# Image Compression Technique Using Huffman Algorithm and Number of Color Bins for Grayscale Images

Dzakwan Muhammad Khairan Putra Purnama - 13524145
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: dzakwan.mkpp@gmail.com, 13524145@std.stei.itb.ac.id

***Abstract* — In this study, we propose a grayscale image compression technique that integrates intensity quantization (color binning) with Huffman coding to achieve efficient lossless compression. The method begins by converting a color image to grayscale, followed by mapping pixel intensities to a reduced number of levels using a specified bin count. These quantized values are then encoded using Huffman coding, which assigns shorter binary codes to more frequent values. The experimental results demonstrate a clear trade-off between compression efficiency and image quality, as measured by the Structural Similarity Index (SSIM). With fewer bins (e.g., 2–4), the compression efficiency reaches over 85%, but at the cost of image quality. As bin counts increase, visual fidelity improves—SSIM values exceed 0.90 at 8 bins and above—yet the compression benefits decrease. The findings suggest that using 8 to 16 bins offers a practical balance, producing compressed images with excellent visual quality while maintaining relatively high compression efficiency. This approach is well-suited for applications requiring both storage optimization and image integrity, such as medical imaging and digital archiving.**

***Keywords — Image compression, Huffman coding, intensity quantization, SSIM.***

## I. Introduction

Digital image compression is an essential process in modern multimedia applications, enabling efficient storage and transmission of image data without significant loss of quality. For grayscale images, which contain only intensity information, compression techniques aim to reduce data redundancy while preserving visual details.

One of the most widely used lossless compression techniques is Huffman coding, with this encoding algorithm will assign shorter codes to more frequent symbols and longer codes to less frequent ones [1]. This method is particularly effective when combined with preprocessing steps that reduce the diversity of pixel values. On the other hand in grayscale images, the number of unique pixel intensity values (ranging from 0 to 255) can be too large and not efficient. Color binning or intensity quantization is a technique used to reduce the number of distinct gray levels by grouping some look like intensities into bins. This not only reduces the entropy of the image but also enhances the performance of the Huffman encoding stage by minimizing the number of symbols to encode [2]. By integrating color binning with Huffman coding, a more compact representation of grayscale images can be achieved. This approach is especially useful in applications requiring lossless compression such as medical imaging, document archiving, and satellite image processing.

Previous studies have successfully implemented the Huffman algorithm as a lossless compression technique for grayscale image data [3]. These implementations primarily focused on utilizing pixel intensity frequency distributions to generate efficient encoding schemes. However, they have not yet explored the integration of bin size as an additional factor in the compression process. In this research, we aim to enhance the existing approach by incorporating intensity binning, which groups similar pixel values into defined ranges (bins) before applying Huffman coding. This modification is expected to reduce the number of distinct symbols, potentially improving compression efficiency, especially for images with high intensity variability.

## II. Literature Review

### A. Digital Image

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

Fig. 1. Example 2D matrix representation of an image.

An image is a two-dimensional representation that can be mathematically modeled as a function *f*(*x*, *y*) (Fig. 1), where x and y represent spatial coordinates, and *f*(*x*, *y*) indicates the intensity of light at a particular point. Each element in the matrix is referred to as a pixel, which holds a specific intensity value. This value, known as the intensity level, determines the brightness of the pixel. A pixel with a higher intensity value appears brighter than one with a lower value [4].

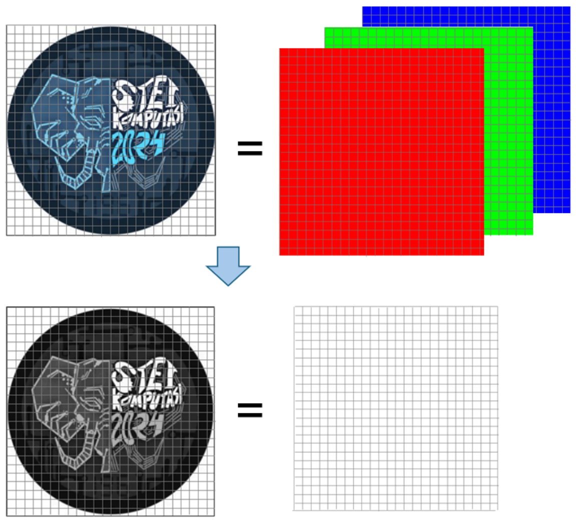


Fig. 2. RGB to grayscale with its matrix representation.

