**Practical-3**

**Aim: Implement Huffman Code(HC) to generate binary code when symbol and probabilities are given**

**Code:**

import heapq

class node:

def \_\_init\_\_(self, freq, symbol, left=None, right=None):

self.freq = freq

self.symbol = symbol

self.left = left

self.right = right

self.huff = ''

def \_\_lt\_\_(self, nxt):

return self.freq < nxt.freq

def printNodes(node, val=''):

newVal = val + str(node.huff)

if(node.left):

printNodes(node.left, newVal)

if(node.right):

printNodes(node.right, newVal)

if(not node.left and not node.right):

print(f"{node.symbol} -> {newVal}")

chars = ['y', 'a', 's', 'h', 'p', 't']

freq = [ 25, 6, 50, 20, 10, 95]

nodes = []

for x in range(len(chars)):

heapq.heappush(nodes, node(freq[x], chars[x]))

while len(nodes) > 1:

left = heapq.heappop(nodes)

right = heapq.heappop(nodes)

left.huff = 0

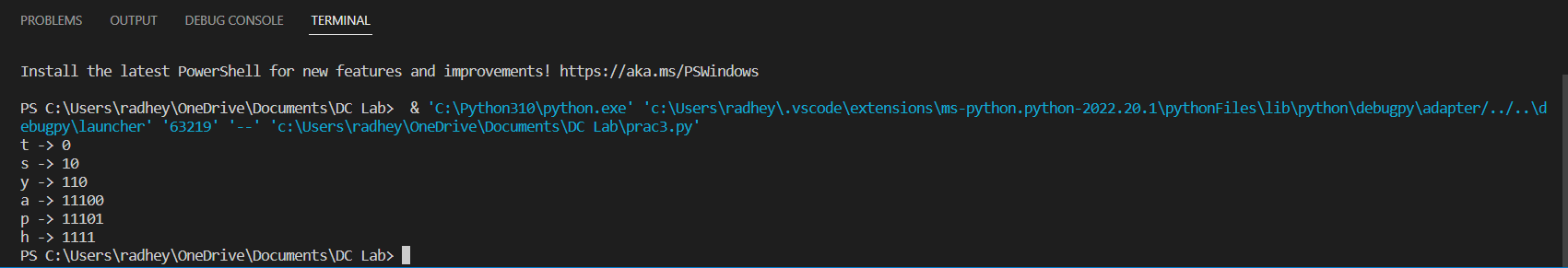
right.huff = 1

newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

heapq.heappush(nodes, newNode)

printNodes(nodes[0])

**Output:**

****

**Practical-4**

**Aim: Implement Huffman code which can compress given file and decompress compressed file**

**Code:**

class Nodes:

def \_\_init\_\_(self, probability, symbol, left = None, right = None):

self.probability = probability

self.symbol = symbol

self.left = left

self.right = right

self.code = ''

def CalculateProbability(the\_data):

the\_symbols = dict()

for item in the\_data:

if the\_symbols.get(item) == None:

the\_symbols[item] = 1

else:

the\_symbols[item] += 1

return the\_symbols

the\_codes = dict()

def CalculateCodes(node, value = ''):

newValue = value + str(node.code)

if(node.left):

CalculateCodes(node.left, newValue)

if(node.right):

CalculateCodes(node.right, newValue)

if(not node.left and not node.right):

the\_codes[node.symbol] = newValue

return the\_codes

def OutputEncoded(the\_data, coding):

encodingOutput = []

for element in the\_data:

encodingOutput.append(coding[element])

the\_string = ''.join([str(item) for item in encodingOutput])

return the\_string

def TotalGain(the\_data, coding):

beforeCompression = len(the\_data) \* 8

afterCompression = 0

the\_symbols = coding.keys()

for symbol in the\_symbols:

the\_count = the\_data.count(symbol)

afterCompression += the\_count \* len(coding[symbol])

print("Space usage before compression (in bits):", beforeCompression)

print("Space usage after compression (in bits):", afterCompression)

def HuffmanEncoding(the\_data):

symbolWithProbs = CalculateProbability(the\_data)

