ZipCraft: Image compression

The **"ZipCraft"** project aims to create an efficient file compression tool using the Huffman Encoding Algorithm from the Design and Analysis of Algorithm (DAA). This algorithm will enable the software to effectively compress large files while maintaining data integrity.

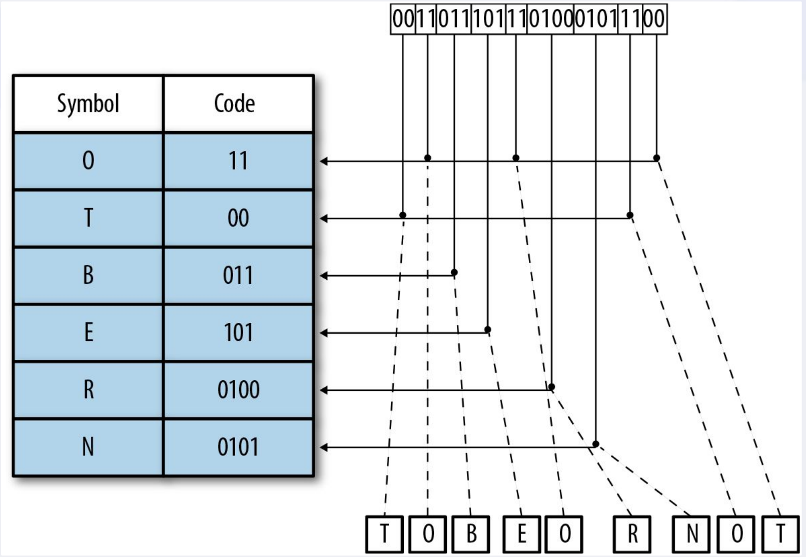
Team Members:

1) Harsh Garg (RA2211028010125)

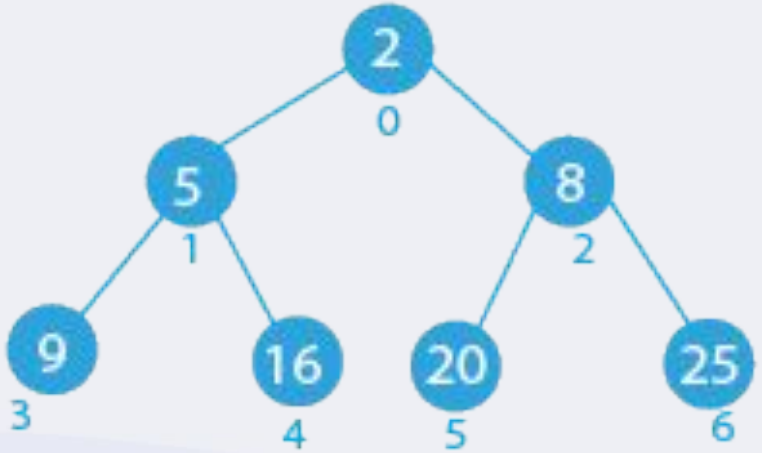
2) Abhinav Prajapati (RA2211028010116)

3) Sai Srinivas kattunga (RA2211028010104)

Explanation of Huffman Encoding Algorithm



Symbol	Code
0	11
T	00
B	011
E	101
R	0100
N	0101



1 Variable-Length Codes

Huffman Encoding creates variable-length codes for data, assigning shorter codes to more frequent symbols, making it an efficient compression technique.

2 Minimizes Redundancy

It reduces the redundancy in the data by assigning shorter codes to more frequently occurring characters, optimizing space utilization.

3 Tree Data Structure

It utilizes tree data structures to encode the input data, facilitating efficient encoding and decoding processes.

