Efficiency Comparison Between RLE-Fibonacci and RLE-Huffman Coding in Grayscale Image Compression

Dzaki Ahmad Al Hussainy - 13524084
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13524084@mahasiswa.itb.ac.id , 13524084@std.stei.itb.ac.id

***Abstract*— This paper presents a comparative analysis of two hybrid lossless compression methods for grayscale images: RLE-Fibonacci and RLE-Huffman. Both techniques begin by applying Run-Length Encoding (RLE) to reduce spatial redundancy by encoding consecutive identical pixel values. The RLE-Fibonacci method then compresses the run-length values using Fibonacci coding, a universal binary code that is efficient for representing small integers. In contrast, the RLE-Huffman approach applies Huffman coding, which assigns variable-length codes based on the frequency of each run-length value. The performance of both methods is evaluated using standard grayscale image datasets and assessed through key metrics such as Compression Ratio (CR).**

Keywords*— *Gray scale, Fibonacii Coding, Run Length Encoding, Huffman Coding, Lossless Compression

I. Introduction

With the growing demands of today's digital era, efficient storage and transmission of data—especially digital images—have become increasingly important. The rise in digital photos and illustrations calls for the use of lossless data compression systems. Lossless compression is particularly crucial in fields such as medical data storage and archival systems.

Grayscale images, which are typically represented using a single channel per pixel, often exhibit large regions of uniform intensity. This characteristic makes them highly suitable for Run-Length Encoding (RLE), a simple yet effective compression algorithm that reduces spatial redundancy by encoding consecutive identical values as a single value paired with a count.

To further enhance the efficiency of RLE, it can be combined with other encoding techniques. One such method is Fibonacci coding, a universal binary encoding scheme that efficiently represents small positive integers, such as the run lengths produced by RLE. This combined approach—referred to as RLE-Fibonacci—can offer improved compression performance, particularly for images with frequent short run lengths.

Another well-established method for compressing data is Huffman coding, which assigns variable-length binary codes to data values based on their frequency of occurrence. When combined with RLE, the resulting RLE-Huffman method leverages both spatial redundancy and statistical coding efficiency to reduce image size.

This paper presents a comparative study of these two hybrid lossless compression approaches: RLE-Fibonacci and RLE-Huffman. Both methods are applied to standard grayscale image datasets and evaluated using metrics such as Compression Ratio (CR) . The goal is to analyze and compare their efficiency, highlight their respective advantages and limitations, and determine their suitability for different types of grayscale image data.

II. Theoritical Framework

A. Digital Image

In the field of image processing, an image is referred as a picture that can be defind as representation that exists in two-dimensional (2D) space. An image is a continous two-dimentional signal that can be perceived by the human visual system. More formally, it can be described mathematically as two-dimentional function, where the coordinates (x,y) indicate specific points on the 2D plane, and the function f(x,y) correspond to the intensity of light, or brightness, at each of those point [1].

Images can be classified based on the number of frames they contain. Broadly, there are two main types: still images and moving images. A still image is a static, single frame that does not change over time. In contrast, a moving image consists of a sequence of frames displayed in rapid succession, creating the illusion of motion. Each individual frame in a moving image is essentially a still image captured at a specific moment in time[1].

A digital image is a discrete representation of a continuous visual signal, obtained through sampling in both space and time [1]. Temporal sampling refers to the process of capturing a sequence of frames over time, as in video. Spatial sampling involves measuring the intensity of light at discrete coordinate points (x, y) within each frame [1].

Digital images are typically represented as two-dimensional matrices of size M x N, where M and N denote the image's resolution in terms of rows and columns, respectively [1].

$$f(x,y) = \begin{bmatrix} f(0,0) & \cdots & f(0,N-1) \\ \vdots & \ddots & \vdots \\ f(M-1,0) & \cdots & f(M-1,N-1) \end{bmatrix}$$

Each element in the matrix corresponds to a pixel (short for picture element), which stores the intensity or color value at a specific spatial location [1]. Each pixel in an image is defined by the function f(x, y), which determines the brightness or darkness at the corresponding coordinates (x, y). Example image 2. 2 have a resolution of 1200 × 1500 pixels contains a total of 1,800,000 individual pixels. This means the image has 1200 rows and 1500 columns, and each pixel represents a specific intensity or color value at a unique position (x, y) within the image [1].

Fig 2.A.1 Ilustration pixel on image

(Source: [1])

The value of f(x, y) depends on the type of image. In a grayscale image, f(x, y) consists of a single intensity value that represents the level of brightness, typically ranging from black to white. In contrast, a colored image uses a vector-valued function f(x, y) = (R, G, B), where each component corresponds to the intensity of the red, green, and blue channels. These three components are combined to produce a wide range of colors perceived in the image.

