RAMAIAH INSTITUTE OF TECHNOLOGY, BANGALORE – 560054
(Autonomous Institute, Affiliated to VTU) Department of Computer

Science & Engineering

20-Mark Component Report

On

ChatGPT based AssignmentImage

Compression

CS35: Discrete Mathematical Structures

Darshan N	**1MS23CS415-T**
Gokulnath S	**1MS23CS409-T**

Under the Guidance

Mamatha A

Ramaiah Institute of Technology

(Autonomous Institute, Affiliated to VTU)
MSR Nagar, MSRIT Post, Bangalore-560054

November 2023 - March 2024

RAMAIAH INSTITUTE OF TECHNOLOGY, BANGALORE – 560054
(Autonomous Institute, Affiliated to VTU)

Department of Computer Science & Engineering

Evaluation Report

Title of the Case Study: Image Compression

Team Member Details		
Sl. No.	USN	Name
1.	1MS23CS409-T	Gokulnath S
2.	1MS23CS415-T	Darshan N

SL No.	Component	Maximum Marks	Marks Obtained
1	Demonstration	10	
2	Presentation	05	
3	Report	05	
	Total Marks	20	

Signature of the Student **Signature of the Faculty**

Signature of Head of the Department

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Image compression is a crucial step in the field of image processing. It allows us to reduce the size of digital images while retaining as much quality as possible. Let’s delve into the details:

1.1 What is Image Compression?

Image compression refers to the process of **reducing the cost** of storing or transmitting digital images.

By applying various techniques, we aim to make images more compact without compromising their visual quality.

1.2 Why Do We Need Image Compression?

Consider a black and white image that has a resolution of 1000*1000 and each pixel uses 8 bits to represent the intensity. So the total no of bits required = 1000*1000*8 = 80, 00,000 bits per image.

As we see just to store a 3-sec video we need so many bits which is very huge. So, we need a way to have proper representation as well to store the information about the image in a minimum no of bits without losing the character of the image. Thus, image compression plays an important role.

1.3 Types of Image Compression:

1.3.1 Lossless Compression:

In **lossless compression**, the goal is to resize the images into a smaller version without compromising image quality.

1.3.2 Lossy compression

Lossy compression reduces the image size by selectively removing non-essential parts.

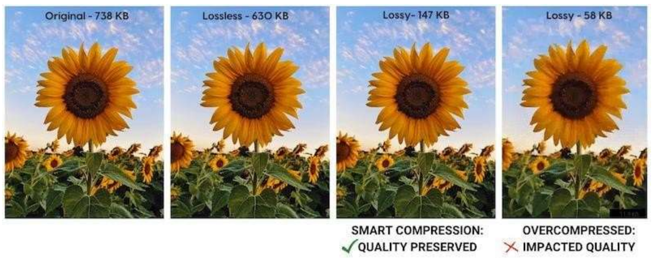


Fig 1.1 lossless and lossy compression

1.4Huffman coding algorithm.

Named after its creator, **David A. Huffman**, this algorithm aims to represent data more efficiently by assigning shorter codes to frequently occurring symbols.

One popular technique for image compression is the Huffman coding algorithm.

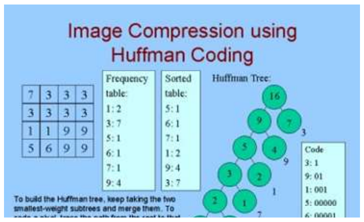


Fig 1.2 Huffman coding algorithm

1.4.1 Benefits

Reduced Storage: Compressed images occupy less space, making them ideal for storage.
Faster Transmission: Smaller image files can be transmitted more quickly over networks.
Lossless and Lossy Options: While JPEG uses lossy compression, Huffman coding itself can be used for lossless compression too.

1.4.2 Here's how Huffman coding works:

Frequency Analysis: Initially, the algorithm analyzes the input image to determine the frequency of occurrence of each pixel value (intensity level). This step helps identify which pixel values are more common and which are less frequent.

Build Huffman Tree: Based on the frequency information, the algorithm constructs a Huffman tree. This binary tree assigns shorter codes (bit sequences) to more frequent pixel values and longer codes to less frequent ones. The tree is built in such a way that the most common pixel values have the shortest codes.