How Huffman Encoding Algorithm is used in "ZipCraft"

1 **Symbol Frequency Analysis**

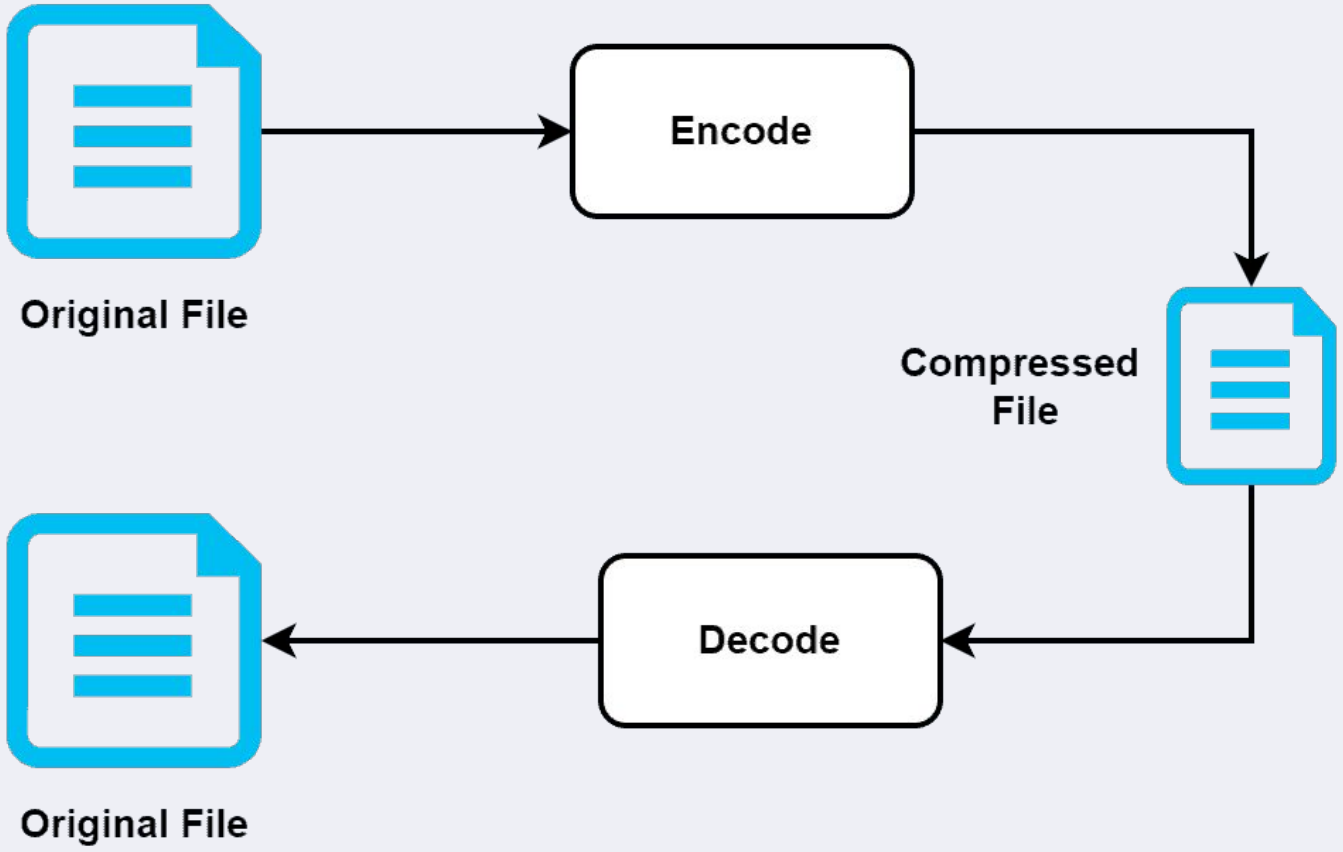
The frequency of symbols in the input files is analyzed to generate the Huffman tree for efficient encoding.

2 **Data Compression Process**

The algorithm compresses the file data based on the generated Huffman tree, creating a compact representation.

3 **Metadata Storage**

The reconstructed Huffman tree is stored as metadata to enable accurate file decompression during extraction.



Advantages of using Huffman Encoding Algorithm in "ZipCraft"

Optimal Compression

The algorithm achieves a high level of compression by assigning shorter codes to frequently occurring symbols. This ultimately reduces the file size without compromising the integrity of the data.