B. Grayscale Image

A grayscale image is an image that represents pixel intensity values using quantized levels of light intensity.[2] Quantization refers to the process of discretizing a continuous range of light intensity at each spatial coordinate (x, y).[2] The main purpose of quantization is to map continuous light values into K discrete intensity levels. Each pixel's intensity is represented by a value in the range [0, K - 1], where this range is referred to as the gray level. The quantization process divides the intensity scale into K distinct values such as 0, 1, 2, …, K – 1 [2]. Typically, K is chosen as a power of two, expressed as $K = 2^m$, where m is a positive integer. In this context, K defines the number of gray levels available, and m determines the number of bits used to represent each pixel [2].

$K = 2^m$

Skala keabuan	Rentang nilai keabuan	Pixel depth
2^1 (2 nilai)	0, 1	1 bit
2^2 (4 nilai)	0, 1, 2, 3	2 bit
2^3 (8 nilai)	0, 1, 2, 3, 4, 5, 6, 7	3 bit
2^4 (16 nilai)	0, 1, ..., 15	4 bit
2^8 (256 nilai)	0, 1, ..., 255	8 bit

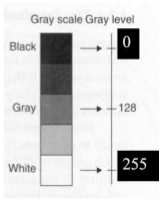


Fig 2.B.1 representaion of K value

(Source: [2])

C. Image Compression

Image compression is a technique used to reduce redundancy in the representation of image data, with the goal of minimizing storage requirements without significantly compromising image quality. A compressed image requires less memory compared to an uncompressed one, and as a result, transmission of compressed images is generally faster and more efficient [3].

The main objective of image compression is to eliminate redundancy such as repetitive data that can be represented more compactly. Redundancy refers to repeating patterns or predictable elements within the image that can be encoded more efficiently. There are three primary types of redundancy in images:

1. Coding Redundancy
   This occurs when the same pixel values are represented using longer or inefficient codewords[3].

2. Spatial/Temporal Redundancy
   This type of redundancy arises when neighboring pixels (in space or in consecutive frames of a video) have similar or identical intensity values [3].

3. Psychovisual Redundancy
   This refers to information that is perceptually less important or even imperceptible to the human visual system [3].

D. Method to File compression

Image compression techniques can generally be divided into two categories based on whether or not they preserve all original data:

1. Lossy compression

   Lossy compression educes the file size significantly by permanently removing certain image details that are less noticeable to the human eye [3]. Although some information is lost in the process, the resulting image still maintains a visually acceptable quality. This method typically achieves a higher compression ratio compared to lossless techniques, making it suitable for applications where perfect accuracy is not essential, such as web graphics and digital photography. Common examples

include JPEG compression and fractal-based image compression [3].

2. Lossless compression

   Lossless compression preserves all the original image data, allowing the image to be perfectly reconstructed after decompression without any loss of information [3]. Although it generally results in lower compression ratios than lossy methods, it is essential in fields that require high fidelity and data integrity, such as medical imaging and X-rays. Examples of lossless compression techniques include Huffman coding, Run-Length Encoding (RLE), and predictive or quantized coding[3].

E. Run-Length Encoding (RLE)

Run-Length Encoding (RLE) is a compression technique used for reducing the size of images that contain many sequential pixels with the same gray level [3]. It works by creating a pair (p, q), where p is the gray level, and q is the number of consecutive pixels that share that grayness—this count is called the run length.

For example, a row like [120, 120, 120, 45, 45] would be encoded as [(120, 3), (45, 2)]. This helps save space by avoiding repeated values.

F. Fibonacii sequance

The Fibonacci sequence is a well-known numerical sequence defined by a specific recurrence relation. Each term in the sequence is the sum of the two preceding terms, starting from the initial values 0 and 1 [4]. Formally, the sequence F_n is defined as follows:

$$F_0 = 0, \quad F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}, \quad \text{for } n \geq 2$$

G. Fibonacii coding

Fibonacci coding encodes an integer into binary using the Fibonacci representation of a number, based on Zeckendorf's Theorem [5]. This theorem says that every positive number can be written as a sum of different Fibonacci numbers, as long as we don't use two that are next to each other in the sequence [5]. So for any number N, we can find a combination of non-consecutive Fibonacci numbers that add up to N. This unique way of writing a number is called its Zeckendorf representation, and it's what we use to build the Fibonacci code [5]. Then, to make decoding easy, we add a special '11' terminator at the end of the binary sequence.

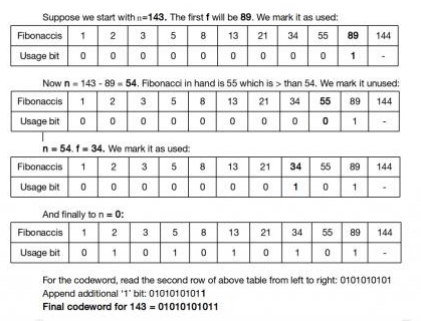


Fig 2.G.1 Ilustration pixel on Fibonacii coding

(Source: [5])

H. Huffman Coding

Huffman coding is a lossless compression method that encodes pixels with higher frequency using fewer bits than those with lower frequency. Huffman coding uses a Huffman tree to create the encoding. The algorithm is:

1. Every grayness intensity is represented as a vertex; each vertex is assigned the frequency of that grayness in every single pixel.
2. Sort all vertices in ascending order based on their frequency.
3. Combine the two vertices with the smallest frequencies into a new vertex. The new vertex's frequency is the sum of the two leaf frequencies.
4. Repeat step 2 until there is only one single binary tree.
5. Label the left edge with zero and the right edge with one.
6. Traverse the binary tree from the root to each leaf. The sequence of edge labels from root to leaf represents the Huffman code for the corresponding gray level.