Generate Huffman Codes: By traversing the Huffman tree, we assign unique binary codes to each pixel value. These codes are used to represent the pixel values in the compressed image.

Encode the Image: During compression, each pixel value in the original image is replaced with its corresponding Huffman code. The resulting compressed image contains these shorter codes instead of the original pixel values.

Decoding: During decompression, the reverse process occurs. The Huffman tree is reconstructed, and the compressed codes are converted back to pixel values.

CHAPTER 2

DMS CONCEPTS APPLIED

In the realm of **discrete mathematics**, **image compression** plays a significant role. Let's explore how mathematical concepts intersect with image compression:

2.1 Image Representation:

In discrete mathematics, an image is often represented as a **vector of pixels**. Each pixel corresponds to a fixed number of bits, determining color intensity (grayscale for black and white images or RGB channels for colored images).
For instance, consider a black and white image with a resolution of 1000x1000 pixels, where each pixel uses 8 bits. The total number of bits required for this image is 80,00,000 bits.
When dealing with videos (e.g., 30 frames per second), the total bits for a 3-second video become a staggering 720,000,000 bits.
Image compression aims to represent this information using fewer bits without losing essential image characteristics

2.2 Transforms in Image Processing:

Transforms are mathematical functions that map from one vector space to another. They change the shape or representation of a function while preserving its fundamental relationship.
In image processing, we transform an image from the **spatial domain** (pixel coordinates) to the **frequency domain**.
Examples of transforms include the **Discrete Cosine Transform (DCT)** and the **Singular Value Decomposition (SVD)**.
These transforms allow us to infer more information about the image in the newly projected vector space

2.3 Discrete Cosine Transform (DCT):

DCT is widely used in image compression.
It converts digital image data from the spatial domain to the frequency domain.
By analyzing frequency components, DCT helps represent the image efficiently while minimizing loss of quality

2.4 Singular Value Decomposition (SVD):

SVD breaks down an image matrix into three matrices: **U**, **Σ** (diagonal matrix), and **V^T** (transpose of V).
It allows us to approximate the original image using a smaller number of singular values (components).

CHAPTER 3

OVERALL VIEW OF THE CASE STUDY

Image compression finds applications in various domains due to its ability to reduce file size while preserving essential information. Here are some notable use cases:

3.1 Television Broadcasting:

Image compression ensures efficient transmission of TV signals, allowing broadcasters to deliver high-quality content to viewers.
It minimizes bandwidth requirements while maintaining acceptable image fidelity

3.2 Remote Sensing via Satellite:

Satellite imagery plays a crucial role in weather forecasting, environmental monitoring, and disaster management.
Image compression optimizes data transmission from satellites to ground stations, enhancing real-time analysis

3.3 Military Communication Systems (Radars):

Military radars rely on compressed images for surveillance, target detection, and reconnaissance.
Efficient compression enables faster data exchange and better situational awareness

3.4 Computer Communication Systems:

Image compression enhances multimedia communication over networks.
Applications include video streaming, online gaming, and social media platforms.

3.5 Geological Surveys:

Geological studies involve analyzing large image datasets.
Compression helps manage data storage and facilitates geological mapping and analysis

3.6Weather Reporting Applications:

Weather stations collect and transmit images for forecasting and climate monitoring.
Image compression optimizes data transfer, especially in remote or resource-constrained areas

CHAPTER 4

IMPLEMENTATION WITH RESULTS

4.1 Read the Image:

First, read the input image into a 2D array (e.g., from a .bmp file).
You can use libraries like **PIL (Pillow)** or **OpenCV** to load the image.

```
from PIL import Image
# Load the image
image_path = 'path/to/your/image.bmp'
image = Image.open(image_path)
pixels = list(image.getdata())
width, height = image.size
```

4.2 Create a Histogram:

Calculate the frequency of each pixel intensity value in the image.
This histogram will help us determine the probabilities for each symbol (pixel intensity).

```
class HuffmanNode:
    def_init_(self, intensity, freq):
        self.intensity = intensity
        self.freq = freq
        self.left = None
        self.right = None
```