A color digital image is generally in RGB mode. An RGB image is a digital image composed of three primary color channels: Red, Green, and Blue, which together form the RGB color model. Each color channel has a bit depth of 8 bits, allowing each channel to represent 256 levels of color intensity, ranging from 0 to 255. The combination of intensities from the three channels produces a wide range of colors that make up a colored image.In each channel, a value of 255 represents the full intensity of that color, while a value of 0 indicates black. The blending of various intensity values across the three channels enables the creation of more complex colors in an RGB image. On the other hand a grayscale image is a digital image that represents pixel intensity values based on shades of gray without involving any other color information. In an 8-bit grayscale image, there are 256 levels of gray ranging from 0 (pure black) to 255 (pure white). Grayscale images are commonly used in digital image analysis because they simplify visual information into a single intensity value, making them easier to process and analyze in various image processing applications. They can be opened across different devices and operating systems. An RGB image can be converted into a grayscale (Fig.2) using Eq. 1. R, G, and B on Eq.1 represent the red, green, and blue color channel intensity values of each pixel, which is ranging from 0 to 255 and **y** is the resulting grayscale intensity value.

$$y = 0.22989 \times R + 0.5870 \times G + 0.1140 \times B$$

(Equation 1)

## B. *Huffman Algorithm*

Huffman coding is a lossless data compression technique developed by David A. Huffman in 1952. It is designed to create efficient binary codes based on the frequency of symbol occurrences in a dataset. The main idea is to reduce the size of data representation without compromising its integrity, making this method widely used in compressing text, images, and other multimedia files. The process begins with the construction of a Huffman tree, which serves as the foundation of the algorithm. This tree is built by repeatedly merging the two least frequent symbols into a new node, and this merging process continues until a single tree is formed. Each symbol in the tree is represented as a branch, with the binary code length inversely proportional to its frequency—more frequent symbols receive shorter codes, while less frequent symbols are assigned longer ones. This characteristic allows Huffman coding to achieve efficient compression.

Step-by-step procedures for constructing a Huffman Tree are as follows:

1. Count the frequency of each symbol in the data.
2. Sort the symbols in ascending order based on their frequencies.
3. Combine the two symbols with the lowest frequencies into a new node and reinsert it into the list.
4. Repeat the merging process until only one tree remains.
5. Assign binary labels to the tree's edges consistently: 0 for the left branch and 1 for the right.
6. The path from the root to each leaf node forms the unique Huffman code for that symbol.

The resulting binary codes are prefix-free, meaning no code is a prefix of another, ensuring unambiguous encoding and decoding. By using this approach, Huffman coding allows for effective data compression without any loss of information, which is why it remains one of the most popular techniques in various domains, especially in digital image compression.

## C. *Quantization Technique*

Quantization is a technique used to simplify continuous or high-resolution values into a limited set of discrete levels. On grayscale images, each pixel typically has an intensity value ranging from 0 to 255. The goal of quantization is to reduce the number of gray levels by grouping pixel intensities into several bins. The main purpose for our technique, this step is to reduce the number of unique grayscale levels before applying the Huffman coding algorithm. Procedures for quantization are as follows:

1. Choose the number of bins (bin_count).
2. Calculate bin-width boundary using Eq. 2.

$$bin\_width = \frac{max\_val - min\_val}{bin\_count - 1}$$

(Equation 2)

3. Determine each bin-boundary using Eq.3.

$$bins = [round(min\_val + i \times bin\_width)\ for\ i\ in\ range(bin\_count)]$$

(Equation 3)

4. Assign pixel values to the nearest bin. On this approach each pixel intensity is mapped to the closest representative value of a bin.

## *D. Structural Similarity Index (SSIM)*

SSIM is a perceptual metric that measures the similarity between two images. Unlike traditional methods like Mean Squared Error (MSE) or Peak Signal-to-Noise Ratio (PSNR), which only consider pixel-wise differences, SSIM aims to model how humans perceive visual similarity by evaluating structural information, luminance, and contrast. SSIM compares local patterns of pixel intensities that have been normalized for luminance and contrast. The SSIM value ranges between -1 and 1, where 1 indicates perfect structural similarity, 0 means no structural correlation, and Values less than 0 are rare and usually indicate very strong dissimilarity. SSIM is a continuous value, and commonly interpreted within certain qualitative categories [5] as shown as on Table 1.