The binary string produced by Huffman coding is known as a prefix code or prefix-free code. A prefix-free code means that no codeword is a prefix of any other codeword. This ensures that each code has a unique sequence of bits at the beginning, making all the codes distinct and allowing unambiguous decoding [6].

III. Implementation

A. Restrictions and Scope of Analysis

This paper will focus exclusively on grayscale images, with pixel intensities ranging from 0 (black) to 255 (white) and represented within K = 8 bits. Additionally, the scope of image analysis will be limited to pixel dimensions ranging from 10x10 to 100x100. To facilitate a clearer analysis and explanation of the Huffman tree, this study will not delve into aspects concerning image metadata, header files, or specific image formatting.

B. Implementation and Case Study

The grayscale image compression process starts by loading the image and converting it into a one-dimensional (1D) array, where each value represents the gray intensity of a pixel. Next, Run-Length Encoding (RLE) is applied to compress repeating pixel values. This step transforms the pixel sequence into a series of pairs, where each pair consists of a gray intensity value and its corresponding run length, effectively reducing redundancy in the image data.

After RLE, the compressed data is processed using two different encoding approaches for comparison. In the first method, the run length values are encoded using Fibonacci encoding, which utilizes unique binary representations based on Zeckendorf's Theorem. In the second method, the gray intensity values are encoded using Huffman coding, which assigns shorter binary codes to more frequently occurring intensities. Finally, the total compressed size from both methods is calculated and compared using the compression ratio to determine which method provides better compression efficiency. Here is the flow of the method used.

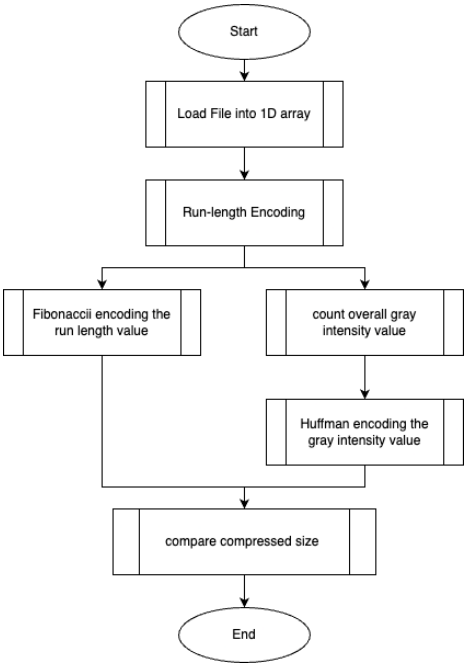


Image 3.B.1.1 Flow chart of compression method with RLE-Fibonacii and RLE Huffman Encoding

(Source: Author's Archive)

B.1 Converting Image Into Pixel Matrix

Before compressing happened the image need to be change into pixel matrix, which is 2D matrix with every single element represent grayscale range 0 to 255.

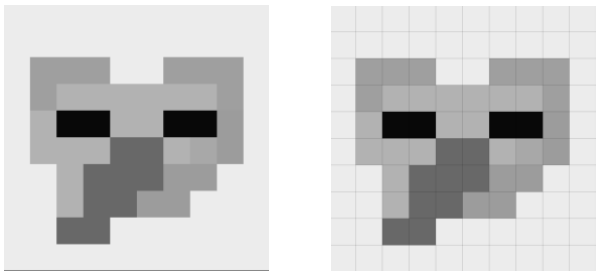


Fig 3.B.1.2 Ilustration 10x10 pixel 2D image grayscale pixel.

(Source: Author's Archive)

For illustration purposes, Figure 3.B.1, which visually resembles an elephant, can be represented as a 2D matrix, where each element corresponds to a pixel's grayscale intensity value. This matrix structure provides a digital representation of the image, suitable for further processing. Here is the visualised matrix.

$$\begin{bmatrix}
236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 \\
236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 \\
236 & 159 & 159 & 159 & 236 & 236 & 159 & 159 & 159 & 236 \\
236 & 159 & 178 & 178 & 178 & 178 & 178 & 178 & 159 & 236 \\
236 & 178 & 8 & 8 & 178 & 178 & 8 & 8 & 156 & 236 \\
236 & 178 & 178 & 178 & 104 & 104 & 178 & 168 & 156 & 236 \\
236 & 236 & 178 & 104 & 104 & 104 & 156 & 156 & 236 & 236 \\
236 & 236 & 178 & 104 & 104 & 156 & 156 & 236 & 236 & 236 \\
236 & 236 & 104 & 104 & 236 & 236 & 236 & 236 & 236 & 236 \\
236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 & 236 & 236
\end{bmatrix}$$