4.3 Build the Huffman Tree:

Combine the two lowest probability leaf nodes into a new node.
Sort the nodes based on their new probability values.
Repeat this process until we get a single root node with probability 1.0.
The resulting binary tree is our **Huffman tree**.

```
def build_huffman_tree(hist):
    heap = [(freq, HuffmanNode(intensity, freq)) for intensity, freq
in hist.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        freq1, node1 = heapq.heappop(heap)
        freq2, node2 = heapq.heappop(heap)
        merged_node = HuffmanNode(None, freq1 + freq2)
        merged_node.left, merged_node.right = node1, node2
        heapq.heappush(heap, (merged_node.freq, merged_node))
    return heap[0][1]
```

4.4 **Assign Huffman Codes**:

Backtrack from the root node to each leaf node.
Assign ‘0’ or ‘1’ to each intermediate node based on the path taken.
These codes represent the Huffman-encoded values for each pixel intensity.

```
def generate_huffman_codes(root, code="", huffman_codes={}):
    if root:
        if root.intensity is not None:
            huffman_codes[root.intensity] = code
        generate_huffman_codes(root.left, code + "0", huffman_codes)
        generate_huffman_codes(root.right, code + "1", huffman_codes)
```

4.5 **Encode the Image**:

Replace each pixel intensity value with its corresponding Huffman code.
The entire image is now represented using these variable-length codes.

```
def encode_image(image, huffman_codes):
    encoded_image = ""
    for row in image:
        for intensity in row:
            encoded_image += huffman_codes[intensity]
    return encoded_image
```