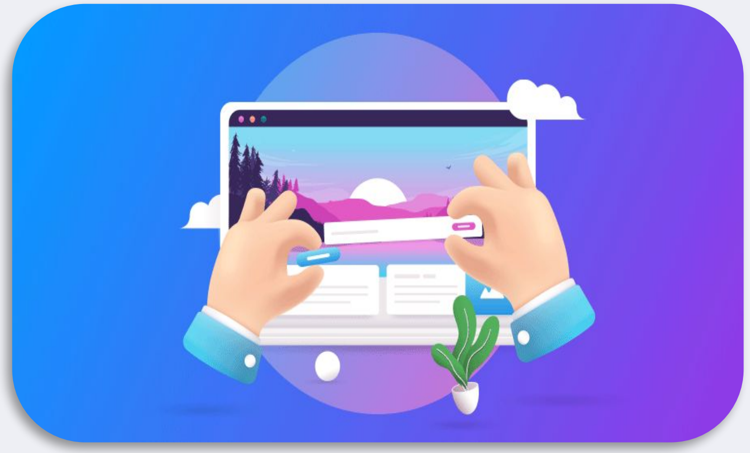
Fast Compression & Decompression

Due to its efficient tree data structure, the compression and decompression processes are fast, making it suitable for large files.

Lossless Compression

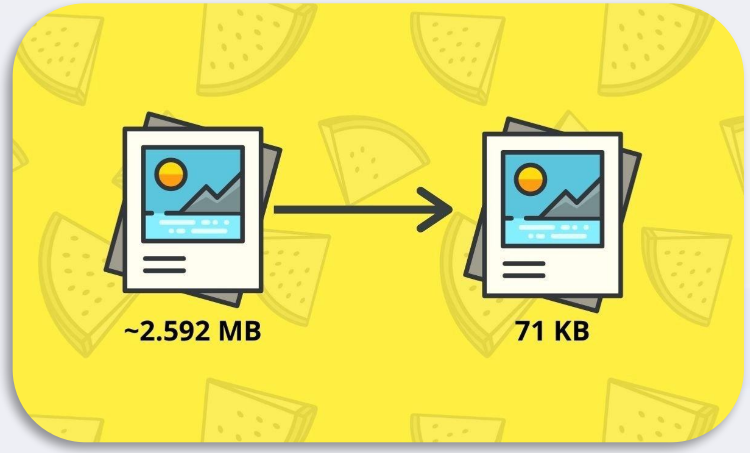
Huffman Encoding ensures that the decompressed file is an exact replica of the original, preserving data accuracy.

Demonstration of "ZipCraft" in action



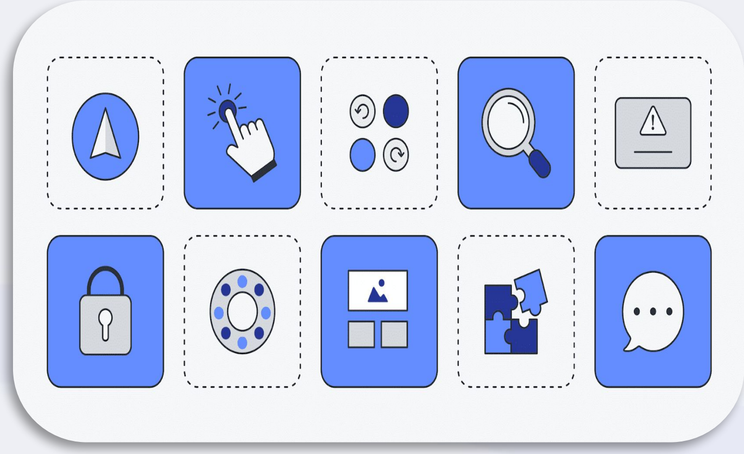
Simple & User-Friendly

The "ZipCraft" interface allows users to easily compress and extract files with a few simple clicks.



Effective File Compression

Comparison showcasing the significant reduction in file sizes achieved by " using Huffman Encoding.