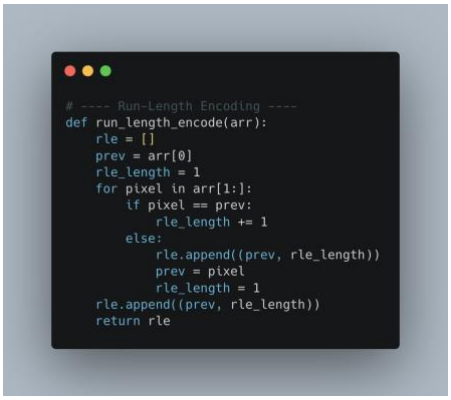
In the implementation, the Pillow (PIL) library is used to read the image file. Pillow is an open-source Python package widely used for image processing tasks, including opening, converting, and manipulating image data. To simplify the application of Run-Length Encoding (RLE), the 2D image matrix must first be flattened into a 1D array. This transformation is performed using the NumPy library, which provides efficient operations on numerical arrays. The following code snippet demonstrates this process:

Image 3.B.1.3 Implementation code for reading file and convert it into 1D array

(Source: Author's Archive)

B.2 Implementing RLE

After converting the image into 1D array RLE is applied to compress the pixel sequence into a series of pairs, where each pair consists of a gray intensity value and its corresponding run length, effectively reducing redundancy in the image data.



```
# ---- Run-Length Encoding ----
def run_length_encode(arr):
    rle = []
    prev = arr[0]
    rle_length = 1
    for pixel in arr[1:]:
        if pixel == prev:
            rle_length += 1
        else:
            rle.append((prev, rle_length))
            prev = pixel
            rle_length = 1
    rle.append((prev, rle_length))
    return rle
```

Fig 3.B.2.1 Implementation code RLE

(Source: Author's Archive)

An example of the resulting 1D array after applying Run-Length Encoding (RLE) can be represented as follows:

[(236, 21), (159, 3), (236, 2), (159, 3), (236, 2), (159, 1), (178, 6), (159, 1), (236, 2), (178, 1), (8, 2), (178, 2), (8, 2), (156, 1), (236, 2), (178, 3), (104, 2), (178, 1), (168, 1), (156, 1), (236, 3), (178, 1), (104, 3), (156, 2), (236, 4), (178, 1), (104, 2), (156, 2), (236, 5), (104, 2), (236, 16)]

In this representation, the first value of each pair denotes the grayscale intensity level (gray value), while the second value represents the corresponding run length, indicating how many consecutive pixels share the same intensity. After obtaining this encoded sequence, two different lossless compression methods can be implemented to further compress the data and evaluate their relative effectiveness.

C. Implementing Fibonacii Coding

The following steps are implemented in the provided code to generate the Fibonacci encoding of a given number:

1. Preallocate Fibonacci Numbers
   A list of Fibonacci numbers is generated starting from 1 and 2.
   Generation continues until the largest Fibonacci number less than or equal to the input number n is found.

2. Find Largest Fibonacci Number ≤ n

   A helper function largestFiboLessOrEqual(n) iteratively computes Fibonacci numbers until it finds the largest one satisfying $F_k \leq n$. This is for index that serves as the starting point for the encoding.

3. Initialize Codeword Array

   An array codeword of size index + 2 is initialized. The +2 ensures space for the binary representation and the mandatory terminating 1.

4. Encode Using Greedy Subtraction

   Beginning from the largest valid Fibonacci index, the algorithm subtracts Fibonacci values from n in descending order. If a Fibonacci number is part of the sum, a 1 is written at that position in the codeword array. If a Fibonacci number is skipped (because it would exceed the remaining value of n), a 0 is written instead.

5. Add Terminator

   After encoding the number, a final 1 is appended to mark the end of the codeword. This ensures the code is self-delimiting and prefix-free.

```
# ---- Fibonacci Encoding ----
# N denote preallocation size
N = 100

fib = [0 for _ in range(N)]

def largestFiboLessOrEqual(n):
    fib[0] = 1
    fib[1] = 2
    i = 2
    while fib[i - 1] <= n:
        fib[i] = fib[i - 1] + fib[i - 2]
        i += 1
    return i - 2

def fibonacciEncoding(n):
    index = largestFiboLessOrEqual(n)
    codeword = ['*' for _ in range(index + 2)] #initialize array
    i = index
    while n:
        codeword[i] = '1'
        n -= fib[i]
        i -= 1
        while i >= 0 and fib[i] > n:
            codeword[i] = '0'
            i -= 1
    codeword[index + 1] = '1'
```

Fig 3.C.1.1 Implementation code of Fibonacii coding

(Source: Author's Archive)

Example the number 21 from the first run-length can be coded into 00000011 in fibonacii encoding.

