers a good balance of **compression efficiency** and **predictable performance**, with straightforward implementation and analysis.

14 Decompression Workflow (Huffman Decoding)

- Read and verify the **header** (HUF1 signature).
- Extract **file extension**, **frequency table**, and **payload bytes**.
- Rebuild the **Huffman Tree** using the frequency table.
- Read the **padding length** and reconstruct the **bit stream**.
- Remove padding bits from the end.
- Traverse the Huffman tree using each bit to **decode symbols**.
- Output the **original uncompressed data + extension**.

15 GitHub Repository Link

All the source code for the compression and decompression tool, along with sample input and output files for various formats (TXT, BMP, CSV, DOCX, PDF, etc.), is available at the GitHub repository:

https://github.com/Usman-Rasheed-Siddiqui/file-compressor

16 Bibliography

The code was modified to achieve better efficiency and optimized time complexity. Some functions were adapted with guidance from:

GeeksForGeeks:
Huffman Coding in Python

In the GeeksForGeeks implementation, the space complexity is stated as $O(n)$ (where $n = k$, the number of unique symbols), because they do not account for the length of `current_node` when generating codes. In our implementation, we include the length of `current_node`, which results in a space complexity of $O(k\ log\ k)$ on average and $O(k^2)$ in the worst case.