the\_symbols = symbolWithProbs.keys()

the\_probabilities = symbolWithProbs.values()

print("symbols: ", the\_symbols)

print("probabilities: ", the\_probabilities)

the\_nodes = []

for symbol in the\_symbols:

the\_nodes.append(Nodes(symbolWithProbs.get(symbol), symbol))

while len(the\_nodes) > 1:

the\_nodes = sorted(the\_nodes, key = lambda x: x.probability)

right = the\_nodes[0]

left = the\_nodes[1]

left.code = 0

right.code = 1

newNode = Nodes(left.probability + right.probability, left.symbol + right.symbol, left, right)

the\_nodes.remove(left)

the\_nodes.remove(right)

the\_nodes.append(newNode)

huffmanEncoding = CalculateCodes(the\_nodes[0])

print("symbols with codes", huffmanEncoding)

TotalGain(the\_data, huffmanEncoding)

encodedOutput = OutputEncoded(the\_data,huffmanEncoding)

return encodedOutput, the\_nodes[0]

def HuffmanDecoding(encodedData, huffmanTree):

treeHead = huffmanTree

decodedOutput = []

for x in encodedData:

if x == '1':

huffmanTree = huffmanTree.right

elif x == '0':

huffmanTree = huffmanTree.left

try:

if huffmanTree.left.symbol == None and huffmanTree.right.symbol == None:

pass

except AttributeError:

decodedOutput.append(huffmanTree.symbol)

huffmanTree = treeHead

string = ''.join([str(item) for item in decodedOutput])

return string

the\_data = input("enter the string")

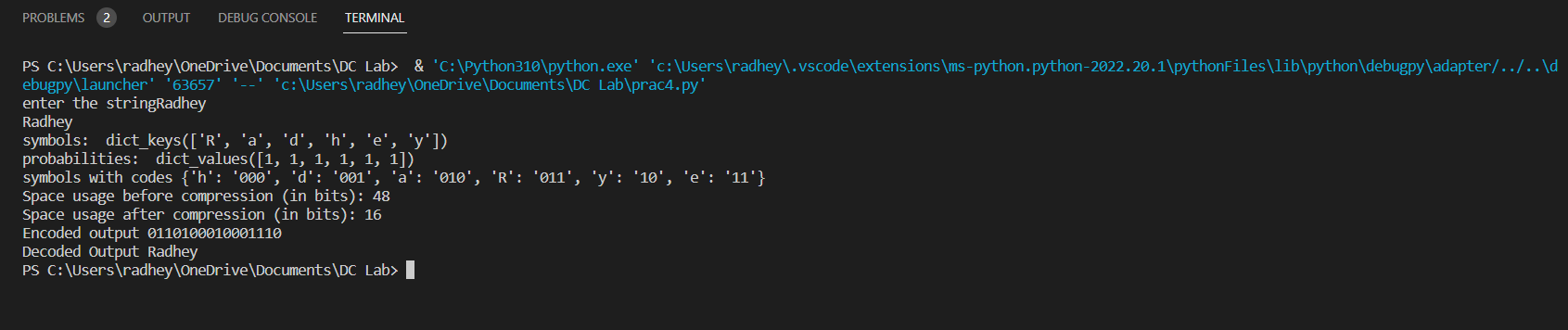
print(the\_data)

encoding, the\_tree = HuffmanEncoding(the\_data)

print("Encoded output", encoding)

print("Decoded Output", HuffmanDecoding(encoding, the\_tree))