TABLE I. SSIM QUALITATIVE CATEGORIES

| SSIM Range | Quality Interpretation |
|---|---|
| 0.90 – 1.00 | Excellent (Almost identical images) |
| 0.70 – 0.89 | Good (Visually similar with slight changes) |
| 0.50 – 0.69 | Fair (Noticeable differences) |
| 0.00 – 0.49 | Poor (Significant structural differences) |

The SSIM index between two image patches x and y is calculated as Eq. 4.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)\times(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)\times(\sigma_x^2 + \sigma_y^2 + C_2)}$$

(Equation 4)

Where:

- $\mu_x, \mu_y$ : the average of $x$ and $y$,
- $\sigma_x^2, \sigma_y^2$ : the variance of $x$ and $y$,
- $\sigma_{xy}$ : the covariance between $x$ and $y$,
- $C_1, C_2$ : constants to stabilize the division when the denominator is small.

## III. METHODOLOGY

The proposed image compression technique involves a sequence of preprocessing and encoding steps designed to reduce the storage size of a grayscale image using intensity quantization and Huffman coding as shown on Figure 3. The process begins with reading the input image and the desired number of quantization bins (bin_count). If the input image is in color, it is first converted to a grayscale image to simplify the data and reduce complexity. Once the grayscale image is obtained, the pixel intensities are quantized according to the specified number of bins. This step reduces the number of distinct intensity levels by grouping similar values, which in turn lowers the entropy of the data and prepares it for more efficient compression. Next, Huffman coding is applied to the quantized image. This algorithm assigns shorter binary codes to more frequent intensity values and longer codes to less frequent ones, producing a compressed bitstream with no information loss (lossless compression). Finally, the results are saved in two formats: the Huffman code dictionary is stored in a .json file (bit_code.json), while the encoded image data is stored as binary in a .bin file (bitdata.bin).

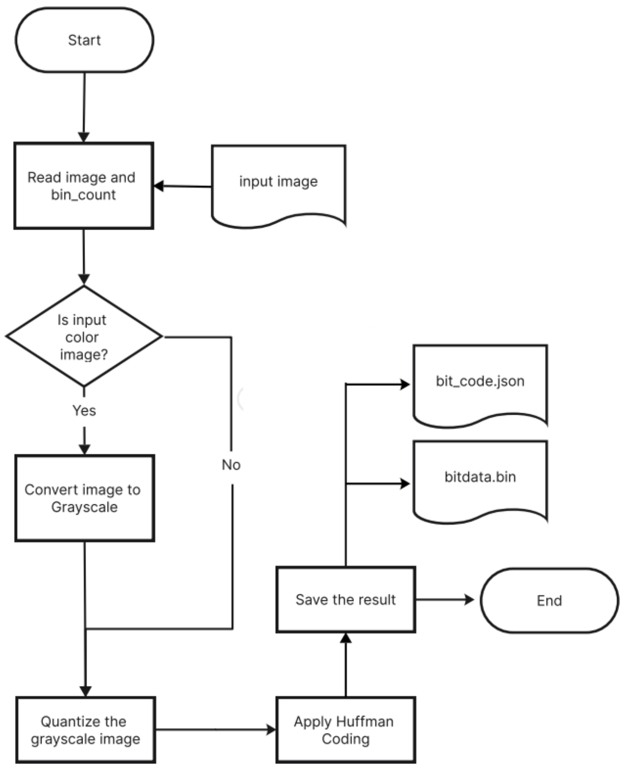


Fig. 3. Image compression procedure.

## *A. Environment Experiment*

The primary development machine used in this research have following specifications:

- Operating System: Windows 11 Pro 64-bit
- Processor: 11th Gen Intel Core i7-1185G7 @ 3.00GHz (8 logical CPUs)
- Memory: 32 GB RAM

This study was developed on a Windows 11 machine using the Anaconda platform to manage packages and dependencies efficiently. The implementation was carried out using Python version 3.8.5, with Jupyter Notebook as the interactive environment for coding, visualization, and result analysis. This setup provides flexibility, ease of debugging, and seamless integration with data science libraries.