Next step is encode gray level using standard binary representation. The result is a mixed-format encoding where data redundancy is reduced by applying different encoding schemes suited to the nature of each component.

```
# Encode (value, count) -> (binary(value), fibonacci(count))
rle_encoded_list = [(decimal_to_binary(val), fibonacciEncoding(count)) for val, count in rle_list]
```

Fig 3.C.1.2 Implementation storing RLE value and Fibonacii code

(Source: Author's Archive)

Gray value	frequencies	Gray value in binary	Fibonacii encoded frequencies
236	21	11101100	00000011
159	3	10011111	0011
236	2	11101100	011
159	3	10011111	0011
236	2	11101100	011
159	1	10011111	11
178	6	10110010	10011
159	1	10011111	11
236	2	11101100	011
178	1	10110010	11
8	2	00001000	011
178	2	10110010	011
8	1	00001000	011
156	1	10011100	11
236	2	11101100	011
178	3	10110010	0011
104	3	01101000	011
178	1	10110010	11
168	1	10101000	11
156	1	10011100	11
236	3	11101100	0011
178	1	10110010	11
104	3	01101000	0011
156	2	10011100	011
236	4	11101100	1011
178	1	10110010	11
104	2	01101000	011
156	2	10011100	011
236	5	11101100	00011
104	2	01101000	011
236	16	11101100	0010011

After that comparing and analysis will continue in section E

D. Implementing Huffman Coding

Huffman coding is applied to encode the gray levels derived from the output of RLE. However, the frequency of each intensity value cannot be directly obtained from the original image and must be computed from the RLE result. Therefore, a helper function is required to traverse the RLE output and accumulate the total frequency of each gray level. The implementation of this helper function is presented as follows:

```
# Step 1: Count frequency of gray values from RLE
def count_gray_frequencies(rle_list):
    freq = defaultdict(int)
    for value, count in rle_list:
        freq[value] += count
    return freq
```

Fig 3.D.1.1 Implementation counting RLE Gray frequencies

(Source: Author's Archive)

The function count_gray_frequencies is designed to calculate the total frequency of each gray level from a list of Run-Length Encoded (RLE) values. In RLE format, each element of the list is a tuple (value, count), where value represents the gray values and count represents how many times it appears consecutively. After applying the function here is the sum of all of the frequencies:

Gray value	236	159	178	8	156	104	168
Frequencies	57	8	15	4	6	9	1

These frequencies are sorted then used as input to the Huffman coding algorithm. Huffman coding constructs an optimal binary tree based on symbol frequencies, assigning shorter binary codes to more frequent gray values and longer codes to less frequent ones. This variable-length, prefix-free encoding minimizes the average number of bits required to represent the intensity values. The resulting Huffman encoding tree as follow

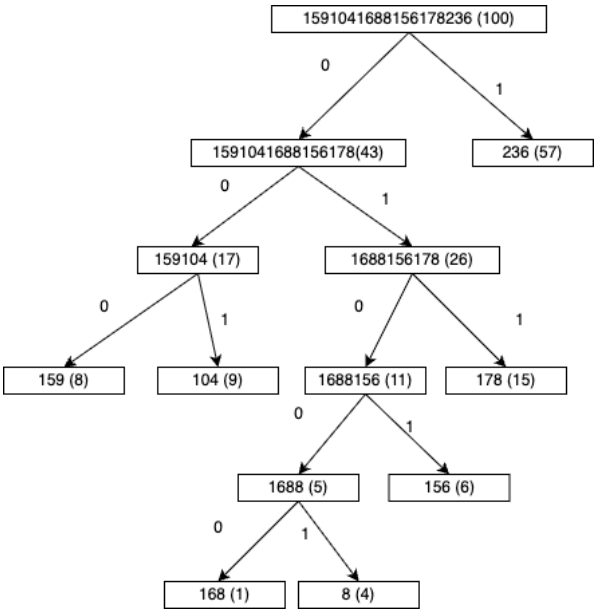


Fig 3.D.1.2 Huffman Tree Result from Huffman Coding

(Source: Author's Archive)

The resulting traversal of the Huffman tree produces the binary code for each gray level. Each leaf node in the tree represents a unique gray value, and the path from the root to that leaf determines its corresponding Huffman code. Moving left adds a 0 to the code, while moving right adds a 1. This traversal process ensures that frequently occurring gray values receive shorter codes, while less frequent values receive longer codes, thereby optimizing the overall compression efficiency. The resulting encoded binary representations for each gray value, obtained by traversing the Huffman tree, are presented as follows:

Gray value	frequencies	Encoded Huffman
168	1	01000
8	4	01001
156	6	0101
159	8	000
104	9	001
178	15	011
236	57	1