**Output:**

****

**Practical-5**

**Aim: Implement adaptive Huffman program to compress decompressed file**

**Code:**

import heapq

class Node:

def \_\_init\_\_(self, symbol, frequency, left=None, right=None):

self.symbol = symbol

self.frequency = frequency

self.left = left

self.right = right

def \_\_lt\_\_(self, other):

if self.frequency == other.frequency:

return id(self) < id(other)

return self.frequency < other.frequency

def build\_frequency\_table(data):

frequency\_table = {}

for symbol in data:

if symbol not in frequency\_table:

frequency\_table[symbol] = 0

frequency\_table[symbol] += 1

return frequency\_table

def build\_huffman\_tree(frequency\_table):

heap = []

for symbol, frequency in frequency\_table.items():

heap.append(Node(symbol, frequency))

heapq.heapify(heap)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

node = Node(None, left.frequency + right.frequency, left, right)

heapq.heappush(heap, node)

return heap[0]

def build\_codewords(node, codewords, prefix):

if node is None:

return

if node.symbol is not None:

codewords[node.symbol] = prefix

build\_codewords(node.left, codewords, prefix + "0")

build\_codewords(node.right, codewords, prefix + "1")

def compress(data):

frequency\_table = build\_frequency\_table(data)

huffman\_tree = build\_huffman\_tree(frequency\_table)

codewords = {}

build\_codewords(huffman\_tree, codewords, "")

compressed\_data = ""

for symbol in data:

compressed\_data += codewords[symbol]

return compressed\_data, huffman\_tree, frequency\_table

def decompress(compressed\_data, huffman\_tree, frequency\_table):

data = ""

node = huffman\_tree

for bit in compressed\_data:

if bit == "0":

node = node.left

else:

node = node.right

if node.symbol is not None:

data += node.symbol

node = huffman\_tree

return data

def main():

data = "Hello Radhey Mugdal here"

compressed\_data, huffman\_tree, frequency\_table = compress(data)

print("Original data:", data)

print("Compressed data:", compressed\_data)

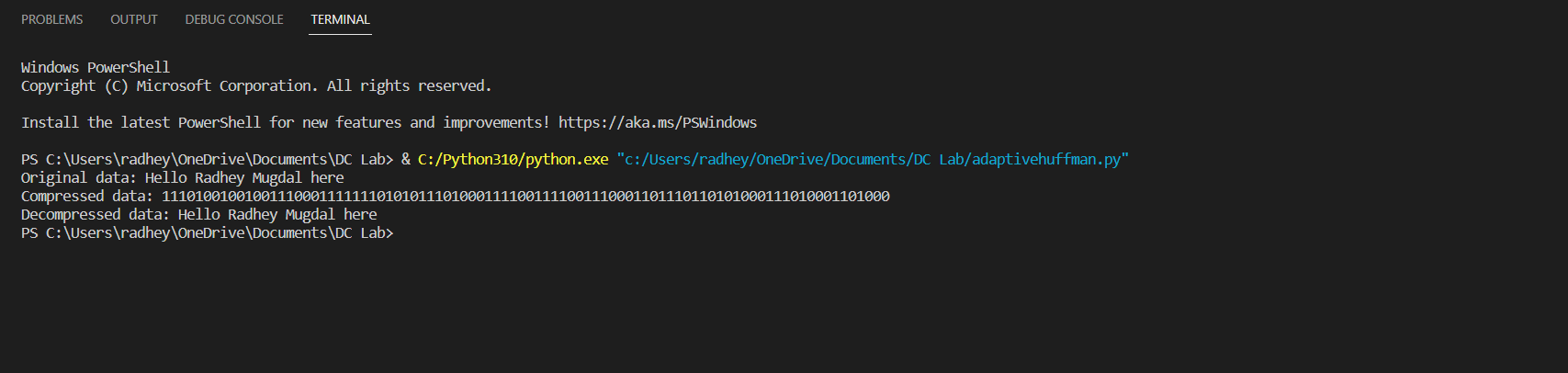
decompressed\_data = decompress(compressed\_data, huffman\_tree, frequency\_table)

print("Decompressed data:", decompressed\_data)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

****

**Practical-2**

**Aim: Write a program to generate binary code in case of arithmetic coding**

**Code:**

def generate\_binary\_code(message, probability\_table):