## *B. Image Data*

The image data used in this study is a color image with a resolution of 500 × 502 pixels, representing the logo of the School of Electrical Engineering and Informatics, Computational Division (STEI-K), Class of 2024 (Figure 4).

Furthermore, eight different values of bin_count were tested: 2, 4, 8, 16, 32, 64, 128, and 256.

Fig. 4. Image as data.

## C. *Convert Image to Grayscale*

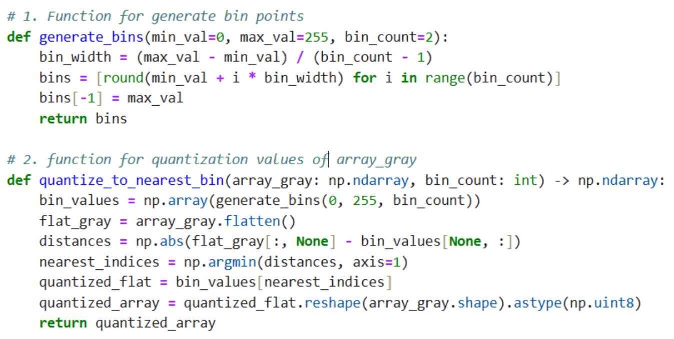
Preprocessing step in our procedures is standard conversion of the color image to grayscale (if necessary when the input is a color image). With this step we guarantee the data for the next step is grayscale image. To calculate gray scale intensity (y) value we used Equation 1. For example if we have a pixel with red intensity (R) = 50, green (G)=75, and blue (B)=100 we will obtain y as 67.

$$y = 0.22989 \times 50 + 0.5870 \times 75 + 0.1140 \times 100$$
$$y = 66.91 \approx 67$$

## D. *Quantization Grayscale Levels*

This procedure transforms grayscale intensity quantization by mapping each pixel value in the image to its nearest bin representative value. This is useful for reducing the number of gray levels in an image, which simplifies data and prepares it for efficient compression. For example if we have bit_count = 4 then we want to calculate the value representation for grayscale level 100, so the calculation would be:

- Calculate bin-width boundary using Eq. 2.
$$bin\_width = \frac{255 - 0}{4 - 1} = 85$$
- Determine each bin-boundary using Eq.3 and we obtained four values as bin points.
$$bins = [0, 85, 170, 255]$$
- Assign pixel values to the nearest bin point. In this example we have 100 as grayscale level, so the nearest bin point is 85.



```python
# 1. Function for generate bin points
def generate_bins(min_val=0, max_val=255, bin_count=2):
    bin_width = (max_val - min_val) / (bin_count - 1)
    bins = [round(min_val + i * bin_width) for i in range(bin_count)]
    bins[-1] = max_val
    return bins

# 2. function for quantization values of array_gray
def quantize_to_nearest_bin(array_gray: np.ndarray, bin_count: int) -> np.ndarray:
    bin_values = np.array(generate_bins(0, 255, bin_count))
    flat_gray = array_gray.flatten()
    distances = np.abs(flat_gray[:, None] - bin_values[None, :])
    nearest_indices = np.argmin(distances, axis=1)
    quantized_flat = bin_values[nearest_indices]
    quantized_array = quantized_flat.reshape(array_gray.shape).astype(np.uint8)
    return quantized_array
```

Fig. 5. Python implementation for quantization procedure.

## E. *Huffman Coding Application*

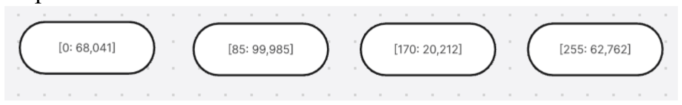
As mentioned earlier, in our experiment we used eight different values of bin_count, but for the sake of simplicity, we will use bin_count = 4 for the construction of the Huffman tree illustration. When bin_count = 4, the pixel values are represented only by one of the four values: 0, 85, 170, or 255. In the image data we used, out of a total of 251,000 pixels, the frequency of each bin shown on Table 2.