This is the implementation code for Huffman coding :

```python
# --- Huffman Encoding ---
# Node structure for Huffman tree
class HuffmanNode:
    def __init__(self, gray=None, freq=None, left=None, right=None):
        self.gray = gray      # Grayscale value
        self.freq = freq      # Frequency
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.freq < other.freq
```

```python
# Step 2–4: Build Huffman tree from frequency
def build_huffman_tree(freq_dict):
    heap = []
    for gray, freq in freq_dict.items():
        heapq.heappush(heap, HuffmanNode(gray=gray, freq=freq))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(freq=left.freq + right.freq, left=left, right=right)
        heapq.heappush(heap, merged)

    return heap[0] if heap else None

# Step 5–6: Traverse tree to get binary Huffman codes
def generate_huffman_codes(tree):
    codes = {}

    def traverse(node, prefix=""):
        if node is None:
            return
        if node.gray is not None:
            codes[node.gray] = prefix
        traverse(node.left, prefix + "0")
        traverse(node.right, prefix + "1")

    traverse(tree)
    return codes
```

Fig 3.D.1.3 Implementation code of Huffman coding

(Source: Author's Archive)

The next step involves encoding the gray levels using the binary representations derived from the Huffman coding tree. Each gray value is replaced by its corresponding Huffman code, which is obtained by traversing the tree structure. Additionally, the run-length count associated with each gray level in the RLE output is encoded separately using a binary representation.

```python
# Apply Huffman encoding to RLE list
def huffman_encode_rle(rle_list, huffman_codes):
    return [(huffman_codes[value], count) for value, count in rle_list]
```

Fig 3.D.1.4 Implementation code of Huffman coding in RLE

(Source: Author's Archive)

The result of applying Huffman encoding to the gray values and binary encoding (e.g., Fibonacci) to the run-length counts from Figure 3.B.1.2 is presented in the table below. Each pair represents a compressed form of (p, q), where:

The first element is the Huffman binary code for the gray value.

The second element is the normal binary encoding.

Gray value	frequencies	Huffman Gray Value	Binary frequencies
236	21	1	00010101
159	3	000	00000011
236	2	1	00000010
159	3	000	00000011
236	2	1	00000010
159	1	000	00000001
178	6	011	00000110
159	1	000	00000001
236	2	1	00000010
178	1	011	00000001
8	2	01001	00000010
178	2	011	00000010
8	1	01001	00000010
156	1	0101	00000001
236	2	1	00000010
178	3	011	00000011
236	2	001	00000010
178	1	011	00000001
168	1	01000	00000001
156	1	0101	00000001
236	3	1	00000011
178	1	011	00000001
104	3	001	00000011
156	2	0101	00000010
236	4	1	00000100
178	1	011	00000001
104	2	001	00000010
156	2	0101	00000010
236	5	1	00000101
104	2	001	00000010
236	16	1	00010000

E. Calculating Compression Ratio

To evaluate the efficiency of each compression method, it is necessary to compare the original file size before compression with the compressed file size after applying a given encoding technique. The compression ratio is calculated using the following formula:

$$Compression\ ratio\ (C_r) = \left(\frac{Compression\ size}{Original\ Size}\right) \times\ 100\ \%$$

In accordance with the Restrictions and Scope of Analysis, the bit-length of each grayscale pixel is defined as K = 8, meaning each gray value is represented using 8-bit binary. Based on Figure 3.B.1, the image contains 100 pixels. Therefore, the original file size is:

$$\text{Original Size} = 100 \times 8 = 800 \text{ bits}$$

1. Fibonacci Encoding Analysis

To determine the total size after compression using Fibonacci encoding, the following summation is used:

$$Cs_{Fib} = \sum_{i=1}^{length(RLE)} (length(fibonaciiEncoded)_i) + length(\ binary\ gray\ value_i))$$

$$Cs_{Fib} = 103\ + 248 = 351\ bits$$

$$Cr_{Fib} = \frac{351}{800} \times\ 100\% = 43.88\%$$

This result indicates that the image size was reduced by approximately 43.88% through Fibonacci-based encoding.

2. Huffman Encoding Analysis

Using Huffman coding, the compressed size is computed as:

$$Cs_{Huff} = \sum_{i=1}^{length(RLE)} (length(huffman)_i) + length(\ binary\ frequencies_i))$$

$$Cs_{Huf} = 85 + 248 = 333\ bits$$

$$Cr_{Huf} = \frac{333}{800} \times\ 100\% = 41.62\%$$

Thus, Huffman compression achieved a size reduction of approximately 41.6%.

3. Basic RLE with Fixed-Length Binary

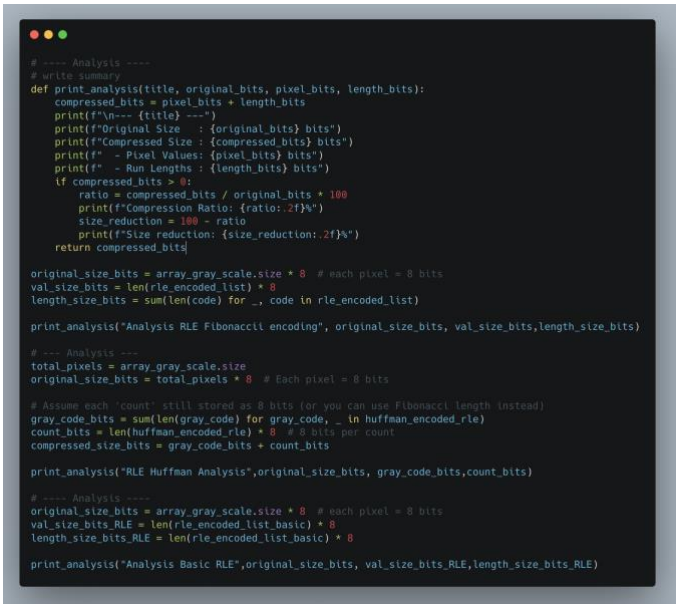
As a control, basic RLE is applied using standard binary representation for both the gray values and run-lengths:

$$Cs_N = \sum_{i=1}^{length(RLE)} (length(binary\ gray\ value)_i) + length(\ binary\ frequencies_i))$$

$$Cs_N = 248 + 248 = 496\ bits$$

$$Cr_N = \frac{496}{800} \times\ 100\% = 62.00\%$$

This shows that basic RLE alone provides a smaller reduction, shrinking the image size by 38%.



```
# ---- Analysis ----
# write summary
def print_analysis(title, original_bits, pixel_bits, length_bits):
    compressed_bits = pixel_bits + length_bits
    print(f"\n--- {title} ---")
    print(f"Original Size   : {original_bits} bits")
    print(f"Compressed Size : {compressed_bits} bits")
    print(f"  - Pixel Values: {pixel_bits} bits")
    print(f"  - Run Lengths : {length_bits} bits")
    if compressed_bits > 0:
        ratio = compressed_bits / original_bits * 100
        print(f"Compression Ratio: {ratio:.2f}%")
        size_reduction = 100 - ratio
        print(f"Size reduction: {size_reduction:.2f}%")
    return compressed_bits

original_size_bits = array_gray_scale.size * 8  # each pixel = 8 bits
val_size_bits = len(rle_encoded_list) * 8
length_size_bits = sum(len(code) for _, code in rle_encoded_list)

print_analysis("Analysis RLE Fibonaccii encoding", original_size_bits, val_size_bits,length_size_bits)

# --- Analysis ---
total_pixels = array_gray_scale.size
original_size_bits = total_pixels * 8  # Each pixel = 8 bits

# Assume each 'count' still stored as 8 bits (or you can use Fibonacci length instead)
gray_code_bits = sum(len(gray_code) for gray_code, _ in huffman_encoded_rle)
count_bits = len(huffman_encoded_rle) * 8  # 8 bits per count
compressed_size_bits = gray_code_bits + count_bits

print_analysis("RLE Huffman Analysis",original_size_bits, gray_code_bits,count_bits)

# ---- Analysis ----
original_size_bits = array_gray_scale.size * 8  # each pixel = 8 bits
val_size_bits_RLE = len(rle_encoded_list_basic) * 8
length_size_bits_RLE = len(rle_encoded_list_basic) * 8

print_analysis("Analysis Basic RLE",original_size_bits, val_size_bits_RLE,length_size_bits_RLE)
```

Fig 3.E.1.1 Implementation code to help calculate Cr

(Source: Author's Archive)

IV. Result And Further Analysis

After calculating all compression ratio and size reduction now can be summarizes into tabel bellow the performance of the three evaluated encoding techniques in terms of compressed size and compression ratio, based on an original grayscale image of 100 pixels (800 bits in uncompressed form):

Compression Method	Compressed Size (bits)	Compression Ratio (%)	Size Reduction (%)
Fibonacci Encoding	351	43.88%	56.12%
Huffman Encoding	333	41.62%	58.38%
Basic RLE (8-bit fixed)	496	62.00%	38.00%

Huffman encoding yielded the smallest compressed size (333 bits) and the highest size reduction (58.38%). This result aligns with Huffman coding's principle of assigning shorter binary codes to more frequent symbols, which significantly reduces redundancy in data with non-uniform frequency distributions.

Fibonacci encoding, while slightly less efficient than Huffman, still achieved a significant reduction (351 bits, 56.12% savings). Its strength lies in encoding numerical values

like run lengths using variable-length prefix-free codes without requiring a frequency table. This makes it beneficial in environments where maintaining symbol frequencies is costly or unnecessary.

Basic RLE using fixed 8-bit binary encoding for both pixel values and run-lengths resulted in the least efficient compression, with only a 38.00% reduction. While RLE is effective for sequences with long repeated values, it lacks the adaptability of variable-length encodings and tends to include overhead when repetition is not dominant.