USER INPUT CODE

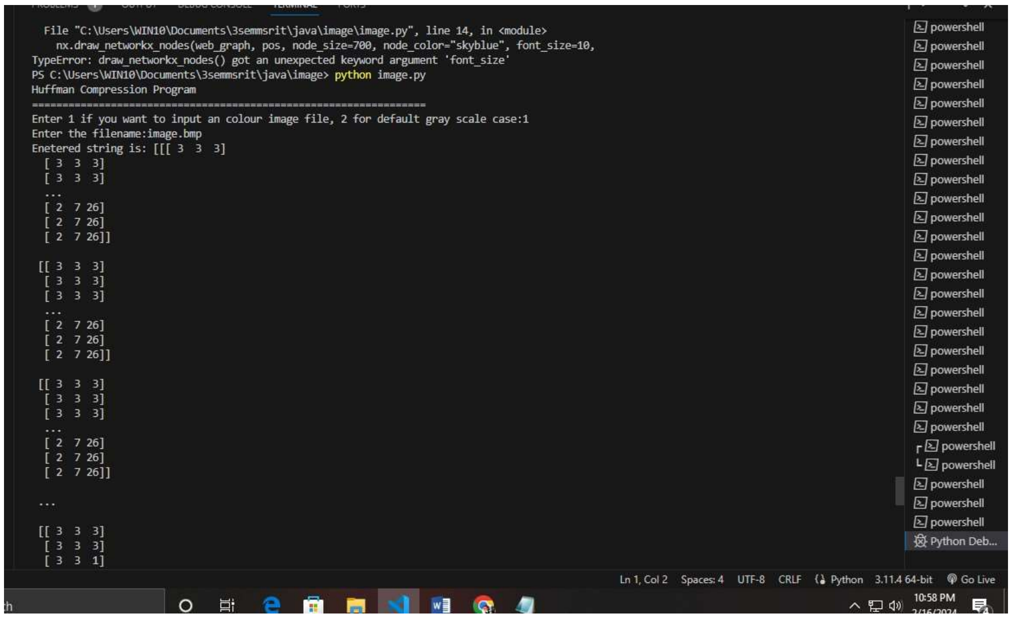
```
import re
import numpy as np
from PIL import Image
print("Huffman Compression Program")
print("==========================================================
=========")
h = int(input("Enter 1 if you want to input an colour image file, 2 for default gray
scale case:"))
if h == 1:
   file = input("Enter the filename:")
   my_string = np.asarray(Image.open(file),np.uint8)
   shape = my_string.shape
   a = my_string
   print ("Enetered string is:",my_string)
   my_string = str(my_string.tolist())
elif h == 2:
   array = np.arange(0, 737280, 1, np.uint8)
   my_string = np.reshape(array, (1024, 720))
   print ("Enetered string is:",my_string)
   a = my_string
```

```
    my_string = str(my_string.tolist())

else:
    print("You entered invalid input")                    # taking user input

letters = []
only_letters = []
for letter in my_string:
    if letter not in letters:
        frequency = my_string.count(letter)             #frequency of each letter repetition
        letters.append(frequency)
        letters.append(letter)
        only_letters.append(letter)

nodes = []
while len(letters) > 0:
    nodes.append(letters[0:2])
    letters = letters[2:]                              # sorting according to frequency
nodes.sort()
huffman_tree = []
huffman_tree.append(nodes)                             #Make each unique character as a leaf
node

def combine_nodes(nodes):
    pos = 0
    newnode = []
    if len(nodes) > 1:
        nodes.sort()
        nodes[pos].append("1")                      # assigning values 1 and 0
        nodes[pos+1].append("0")
        combined_node1 = (nodes[pos] [0] + nodes[pos+1] [0])
        combined_node2 = (nodes[pos] [1] + nodes[pos+1] [1]) # combining the nodes
to generate pathways
        newnode.append(combined_node1)
        newnode.append(combined_node2)
        newnodes=[]
        newnodes.append(newnode)
        newnodes = newnodes + nodes[2:]
        nodes = newnodes
        huffman_tree.append(nodes)
        combine_nodes(nodes)
    return huffman_tree                                # huffman tree generation

newnodes = combine_nodes(nodes)

huffman_tree.sort(reverse = True)
print("Huffman tree with merged pathways:")

checklist = []
```

```
for level in huffman_tree:
    for node in level:
        if node not in checklist:
            checklist.append(node)
        else:
            level.remove(node)
count = 0
for level in huffman_tree:
    print("Level", count,":",level)             #print huffman tree
    count+=1
print()

letter_binary = []
if len(only_letters) == 1:
    lettercode = [only_letters[0], "0"]
    letter_binary.append(letter_code*len(my_string))
else:
    for letter in only_letters:
        code =""
        for node in checklist:
            if len (node)>2 and letter in node[1]:           #genrating binary code
                code = code + node[2]
        lettercode =[letter,code]
        letter_binary.append(lettercode)
print(letter_binary)
print("Binary code generated:")
for letter in letter_binary:
    print(letter[0], letter[1])

bitstring =""
for character in my_string:
    for item in letter_binary:
        if character in item:
            bitstring = bitstring + item[1]
binary ="0b"+bitstring
print("Your message as binary is:")
                                        # binary code generated

uncompressed_file_size = len(my_string)*7
compressed_file_size = len(binary)-2
print("Your original file size was", uncompressed_file_size,"bits. The compressed
size is:",compressed_file_size)
print("This is a saving of ",uncompressed_file_size-compressed_file_size,"bits")
output = open("compressed.txt","w+")
print("Compressed file generated as compressed.txt")
output = open("compressed.txt","w+")
print("Decoding ......")
output.write(bitstring)
```