TABLE II. FREQUENCY FOR EACH BIN

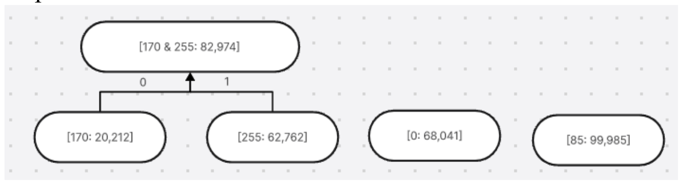
| **Bin** | 0 | 85 | 170 | 255 |
|---|---|---|---|---|
| **Freq.** | 68,041 | 99,985 | 20,212 | 62,762 |

Step by step construction of huffman tree as follow:

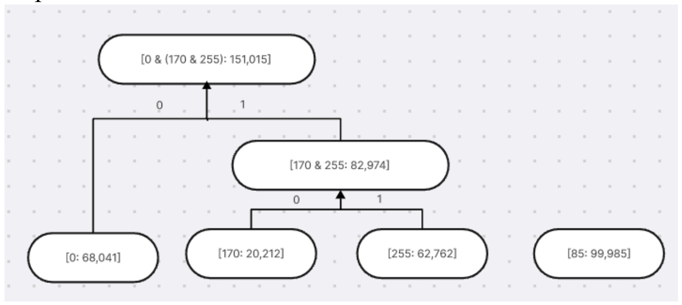
Step 1:



Step 2:



Step 3:



Step 4:

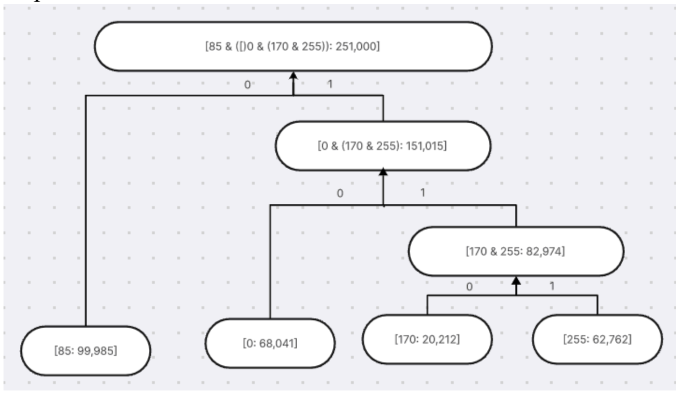


Fig. 6. Huffman tree construction step by step (bit_count=4).

As shown on Figure 6, the Huffman tree construction begins by initializing each grayscale bin value as a separate node, sorted by frequency (Step 1). In Step 2, the two lowest-frequency nodes (170 and 255) are merged into a new node with a combined frequency. Step 3 continues by merging this new node with the next lowest node (0), forming a larger subtree. Finally, in Step 4, the remaining two nodes bin 85 and the subtree are merged to form the root node. This process results in a binary tree where shorter codes are assigned to more frequent values, enabling efficient and lossless data compression.

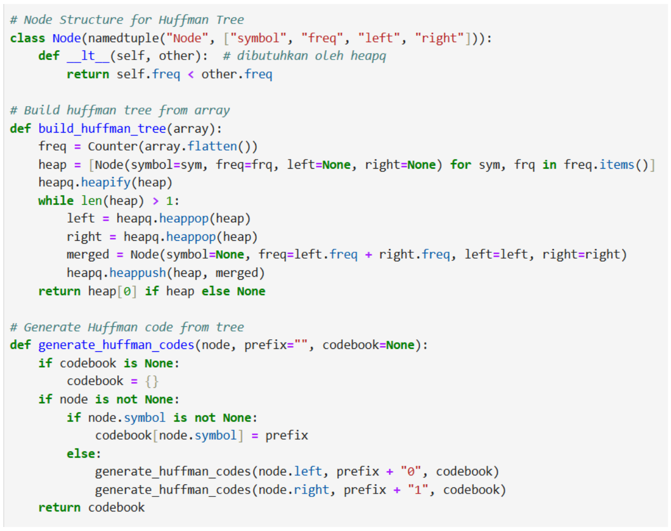
To evaluate the performance of the Huffman coding process, we calculated both the compression ratio and compression efficiency. Initially, the original grayscale image consisted of 251,000 pixels, each stored using 8 bits, resulting in a total uncompressed size of 2,008,000 bits. After applying Huffman coding based on the quantized bin frequencies, the total number of bits required for the compressed image was 484,989 bits.