Previous research [7] recommended the use of Run-Length Encoding (RLE) combined with Huffman coding to achieve higher compression efficiency, particularly in images dominated by repetitive grayscale intensity values. To assess the validity of this claim, three compression strategies—RLE with Fibonacci encoding, RLE with Huffman encoding, and basic RLE with fixed-length binary—were applied to a grayscale image consisting of 1,036,800 bits (corresponding to 129,600 pixels, each represented using 8 bits).

```
--- Analysis RLE Fibonaccii encoding ---
Original Size   : 1036800 bits
Compressed Size : 22778 bits
  - Pixel Values: 10088 bits
  - Run Lengths : 12690 bits
Compression Ratio: 2.20%
Size reduction: 97.80%

--- RLE Huffman Analysis ---
Original Size   : 1036800 bits
Compressed Size : 12490 bits
  - Pixel Values: 2402 bits
  - Run Lengths : 10088 bits
Compression Ratio: 1.20%
Size reduction: 98.80%

--- Analysis Basic RLE ---
Original Size   : 1036800 bits
Compressed Size : 20176 bits
  - Pixel Values: 10088 bits
  - Run Lengths : 10088 bits
Compression Ratio: 1.95%
Size reduction: 98.05%
```

The combination of RLE and Huffman encoding produced the most efficient compression, reducing the image size by 98.80% and achieving a compressed size of only 12,490 bits. This outcome demonstrates superior performance compared to RLE with Fibonacci encoding (97.80% reduction) and basic RLE (98.05% reduction).

These findings support the recommendation made in [7], confirming that Huffman encoding, when applied after RLE, significantly enhances compression efficiency. The improvement is attributed to Huffman's capacity to assign shorter codes to frequently occurring symbols, which complements the output of RLE by effectively compressing repeated patterns in grayscale images.

V. Summery And Recomandation

Across both experiments, Huffman encoding consistently provided superior compression results. This is attributed to its optimal use of variable-length prefix codes based on frequency distribution, making it especially effective after applying RLE, which flattens repeated pixel values. Fibonacci encoding, while slightly less efficient in terms of space savings, offers simplicity and deterministic decoding, making it suitable in systems where codebook transmission is impractical or forbidden.

Basic RLE using fixed 8-bit binary representations is the simplest to implement but suffers from a lack of adaptiveness to data frequency. Its performance is acceptable in highly uniform images but suboptimal otherwise.

Based on the above findings, Huffman encoding following RLE is the most efficient compression strategy for grayscale images in both small and large data settings. Fibonacci encoding may be considered a viable alternative where computational simplicity or prefix-free encoding is required. Basic RLE serves as a baseline method but is not recommended for scenarios where maximum compression is critical.

Future work is encouraged to explore fractal-based or hybrid compression approaches, particularly for images containing self-similar patterns, as they may provide even greater compression yields.

VI. Aknowledgment

The author extends heartfelt gratitude to Allah S.W.T for providing wisdom, perseverance, and opportunity to complete this paper successfully. Sincere appreciation is all extended to Mr. Dr. Ir. Rinaldi Munir, M.T., and Mr. Arrival Dwi Sentosa, S.Kom., M.T. as the lecturer of the IF1220 Discrete Mathematics course

VII. Appendix

This appendix contains supporting materials, including annotated source code for RLE, Fibonacci, and Huffman encoding, the grayscale test image, and sample compression outputs. It also provides compression ratio calculations and data tables used in the analysis. All resources can be accessed via the GitHub repository

https://github.com/HussainDzaki/Efficiency-Comparison-Between-RLE-Fibonacci-and-RLE-Huffman-Coding-in-Grayscale-Image-Compression/blob/main/README.md

REFERENCE

[1] Munir, Rinaldi. 2024. *Pengantar pemrosesan citra digital (bagian 1)* . https://informatika.stei.itb.ac.id/~rinaldi.munir/Citra/2024-2025/01-Pengantar-Pemrosesan-Citra-Digital-Bag1-2024.pdf (accessed on 18 June 2025)

[2] Munir, Rinaldi. 2024. *Pembentukan Citra dan Digitalisasi Citra*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Citra/2022-2023/03-Pembentukan-Citra-dan-Digitalisasi-Citra-2022. (accessed on 20 June 2025)

[3] Munir, Rinaldi. 2024. *Pemampatan Citra (bagian 1)*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Citra/2024-2025/25-Image-Compression-Bagian1-2024.pdf. (accessed on 20 June 2025)

[4] Munir, Rinaldi. 2024. *Deretan, Rekursim dan Relasi Rekurens (bagian 1)*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/10-Deretan,%20rekursi-dan-relasi-rekurens-(Bagian1)-2024.pdf. (accessed on 20 June 2025)

[5] GeeksForGeeks, 2023, *Fibonacci Coding*. https://www.geeksforgeeks.org/dsa/fibonacci-coding/. on 20 June 2025)

[6] Munir, Rinaldi. 2024. *Pohon (bagian 2)*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf. (accessed on 20 June 2025)

[7] Angkisan, Carlo. 2025. *Implementasi Algoritma Huffman untuk Optimasi Kompresi Data pada Penyimpanan Citra Digital.* https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/Makalah/Makalah-IF1220-Matdis-2024%20(70).pdf (accessed on 18 June 2025)

PERNYATAAN

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

Bandung, 20 Juni 2025

Dzaki Ahmad Al Hussainy, 13524084