Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Information Technology

Department of Artificial Intelligence and Data Science

Name: Siddhesh Dilip Khairnar		
Class: TY	Division: B	Roll No: 372028
Semester: V	Academic Year: 2023-2024	
Subject Name & Code: Image Processing: ADUA31205(B)		
Title of Assignment: Perform image compression using any basic algorithm (e.g., Huffman coding, run length coding, symbol-based encoding).		
Date of Performance: 04-10-2023	Date of Submission: 01-11-2023	

ASSIGNMENT NO. 7

Name : Siddhesh Dilip Khairnar
PRN no : 22110398
Roll no : 372028

Page No.
Date

IP Assignment no 7

Aim: perform image compression using any basic algorithm (e.g. Huffman coding, run length coding, symbol based encoding)

Learning objectives :→

1. Understand the basic principles of Huffman coding.
2. Calculate symbol frequencies in an image to prepare for compression
3. Create a Huffman tree based on the symbol frequencies.
4. Generate Huffman codes for symbols within the tree.
5. Compress an image by replacing symbol with their Huffman codes.

Theory :→

Huffman coding is a variable length encoding algorithm used for data compression. It's often used to compress image. Here's a basic overview of how it work: →

1) **Calculate the frequency of each symbol:** In the case of image compression, symbol could be pixel or groups of pixels. Count how often each symbol appear in the image.

2) **Build a Huffman codes:** Create a Binary tree where each leaf node represents a symbol & the path from the root to each leaf node represent its binary code. Symbols with higher frequencies should be closer to the root of the tree.

3) **Generates Huffman codes:** Traverse the tree to create unique binary codes for each symbol. Codes for symbol closer to the root will be shorter and codes for symbol farther from the root will be longer.

4. Compress the images :— Replace each symbol in the image with its corresponding Huffman code. This will reduce the overall size of the image as frequently occuring symbols will have shorter codes.
5. Decompression :→ To decompress the image, use the Huffman tree to reverse the process by converting the Huffman codes back into symbols.

Huffman coding is efficient for data with non-uniform symbol frequencies, making it suitable for image compression where some pixel values occur more frequently than other.

Conclusion :

- Huffman coding is an effective variable-length encoding algorithm used for image compression. It work by assigning shorter binary codes to more frequently occurring symbols in the image, reducing the overall data size.

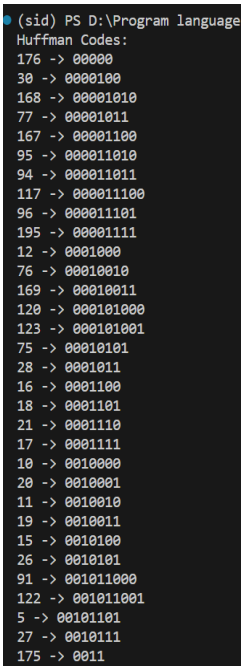
31/10/22

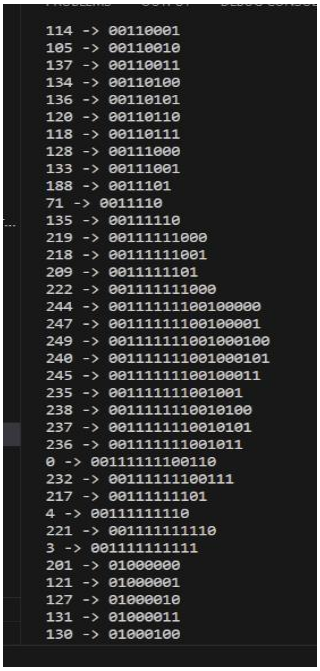
Program Code:

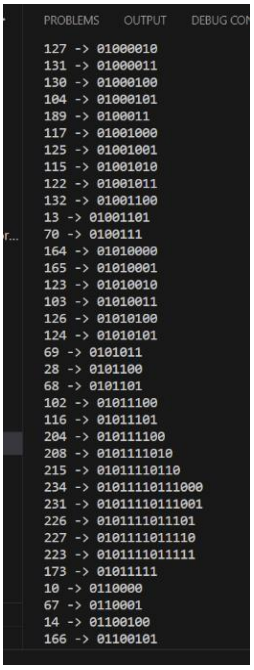
```
import cv2
import numpy as np
from collections import Counter
from heapq import heappush, heappop, heapify


class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

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


def calc_freq(image):
    # Calculate frequency of each pixel value
    freq_dict = Counter(image.flatten())
    return freq_dict


def huffman_tree(freq_dict):
    # Create priority queue from frequency dict
    heap = [[weight, Node(char, weight)] for char, weight in
freq_dict.items()]
    heapify(heap)

    # Build Huffman Tree
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        node = Node(None, lo[0] + hi[0])
        node.left = lo[1]
        node.right = hi[1]
        heappush(heap, [node.freq, node])

    return heap[0][1]


def huffman_encoding(node, binary_string='', code={}):
    # Generate Huffman encoding for each pixel value
    if node is not None:
        if node.char is not None:
            code[node.char] = binary_string
        huffman_encoding(node.left, binary_string + '0', code)
```

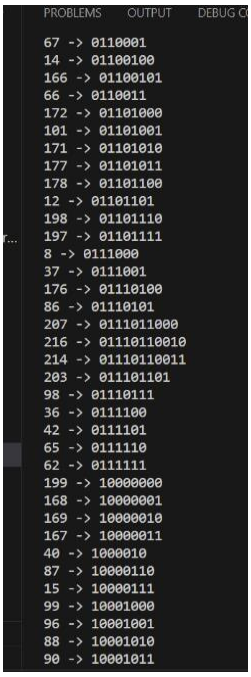
```
        huffman_encoding(node.right, binary_string + '1', code)
    return code


def compress_image(image, code):
    # Compress image using Huffman encoding
    rows, cols = image.shape
    compressed_image = ''
    for i in range(rows):
        for j in range(cols):
            compressed_image += code[image[i, j]]
    return compressed_image


# Load grayscale image
image = cv2.imread(
    "C:/Users/asus/Downloads/EvXjoAkUYAE5K7O.jpg", cv2.IMREAD_GRAYSCALE)

# Calculate frequency of each pixel value
freq_dict = calc_freq(image)

# Build Huffman Tree
tree = huffman_tree(freq_dict)

# Generate Huffman encoding for each pixel value
code = huffman_encoding(tree)

# Compress image using Huffman encoding
compressed_image = compress_image(image, code)

print('Huffman Codes:')
for pixel_value, huffman_code in code.items():
    print(f'{pixel_value} -> {huffman_code}')

# Calculate average number of bits
average_bits = sum(len(code[pixel_value]) * freq for pixel_value,
                   freq in freq_dict.items()) / sum(freq_dict.values())
print('Average number of bits:', average_bits)
```

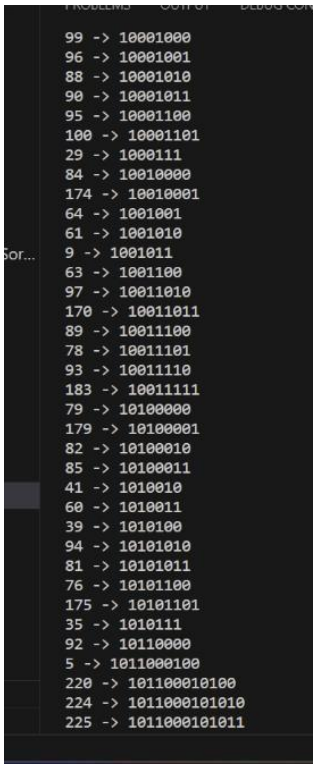
Output:

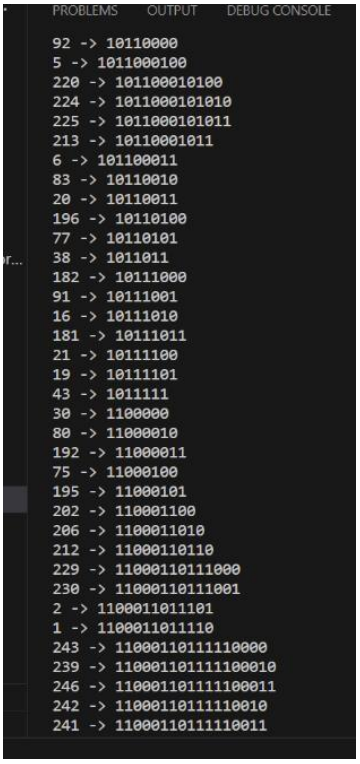

(sid) PS D:\Program language
Huffman Codes:
176 -> 00000
30 -> 0000100
168 -> 00001010
77 -> 00001011
167 -> 00001100
95 -> 000011010
94 -> 000011011
117 -> 000011100
96 -> 000011101
195 -> 00001111
12 -> 0001000
76 -> 00010010
169 -> 00010011
120 -> 000101000
123 -> 000101001
75 -> 00010101
28 -> 0001011
16 -> 0001100
18 -> 0001101
21 -> 0001110
17 -> 0001111
10 -> 0010000
20 -> 0010001
11 -> 0010010
19 -> 0010011
15 -> 0010100
26 -> 0010101
91 -> 001011000
122 -> 001011001
5 -> 00101101
27 -> 0010111
175 -> 0011

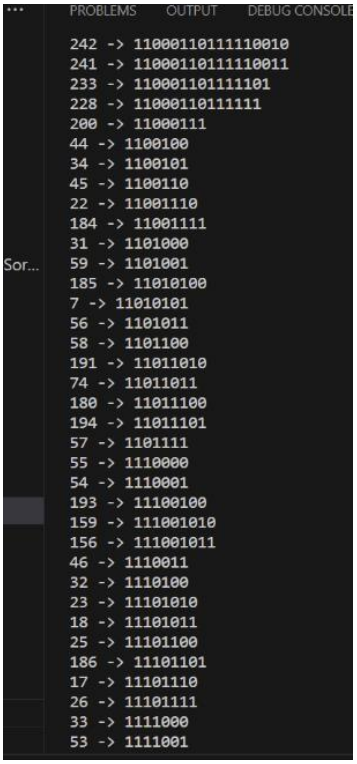

114 -> 00110001
105 -> 00110010
137 -> 00110011
134 -> 00110100
136 -> 00110101
120 -> 00110110
118 -> 00110111
128 -> 00111000
133 -> 00111001
188 -> 0011101
71 -> 0011110
135 -> 00111110
219 -> 001111111000
218 -> 001111111001
209 -> 0011111101
222 -> 001111111000
244 -> 0011111110010000
247 -> 0011111110010001
249 -> 00111111100100100
240 -> 00111111100100101
245 -> 0011111110010011
235 -> 001111111001001
238 -> 001111111010100
237 -> 001111111010101
236 -> 001111111001011
0 -> 0011111110011O
232 -> 0011111110011l
217 -> 001111111101
4 -> 00111111110
221 -> 001111111110
3 -> 001111111111
201 -> 01000000
121 -> 01000001
127 -> 01000010
131 -> 01000011
130 -> 01000100


PROBLEMS OUTPUT DEBUG CON
127 -> 01000010
131 -> 01000011
130 -> 01000100
104 -> 01000101
189 -> 0100011
117 -> 01001000
125 -> 01001001
115 -> 01001010
122 -> 01001011
132 -> 01001100
13 -> 01001101
70 -> 0100111
164 -> 01010000
165 -> 01010001
123 -> 01010010
103 -> 01010011
126 -> 01010100
124 -> 01010101
69 -> 0101011
28 -> 0101100
68 -> 0101101
102 -> 01011100
116 -> 01011101
204 -> 010111100
208 -> 0101111010
215 -> 01011110110
234 -> 010111101111000
231 -> 010111101111001
226 -> 01011110111O1
227 -> 01011110111l0
223 -> 010111101111l1
173 -> 01011111
10 -> 0110000
67 -> 0110001
14 -> 01100100
166 -> 01100101