$$\begin{aligned} bits\_comp &= (99,985\times1) + (68,041\times2) + (20,212\times3) + (62,762\times3)\ bits \\ bits\_comp &= 99,985 + 136,082 + 60,636 + 188,286\ bits \\ &= 484,989\ bits \end{aligned}$$

The compression ratio is calculated by dividing the original size by the compressed size, yielding a ratio of approximately 4.14:1. Meanwhile, the compression efficiency, which represents the percentage of data reduction, is obtained using the formula:

$$eff = \left(1 - \frac{bit\_comp}{bit\_original}\right) \times 100\% = \left(1 - \frac{484,989}{2,008,000}\right) \times 100\%$$
$$= 75.85\%$$

As a result we obtained an efficiency of about 75.85%. These results indicate that Huffman coding provides significant reduction in data size while preserving the original information. Three main functions used for constructing the Huffman Tree are implemented in Python, as shown in Figure 7.



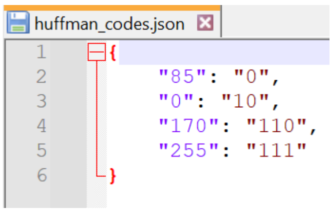
```python
# Node Structure for Huffman Tree
class Node(namedtuple("Node", ["symbol", "freq", "left", "right"])):
    def __lt__(self, other):  # dibutuhkan oleh heapq
        return self.freq < other.freq

# Build huffman tree from array
def build_huffman_tree(array):
    freq = Counter(array.flatten())
    heap = [Node(symbol=sym, freq=frq, left=None, right=None) for sym, frq in freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(symbol=None, freq=left.freq + right.freq, left=left, right=right)
        heapq.heappush(heap, merged)
    return heap[0] if heap else None

# Generate Huffman code from tree
def generate_huffman_codes(node, prefix="", codebook=None):
    if codebook is None:
        codebook = {}
    if node is not None:
        if node.symbol is not None:
            codebook[node.symbol] = prefix
        else:
            generate_huffman_codes(node.left, prefix + "0", codebook)
            generate_huffman_codes(node.right, prefix + "1", codebook)
    return codebook
```

Fig. 7. Huffman coding implementation on Python.

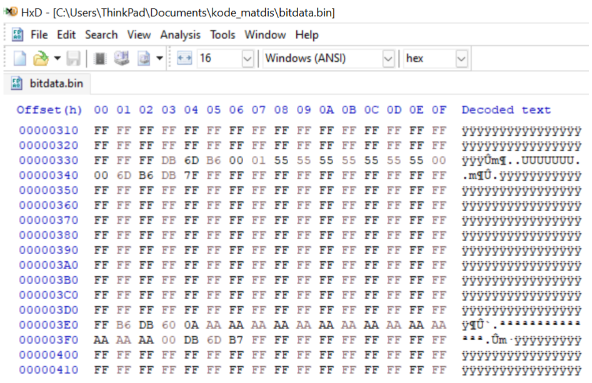
## F. Save The Compression Result



huffman_codes.json

```json
{
    "85": "0",
    "0": "10",
    "170": "110",
    "255": "111"
}
```

Fig. 8. Huffman code on .json.



HxD - [C:\Users\ThinkPad\Documents\kode_matdis\bitdata.bin]

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000310  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000320  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000330  FF FF FF DB 6D B6 00 01 55 55 55 55 55 55 55 00  ÿÿÿÛm¶..UUUUUUU.
00000340  00 6D B6 DB 7F FF FF FF FF FF FF FF FF FF FF FF  .m¶Û.ÿÿÿÿÿÿÿÿÿÿÿ
00000350  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000360  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000370  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000380  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000390  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000003A0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000003B0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000003C0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000003D0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000003E0  FF B6 DB 60 0A AA AA AA AA AA AA AA AA AA AA AA  ÿ¶Û`.ªªªªªªªªªªª
000003F0  AA AA AA 00 DB 6D B7 FF FF FF FF FF FF FF FF FF  ªªª.Ûm·ÿÿÿÿÿÿÿÿÿ
00000400  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000410  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Fig. 9. .bin contents as bit stream data of encoding result

The final result of the compression procedure is stored in two separate output files: a .json file and a .bin file. The .json file contains the Huffman codes assigned to each quantized bin value, serving as a reference or lookup table during the decoding process (Figure 8). Meanwhile, the .bin file stores the encoded bitstream generated from the quantized grayscale array using the Huffman codes (Figure 9). This separation allows for efficient data storage and easy reconstruction of the original image during decompression.