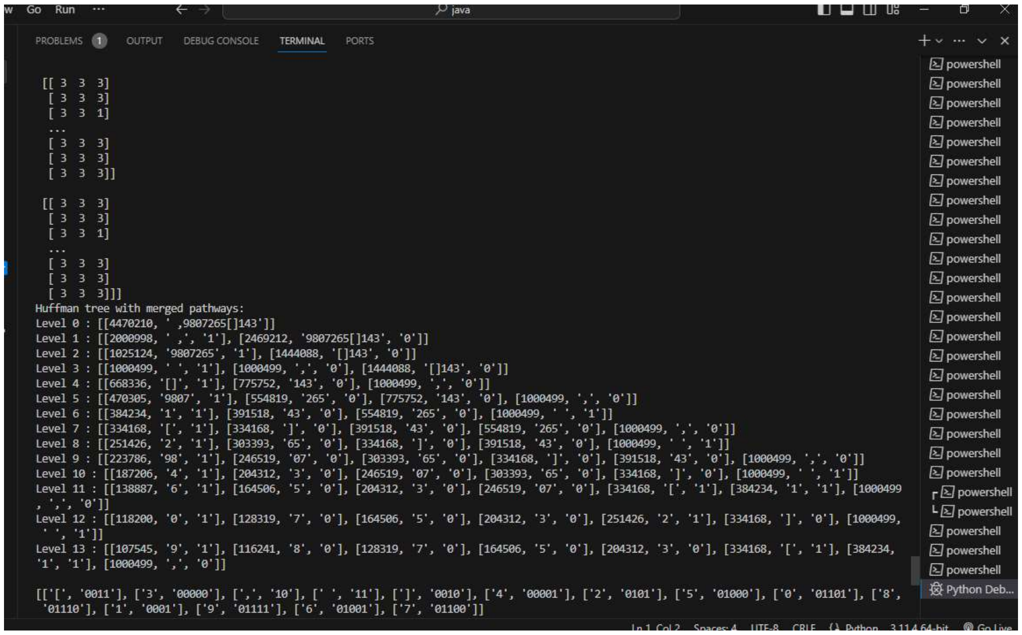
```
bitstring = str(binary[2:])
uncompressed_string =""
code =""
for digit in bitstring:
    code = code+digit
    pos=0                                    #iterating and decoding
    for letter in letter_binary:
        if code ==letter[1]:
            uncompressed_string=uncompressed_string+letter_binary[pos] [0]
            code=""
        pos+=1

print("Your UNCOMPRESSED data is:")
if h == 1:
    temp = re.findall(r'\d+', uncompressed_string)
    res = list(map(int, temp))
    res = np.array(res)
    res = res.astype(np.uint8)
    res = np.reshape(res, shape)
    print(res)
    print("Observe the shapes and input and output arrays are matching or not")
    print("Input image dimensions:",shape)
    print("Output image dimensions:",res.shape)
    data = Image.fromarray(res)
    data.save('uncompressed.png')
    if a.all() == res.all():
        print("Success")
if h == 2:
    temp = re.findall(r'\d+', uncompressed_string)
    res = list(map(int, temp))
    print(res)
    res = np.array(res)
    res = res.astype(np.uint8)
    res = np.reshape(res, (1024, 720))
    print(res)
    data = Image.fromarray(res)
    data.save('uncompressed.png')
    print("Success")
```