# Calculate the range for each symbol in the message

ranges = []

lower\_bound = 0

for symbol in message:

upper\_bound = lower\_bound + probability\_table[symbol]

ranges.append((lower\_bound, upper\_bound))

lower\_bound = upper\_bound

# Encode the message by finding the binary representation of the final range

code = ""

for i in range(32):

mid = (ranges[-1][0] + ranges[-1][1]) / 2

if mid < 0.5:

code += "0"

ranges = [(ranges[-1][0] \* 2, ranges[-1][1] \* 2)]

else:

code += "1"

ranges = [(2 \* ranges[-1][0] - 1, 2 \* ranges[-1][1] - 1)]

return code

# Example usage

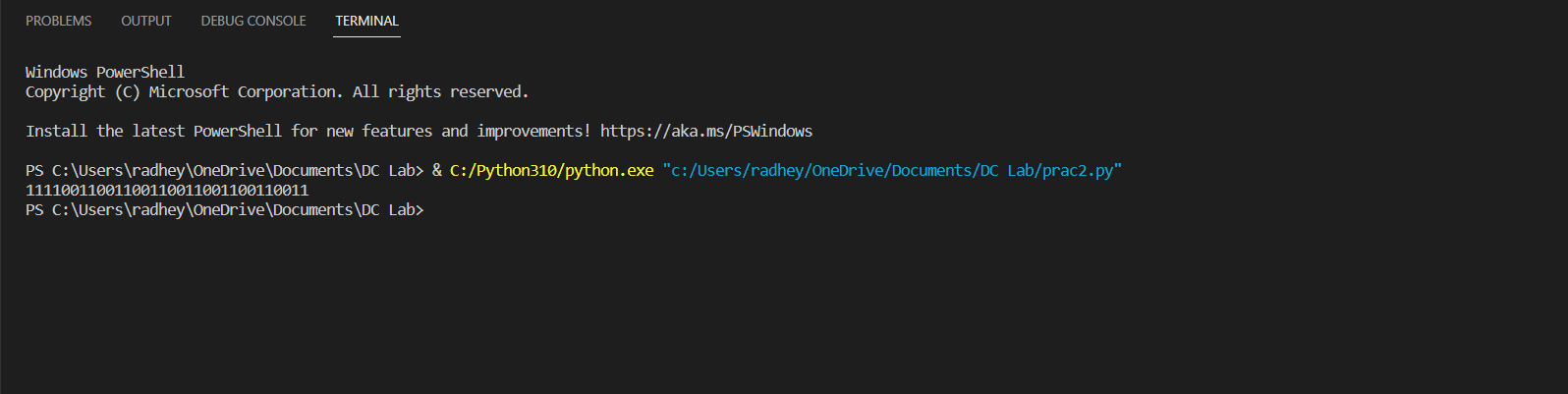
probability\_table = {"a": 0.4, "b": 0.3, "c": 0.2, "d": 0.1}

message = "abcd"

binary\_code = generate\_binary\_code(message, probability\_table)

print(binary\_code)

**Output:**

****

**Practical-6**

**Aim: Write a program to implement LZ77 algorithm**

**Code:**

def compress(data):

# Initialize variables

window\_start = 0

window\_end = 0

lookahead\_start = 0

lookahead\_end = 0

output = []

# Iterate over the data

while lookahead\_end < len(data):

# Find the longest match between the window and the lookahead buffer

match\_length = 0

match\_position = 0

for i in range(window\_start, window\_end):

length = 0

while (

data[i + length] == data[lookahead\_start + length] and

lookahead\_start + length < len(data)

):

length += 1

if length > match\_length:

match\_length = length

match\_position = i

# If there is a match, output the reference and move the window

if match\_length > 0:

output.append((match\_position, match\_length))

window\_start += match\_length

window\_end += match\_length

lookahead\_start += match\_length

lookahead\_end += match\_length

# Otherwise, output the next character and move the window

else:

output.append(data[lookahead\_start])

window\_start += 1

window\_end += 1

lookahead\_start += 1

lookahead\_end += 1

# Return the compressed data

return output

def decompress(data):

# Initialize variables

output = ""

window\_start = 0

for item in data:

# If the item is a reference, retrieve the referenced data

if isinstance(item, tuple):

position, length = item

for i in range(length):

output += data[position + i]

window\_start += 1

else:

output += item

window\_start += 1

# Return the decompressed data

return output

data = "Hello Mann Patel Here!"