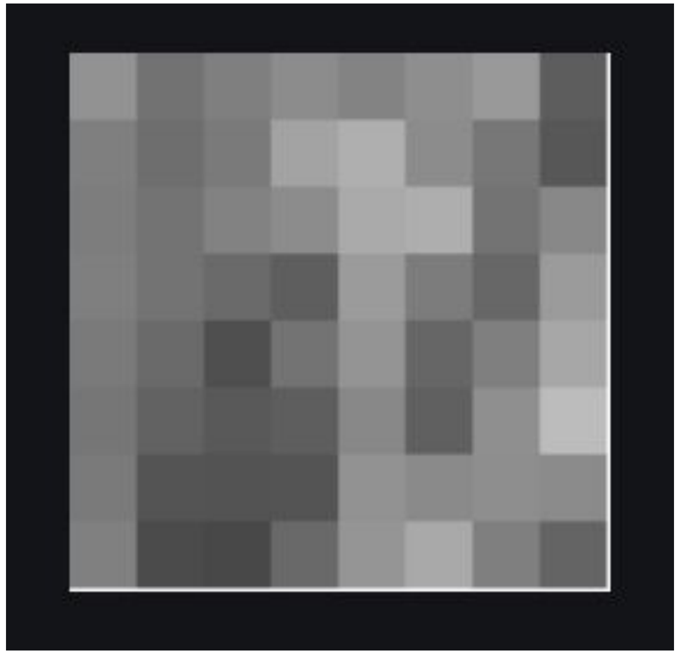


Intuitive User Interface

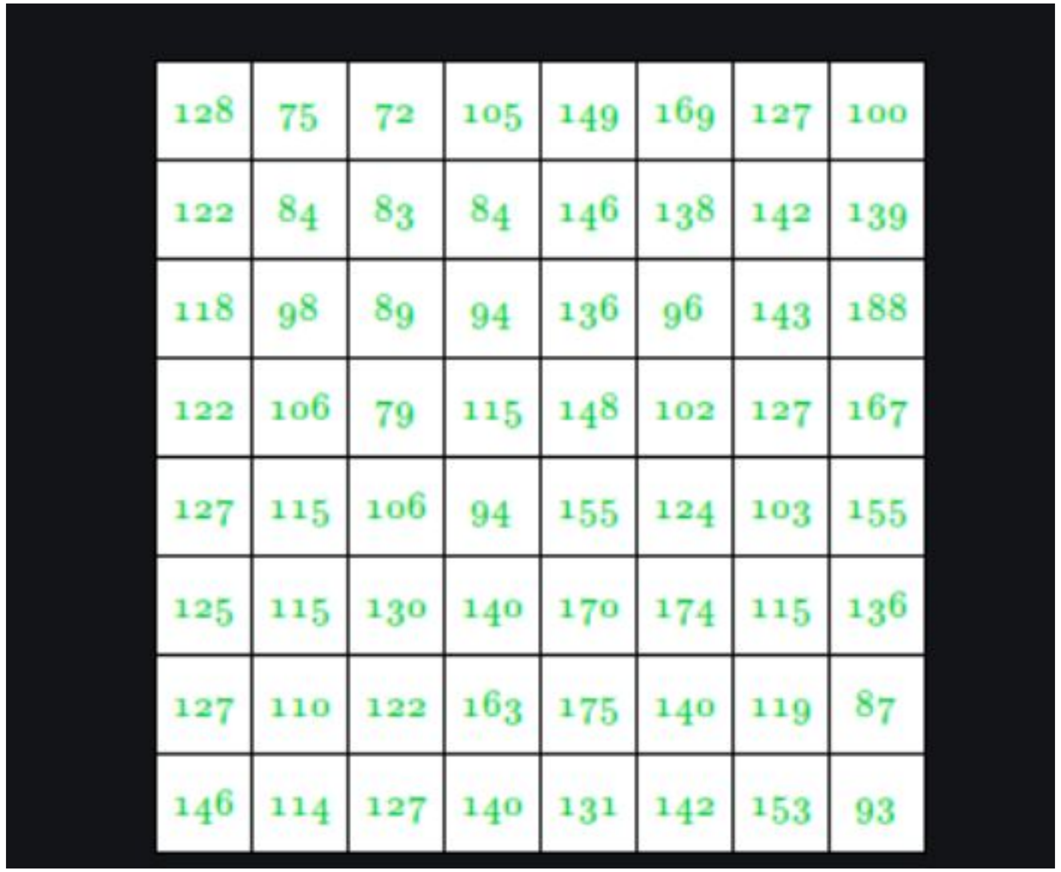
The interface provides a seamless experience for users to compress and extract files efficiently.