RESULTS



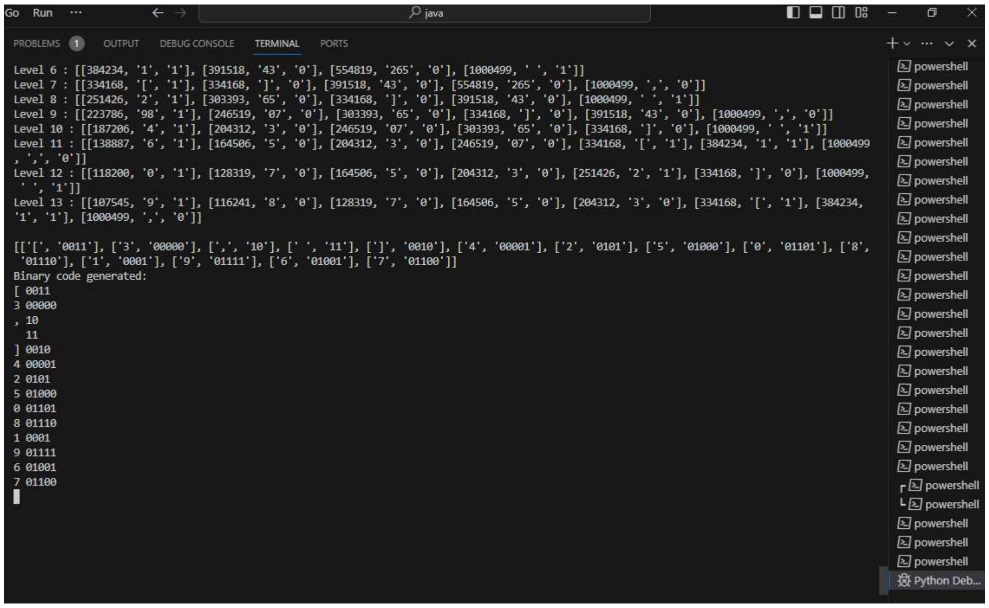
```
  File "C:\Users\WIN10\Documents\3semmsrit\java\image\image.py", line 14, in <module>
    nx.draw_networkx_nodes(web_graph, pos, node_size=700, node_color="skyblue", font_size=10,
TypeError: draw_networkx_nodes() got an unexpected keyword argument 'font_size'
PS C:\Users\WIN10\Documents\3semmsrit\java\image> python image.py
Huffman Compression Program
=================================================================
Enter 1 if you want to input an colour image file, 2 for default gray scale case:1
Enter the filename:image.bmp
Enetered string is: [[[ 3  3  3]
  [ 3  3  3]
  [ 3  3  3]
  ...
  [ 2  7 26]
  [ 2  7 26]
  [ 2  7 26]]

 [[ 3  3  3]
  [ 3  3  3]
  [ 3  3  3]
  ...
  [ 2  7 26]
  [ 2  7 26]
  [ 2  7 26]]

 [[ 3  3  3]
  [ 3  3  3]
  [ 3  3  3]
  ...
  [ 2  7 26]
  [ 2  7 26]
  [ 2  7 26]]

 ...

 [[ 3  3  3]
  [ 3  3  3]
  [ 3  3  1]
```

Fig 4.1 output



```
 [[ 3  3  3]
  [ 3  3  3]
  [ 3  3  1]
  ...
  [ 3  3  3]
  [ 3  3  3]
  [ 3  3  3]]

 [[ 3  3  3]
  [ 3  3  3]
  [ 3  3  1]
  ...
  [ 3  3  3]
  [ 3  3  3]
  [ 3  3  3]]]
Huffman tree with merged pathways:
Level 0 : [[4470210, ' ,9807265[]143']]
Level 1 : [[2000998, ' ,', '1'], [2469212, '9807265[]143', '0']]
Level 2 : [[1025124, '9807265', '1'], [1444088, '[]143', '0']]
Level 3 : [[1000499, ' ', '1'], [1000499, ',', '0'], [1444088, '[]143', '0']]
Level 4 : [[668336, '[]', '1'], [775752, '143', '0'], [1000499, ',', '0']]
Level 5 : [[470305, '9807', '1'], [554819, '265', '0'], [775752, '143', '0'], [1000499, ',', '0']]
Level 6 : [[384234, '1', '1'], [391518, '43', '0'], [554819, '265', '0'], [1000499, ' ', '1']]
Level 7 : [[334168, '[', '1'], [334168, ']', '0'], [391518, '43', '0'], [554819, '265', '0'], [1000499, ',', '0']]
Level 8 : [[251426, '2', '1'], [303393, '65', '0'], [334168, ']', '0'], [391518, '43', '0'], [1000499, ' ', '1']]
Level 9 : [[223786, '98', '1'], [246519, '07', '0'], [303393, '65', '0'], [334168, ']', '0'], [391518, '43', '0'], [1000499, ',', '0']]
Level 10 : [[187206, '4', '1'], [204312, '3', '0'], [246519, '07', '0'], [303393, '65', '0'], [334168, ']', '0'], [1000499, ' ', '1']]
Level 11 : [[138887, '6', '1'], [164506, '5', '0'], [204312, '3', '0'], [246519, '07', '0'], [334168, '[', '1'], [384234, '1', '1'], [1000499
, ',', '0']]
Level 12 : [[118200, '0', '1'], [128319, '7', '0'], [164506, '5', '0'], [204312, '3', '0'], [251426, '2', '1'], [334168, ']', '0'], [1000499,
 ' ', '1']]
Level 13 : [[107545, '9', '1'], [116241, '8', '0'], [128319, '7', '0'], [164506, '5', '0'], [204312, '3', '0'], [334168, '[', '1'], [384234,
'1', '1'], [1000499, ',', '0']]

[['[', '0011'], ['3', '00000'], [',', '10'], [' ', '11'], [']', '0010'], ['4', '00001'], ['2', '0101'], ['5', '01000'], ['0', '01101'], ['8',
 '01110'], ['1', '0001'], ['9', '01111'], ['6', '01001'], ['7', '01100']]
```