## IV. Result And Discussion

The compression evaluation was conducted using an image with an original size of 2,008,000 bits and varying the number of bins from 2 to 256. The experiment aimed to assess how quantization levels (via bin counts) affect compression performance and image quality, measured by Compression Efficiency and SSIM (Structural Similarity Index).

TABLE III. Experiment Results

| No | Parameters | | Evaluation Results | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Original Size (bit) | Number of Bins | Compressed Size (bit) | Bit Data Compression Ratio | Bit Data Efficiency (%) | Huffman Code File Size (bit) | File Compression Size Total | File Compression Efficiency | SSIM |
| 1 | 2,008,000 | 2 | 251,000.00 | 8.000 | 87.500 | 280 | 251,280 | 87.486 | 0.341 |
| 2 | | 4 | 484,989.00 | 4.140 | 75.850 | 584 | 485,573 | 75.818 | 0.656 |
| 3 | | 8 | 567,569.00 | 3.538 | 71.730 | 1,296 | 568,865 | 71.670 | 0.904 |
| 4 | | 16 | 701,172.00 | 2.864 | 65.080 | 2,528 | 703,700 | 64.955 | 0.969 |
| 5 | | 32 | 889,747.00 | 2.257 | 55.690 | 5,328 | 895,075 | 55.425 | 0.985 |
| 6 | | 64 | 1,059,775.00 | 1.895 | 47.220 | 11,216 | 1,070,991 | 46.664 | 0.994 |
| 7 | | 128 | 1,262,950.00 | 1.590 | 37.100 | 23,184 | 1,286,134 | 35.950 | 0.998 |
| 8 | | 256 | 1,453,431.00 | 1.382 | 27.620 | 48,104 | 1,501,535 | 25.222 | 1.000 |

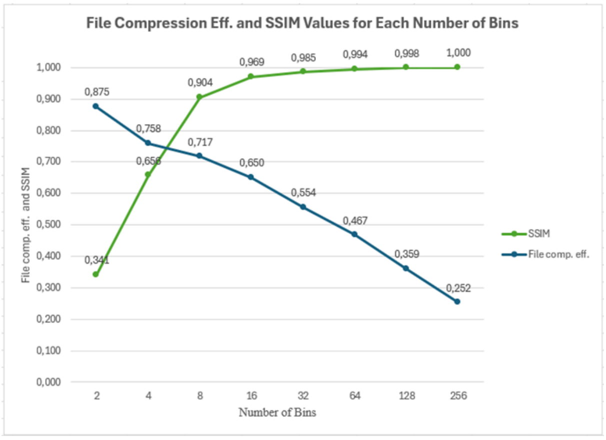


Fig. 10. File Compression Eff. and SSIM Values for each number of bins.

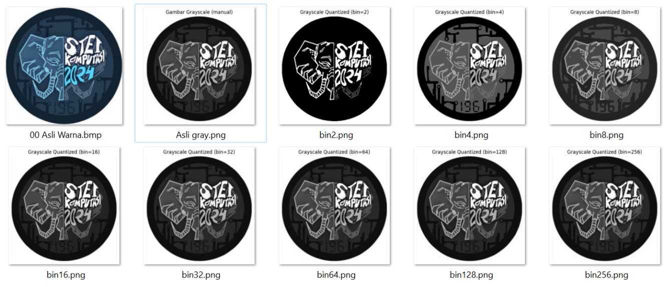


Fig. 11. Image visualization for original image and various bin count.

Table III and Figure 10 present the evaluation results of compression performance using various bin counts ranging from 2 to 256. As the number of bins increases, the Structural Similarity Index (SSIM) also improves, indicating better visual fidelity to the original grayscale image. For example, with only 2 bins, the SSIM score is 0.341—classified as Poor—while increasing the bin count to 4 and 8 raises the SSIM to 0.656 (Fair) and 0.904 (Good), respectively. At 16 bins and beyond, the SSIM enters the Excellent range, reaching 0.969 at 16, and plateauing near perfection (1.000) at 256 bins, showing almost no visible difference compared to the original. This quality improvement, however, comes at the expense of compression efficiency. At 2 bins, the system achieves the highest compression efficiency of 87.49%, compressing the 2,008,000-bit image down to only 251,000 bits. As the bin count increases, the compressed size also grows, reducing efficiency. At 256 bins, the compression ratio drops to 1.38, and the efficiency falls to 25.22%.