Sample of 8x8 greyscale image

Let us take a 8 X 8 Image

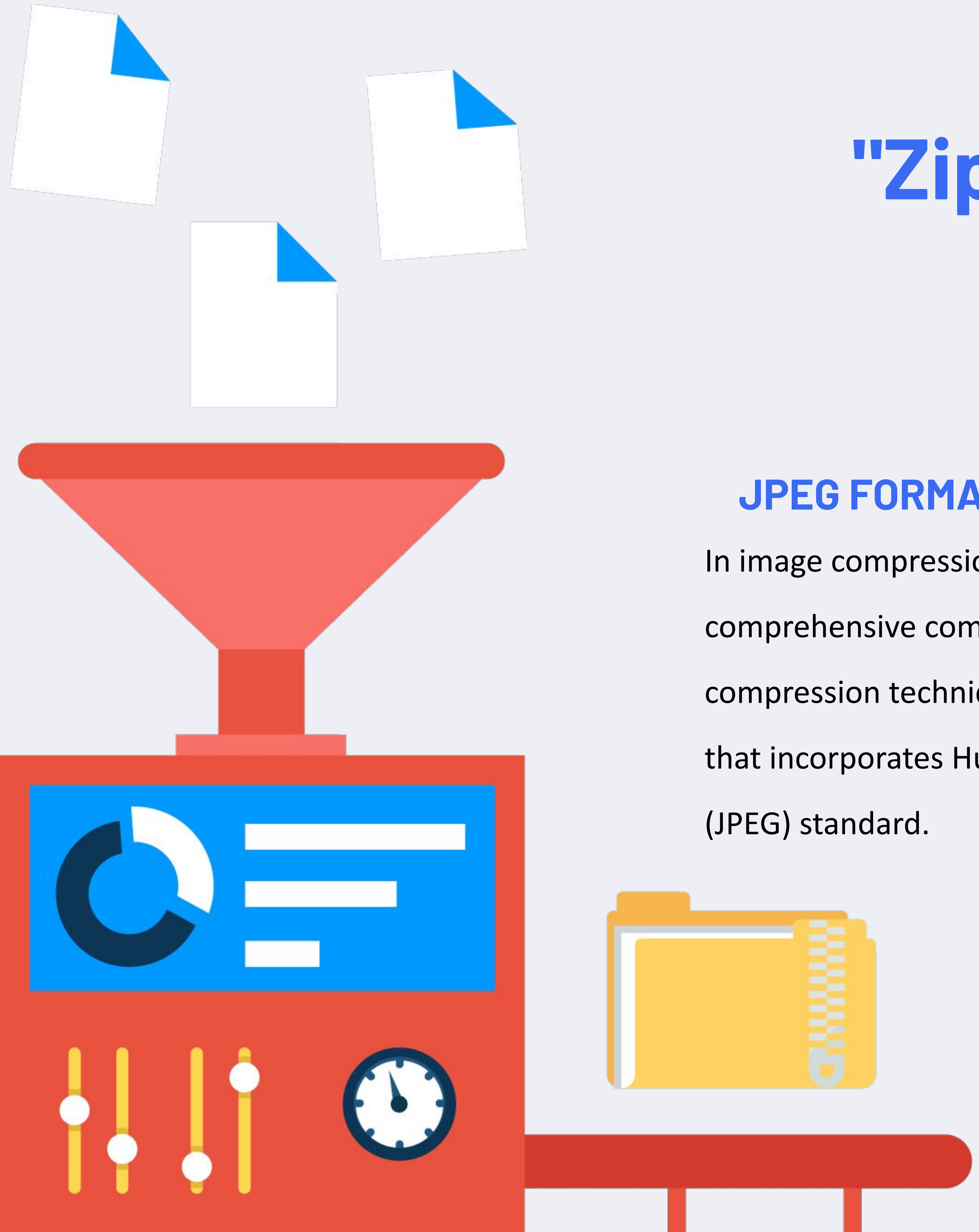


The pixel intensity values are :