PROBLEMS OUTPUT DEBUG C
67 -> 0110001
14 -> 01100100
166 -> 01100101
66 -> 0110011
172 -> 01101000
101 -> 01101001
171 -> 01101010
177 -> 01101011
178 -> 01101100
12 -> 01101101
198 -> 01101110
197 -> 01101111
8 -> 0111000
37 -> 0111001
176 -> 01110100
86 -> 01110101
207 -> 0111011000
216 -> 01110110010
214 -> 01110110011
203 -> 011101101
98 -> 01110111
36 -> 0111100
42 -> 0111101
65 -> 0111110
62 -> 0111111
199 -> 10000000
168 -> 10000001
169 -> 10000010
167 -> 10000011
40 -> 1000010
87 -> 10000110
15 -> 10000111
99 -> 10001000
96 -> 10001001
88 -> 10001010


99 -> 10001000
96 -> 10001001
88 -> 10001010
90 -> 10001011
95 -> 10001100
100 -> 10001101
29 -> 1000111
84 -> 10010000
174 -> 10010001
64 -> 1001001
61 -> 1001010
9 -> 1001011
63 -> 1001100
97 -> 10011010
170 -> 10011011
89 -> 10011100
78 -> 10011101
93 -> 10011110
183 -> 10011111
79 -> 10100000
179 -> 10100001
82 -> 10100010
85 -> 10100011
41 -> 1010010
60 -> 1010011
39 -> 1010100
94 -> 10101010
81 -> 10101011
76 -> 10101100
175 -> 10101101
35 -> 1010111
92 -> 10110000
5 -> 1011000100
220 -> 101100010100
224 -> 1011000101010
225 -> 1011000101011


PROBLEMS OUTPUT DEBUG CONSOLE
92 -> 10110000
5 -> 1011000100
220 -> 101100010100
224 -> 1011000101010
225 -> 1011000101011
213 -> 10110001011
6 -> 101100011
83 -> 10110010
20 -> 10110011
196 -> 10110100
77 -> 10110101
38 -> 1011011
182 -> 10111000
91 -> 10111001
16 -> 10111010
181 -> 10111011
21 -> 10111100
19 -> 10111101
43 -> 1011111
30 -> 1100000
80 -> 11000010
192 -> 11000011
75 -> 11000100
195 -> 11000101
202 -> 110001100
206 -> 1100011010
212 -> 11000110110
229 -> 110001101111000
230 -> 110001101111001
2 -> 1100011011101
1 -> 1100011011110
243 -> 110001101111110000
239 -> 110001101111110010
246 -> 110001101111110011
242 -> 110001101111110010
241 -> 110001101111110011


PROBLEMS OUTPUT DEBUG CONSOLE
242 -> 110001101111110010
241 -> 110001101111110011
233 -> 110001101111101
228 -> 11000110111111
200 -> 11000111
44 -> 1100100
34 -> 1100101
45 -> 1100110
22 -> 11001110
184 -> 11001111
31 -> 1101000
59 -> 1101001
185 -> 11010100
7 -> 11010101
56 -> 1101011
58 -> 1101100
191 -> 11011010
74 -> 11011011
180 -> 11011100
194 -> 11011101
57 -> 1101111
55 -> 1110000
54 -> 1110001
193 -> 11100100
159 -> 111001010
156 -> 111001011
46 -> 1110011
32 -> 1110100
23 -> 11101010
18 -> 11101011
25 -> 11101100
186 -> 11101101
17 -> 11101110
26 -> 11101111
33 -> 1111000
53 -> 1111001

```
97 -> 1111100000
232 -> 1111100001000
238 -> 111110000100100
242 -> 111110000100101
234 -> 11111000010011
241 -> 111110000101000
240 -> 111110000101001
253 -> 1111100001010100
254 -> 1111100001010101
246 -> 111110000101011
231 -> 1111100001011
227 -> 11111000011
196 -> 111110001
212 -> 111110010
224 -> 1111100110
118 -> 1111100111
9 -> 11111010
14 -> 11111011
99 -> 1111110000
98 -> 1111110001
115 -> 1111110010
119 -> 1111110011
29 -> 11111101
206 -> 111111100
78 -> 111111101
13 -> 11111111
Average number of bits: 7.228582148295827
```