Fig 4.2 output



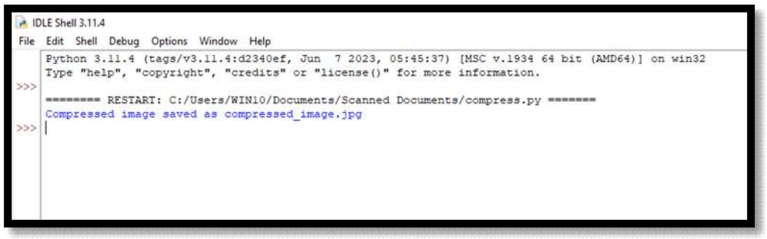
```
Level 6 : [[384234, '1', '1'], [391518, '43', '0'], [554819, '265', '0'], [1000499, ' ', '1']]
Level 7 : [[334168, '[', '1'], [334168, ']', '0'], [391518, '43', '0'], [554819, '265', '0'], [1000499, ',', '0']]
Level 8 : [[251426, '2', '1'], [303393, '65', '0'], [334168, ']', '0'], [391518, '43', '0'], [1000499, ' ', '1']]
Level 9 : [[223786, '98', '1'], [246519, '07', '0'], [303393, '65', '0'], [334168, ']', '0'], [391518, '43', '0'], [1000499, ',', '0']]
Level 10 : [[187206, '4', '1'], [204312, '3', '0'], [246519, '07', '0'], [303393, '65', '0'], [334168, ']', '0'], [1000499, ' ', '1']]
Level 11 : [[138887, '6', '1'], [164506, '5', '0'], [204312, '3', '0'], [246519, '07', '0'], [334168, '[', '1'], [384234, '1', '1'], [1000499
, ',', '0']]
Level 12 : [[118200, '0', '1'], [128319, '7', '0'], [164506, '5', '0'], [204312, '3', '0'], [251426, '2', '1'], [334168, ']', '0'], [1000499,
 ' ', '1']]
Level 13 : [[107545, '9', '1'], [116241, '8', '0'], [128319, '7', '0'], [164506, '5', '0'], [204312, '3', '0'], [334168, '[', '1'], [384234,
'1', '1'], [1000499, ',', '0']]

[['[', '0011'], ['3', '00000'], [',', '10'], [' ', '11'], [']', '0010'], ['4', '00001'], ['2', '0101'], ['5', '01000'], ['0', '01101'], ['8',
 '01110'], ['1', '0001'], ['9', '01111'], ['6', '01001'], ['7', '01100']]
Binary code generated:
[ 0011
3 00000
, 10
  11
] 0010
4 00001
2 0101
5 01000
0 01101
8 01110
1 0001
9 01111
6 01001
7 01100
```

Fig 4.3 output

Code for Image Compression using python

```
from PIL import Image
# Open the image
input_image_path = "input_image.jpg"
image = Image.open('Original_image.jpg') #image name 1.jpg
# Compress and save the image in JPEG format
output_image_path = "Compressed_image.jpg"
image.save(output_image_path, "JPEG", quality=70)
print(f"Compressed image saved as {output_image_path}")
```

Result



```
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun  7 2023, 05:45:37) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: C:/Users/WIN10/Documents/Scanned Documents/compress.py ========
Compressed image saved as compressed_image.jpg
>>>
```

Fig 4.4 output

CHAPTER 5

CONCLUSION

The goal of image compression is to minimize the storage needs while keeping a good visual quality of the image. Two kinds of compression exist: lossless (BMP, GIF, PNG…) and lossy (JPEG). While lossy compression is attractive, it should be used carefully because some images, like cartoons or logos, may be visually impacted by the compression.