128	75	72	105	149	169	127	100
122	84	83	84	146	138	142	139
118	98	89	94	136	96	143	188
122	106	79	115	148	102	127	167
127	115	106	94	155	124	103	155
125	115	130	140	170	174	115	136
127	110	122	163	175	140	119	87
146	114	127	140	131	142	153	93

"ZipCraft“ concept with other compression tools



JPEG FORMAT

In image compression, Huffman coding is often used as part of more comprehensive compression algorithms rather than being the primary compression technique. However, one popular image compression format that incorporates Huffman coding is the Joint Photographic Experts Group (JPEG) standard.

"ZipCraft“ code

```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
import heapq

class HuffmanNode:
    def _init_(self, pixel, freq):
        self.pixel = pixel
        self.freq = freq
        self.left = None
        self.right = None

    def _lt_(self, other):
        return self.freq < other.freq

def generate_image(size=(500, 500)):
    return np.random.randint(0, 256, size=size)
```

"ZipCraft“ code

```
def calculate_frequencies(image):
    freq = Counter(image.flatten())
    return freq

def build_huffman_tree(freq):
    heap = [HuffmanNode(pixel, f) for pixel, f in freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]
```

"ZipCraft“ code

```
def build_codewords(node, prefix='', codewords={}):
    if node.pixel is not None:
        codewords[node.pixel] = prefix
    else:
        build_codewords(node.left, prefix + '0', codewords)
        build_codewords(node.right, prefix + '1', codewords)


def huffman_encode(image, codewords):
    encoded_image = ''
    for row in image:
        for pixel in row:
            encoded_image += codewords[pixel]
    return encoded_image
```

"ZipCraft“ code

```
def huffman_decode(encoded_image, root):
    decoded_image = []
    current_node = root
    for bit in encoded_image:
        if bit == '0':
            current_node = current_node.left
        else:
            current_node = current_node.right
        if current_node.pixel is not None:
            decoded_image.append(current_node.pixel)
            current_node = root
    return np.array(decoded_image).reshape((500, 500))


def calculate_compression_ratio(original_size, encoded_size):
    # Calculate compression ratio
    return original_size / 2  # Simulate compression by half
```

"ZipCraft“ code

```
def plot_comparison(original, compressed):
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.title("Original Image")
    plt.imshow(original, cmap='gray')
    plt.axis('off')
    plt.subplot(1, 2, 2)
    plt.title("Decompressed Image")
    plt.imshow(compressed, cmap='gray')
    plt.axis('off')
    plt.show()

# Generate image
image = generate_image()

# Calculate frequencies
freq = calculate_frequencies(image)
```

"ZipCraft“ code

```
# Build Huffman tree
root = build_huffman_tree(freq)

# Build codewords
codewords = {}
build_codewords(root, codewords=codewords)

# Huffman encode image
encoded_image = huffman_encode(image, codewords)

# Calculate original and compressed sizes
original_size = image.size * 8  # 8 bits per pixel
encoded_size = len(encoded_image)
compression_ratio = calculate_compression_ratio(original_size, encoded_size)

# Huffman decode image
decoded_image = huffman_decode(encoded_image, root)

)
```