Figure 10 clearly illustrates this trade-off with two opposing curves: SSIM (green) increases rapidly and then stabilizes, while compression efficiency (blue) steadily declines. The most notable SSIM gain occurs between 2 and 16 bins, after which additional bins yield diminishing perceptual returns. Meanwhile, efficiency continues to decline with each increase in bin count due to the rising number of unique intensity levels, which limits the effectiveness of Huffman coding.

Figure 11 provides a visual comparison of the quantized images at each bin level. At lower bin counts (e.g., 2 and 4), images appear heavily posterized, with significant loss of detail and tonal variation, matching their low SSIM scores. As the bin count increases, the visual quality improves, and by 128 or 256 bins, the images become nearly indistinguishable from the original. In conclusion, the results highlight a fundamental trade-off in grayscale image compression using quantization and Huffman coding. Low bin counts yield smaller file sizes but degrade visual quality, while high bin counts preserve detail but reduce compression benefits. For applications requiring a balance between size and quality, using 8 to 16 bins is recommended, offering SSIM values above 0.90 (excellent) while maintaining relatively good compression ratios.

## V. Conclusion And Suggestions

This paper demonstrates the effectiveness of combining intensity quantization with Huffman coding for grayscale image compression. The experimental results highlight a significant trade-off between compression efficiency and image quality. Lower bin counts drastically reduce the file size with compression efficiencies reaching up to 87.49%, but at the expense of visual quality, indicated by lower SSIM values. Conversely, higher bin counts preserve more image details but result in reduced compression efficiency. Based on the analysis, the optimal range for balancing quality and compression lies between 8 to 16 bins, where SSIM values remain above 0.90 while maintaining acceptable compression rates. This range is ideal for applications that demand both compact storage and high image fidelity.

Suggestions for future research include exploring adaptive binning techniques that dynamically adjust bin ranges based on local image characteristics, as well as combining this approach with other lossless or near-lossless algorithms such as Run-Length Encoding or Arithmetic Coding for even better performance. Additionally, testing the technique on a broader set of images from various domains—such as medical scans, satellite photos, and handwritten documents—can further validate its general applicability and robustness.

## VI. Appendix

The complete source code for this paper can be accessed through the following GitHub repository link:
https://github.com/dzakwanmkpp/matematikaDikrit.git

## VII. Acknowledgment or Gratitude

The author expresses gratitude to God Almighty for His abundant blessings and grace throughout the process of writing this paper, enabling its timely completion. Appreciation is also extended to Mr. Dr. Ir. Rinaldi Munir, M.T., lecturer of the IF1220 Discrete Mathematics course for class K01, for his guidance and valuable knowledge. He also provided a website as a learning resource for this course, which greatly facilitated a deeper understanding of the material. Furthermore, heartfelt thanks are conveyed to the author’s parents for their unwavering support, encouragement, and prayers, which have been a source of motivation and strength in every step of the journey.

## References

[1] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098–1101. doi:10.1109/JRPROC.1952.273898

[2] Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing (4th ed.). Pearson. (See Chapter 8: Image Compression)

[3] Angkisan, C. (2024). Implementasi Algoritma Huffman untuk Optimasi Kompresi Data pada Penyimpanan Citra Digital. Makalah-IF1220-Matdis-2024(70). Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/Makalah/Makalah-IF1220-Matdis-2024%20(70).pdf [accessed: 12 June 2025]

[4] Jain, A. K. (1989). Fundamentals of Digital Image Processing. Prentice-Hall.

[5] Hore, A., & Ziou, D. (2010). Image quality metrics: PSNR vs. SSIM. 20th International Conference on Pattern Recognition (ICPR), pp. 2366–2369. DOI:10.1109/ICPR.2010.579

## STATEMENT

I hereby declare that the paper I have written is entirely my own work. It is not an adaptation, translation, or copy of someone else's paper, and it does not contain any elements of plagiarism.

Bandung, 20 June 2025

[illegible]

Dzakwan Muhammad Khairan P. P. - 13524145