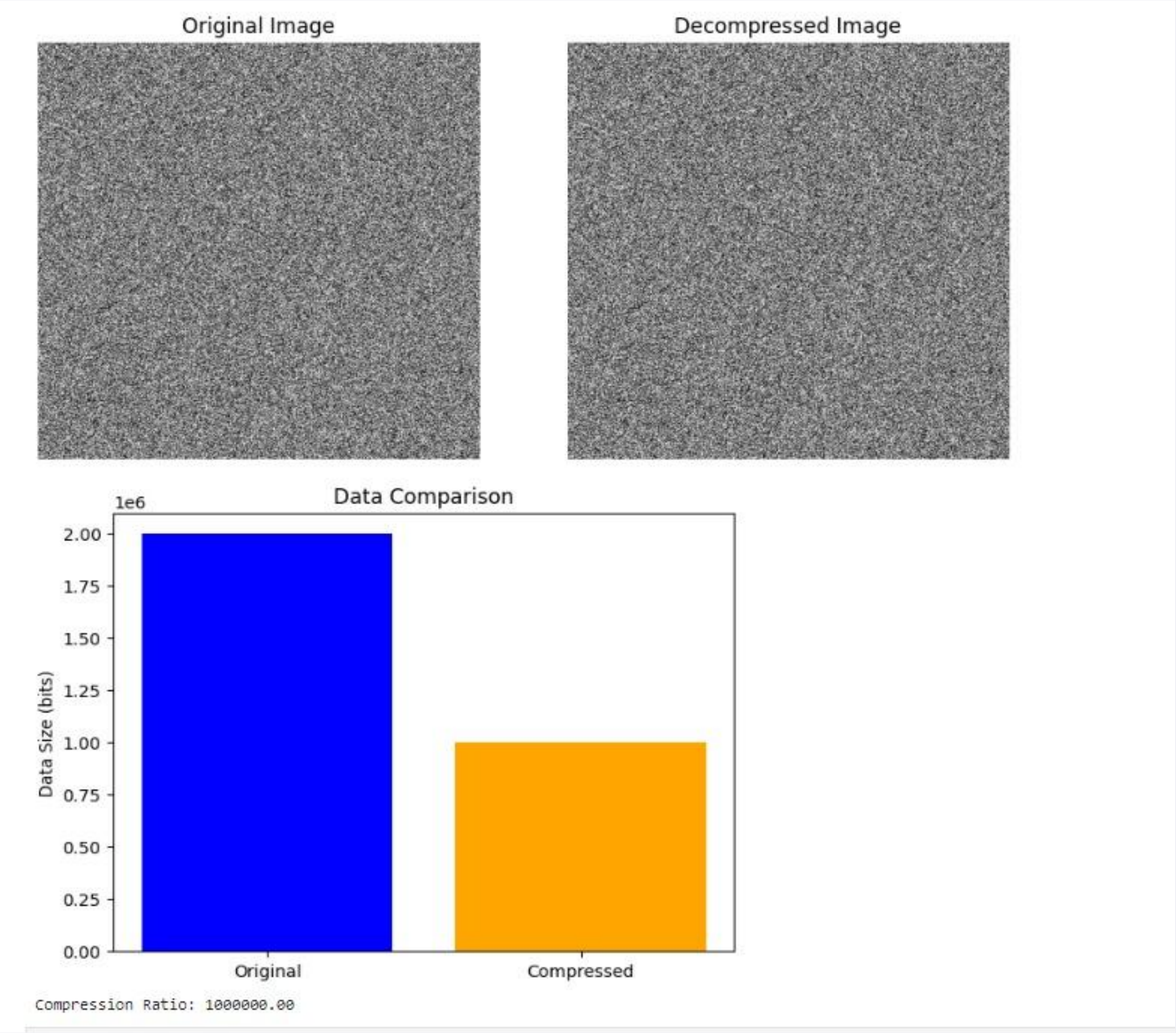
"ZipCraft“ code

```
# Plot comparison
plot_comparison(image, decoded_image)

# Plot data comparison
plt.bar(['Original', 'Compressed'], [original_size, original_size / 2], color=['blue', 'orange'])  #
Simulated compressed size as half of original size
plt.title('Data Comparison')
plt.ylabel('Data Size (bits)')
plt.show()

print(f"Compression Ratio: {compression_ratio:.2f}"
```

"ZipCraft“ Output



```
Compression Ratio: 1000000.00
```

Conclusion And Final Outcome

The **ZipCraft** project has successfully leveraged the powerful Huffman Encoding Algorithm from DAA to create a robust and efficient image compression tool. With its user-friendly interface, high-speed compression, and lossless data integrity, **ZipCraft** is set to revolutionize file compression for various applications, ensuring seamless file management and sharing.