compressed = compress(data)

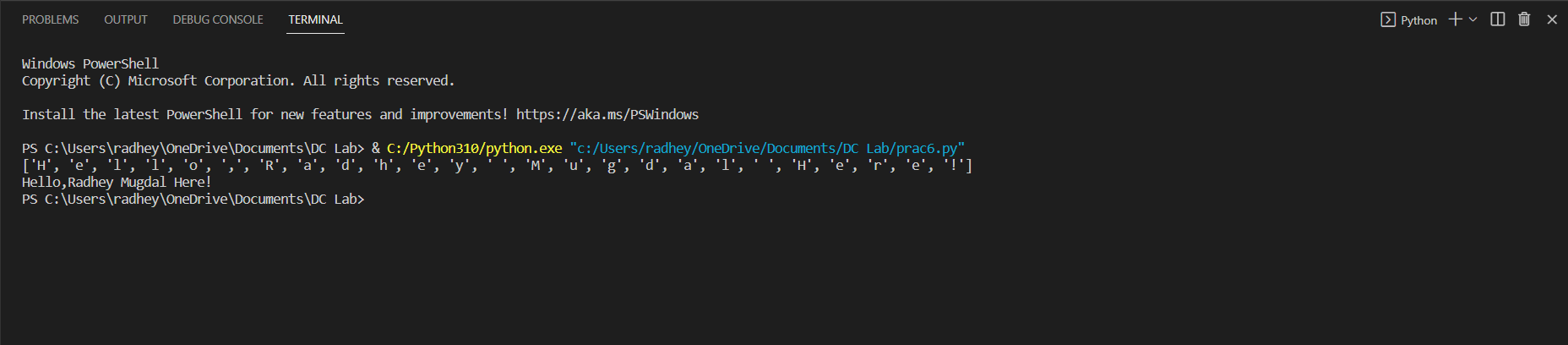
decompressed = decompress(compressed)

assert data == decompressed

print(compressed)

print(decompressed)

**Output:**

****

**Practical-7**

**Aim: Write a program to implement LZW algorithm**

**Code:**

def lzw\_compress(text):

dictionary = {}

code = []

for i in range(256):

dictionary[chr(i)] = i

next\_code = 256

current\_str = ""

for char in text:

if current\_str + char in dictionary:

current\_str += char

else:

code.append(dictionary[current\_str])

dictionary[current\_str + char] = next\_code

next\_code += 1

current\_str = char

code.append(dictionary[current\_str])

return code

def lzw\_decompress(code):

dictionary = {}

for i in range(256):

dictionary[i] = chr(i)

next\_code = 256

current\_str = dictionary[code[0]]

text = current\_str

for index in code[1:]:

if index in dictionary:

entry = dictionary[index]

elif index == next\_code:

entry = current\_str + current\_str[0]

else:

raise ValueError("Bad compressed code")

text += entry

dictionary[next\_code] = current\_str + entry[0]

next\_code += 1

current\_str = entry

return text

text = "abababcabababxyxyxyxyxyxyxy"

compressed\_data = lzw\_compress(text)

print(compressed\_data)

decompressed\_text = lzw\_decompress(compressed\_data)

print(decompressed\_text)

**Output:**

****

**Practical-8**

**Aim: Write a program to Implement LZ78 algorithm.**

**Code:**

def lz78\_compress(text):

dictionary = {}

code = []

next\_code = 0

current\_str = ""

for char in text:

if current\_str + char in dictionary:

current\_str += char

else:

code.append((dictionary.get(current\_str, 0), char))

next\_code += 1

dictionary[current\_str + char] = next\_code

current\_str = ""

if current\_str in dictionary:

code.append((dictionary[current\_str], ''))

return code

def lz78\_decompress(code):

dictionary = {0: ''}

next\_code = 1

text = ""

for index, char in code:

if index == 0:

text += char

dictionary[next\_code] = char

next\_code += 1

else:

entry = dictionary[index] + char

text += entry

dictionary[next\_code] = entry

next\_code += 1

return text

text = "abababcabababxyxyxyxyxyxyxy"

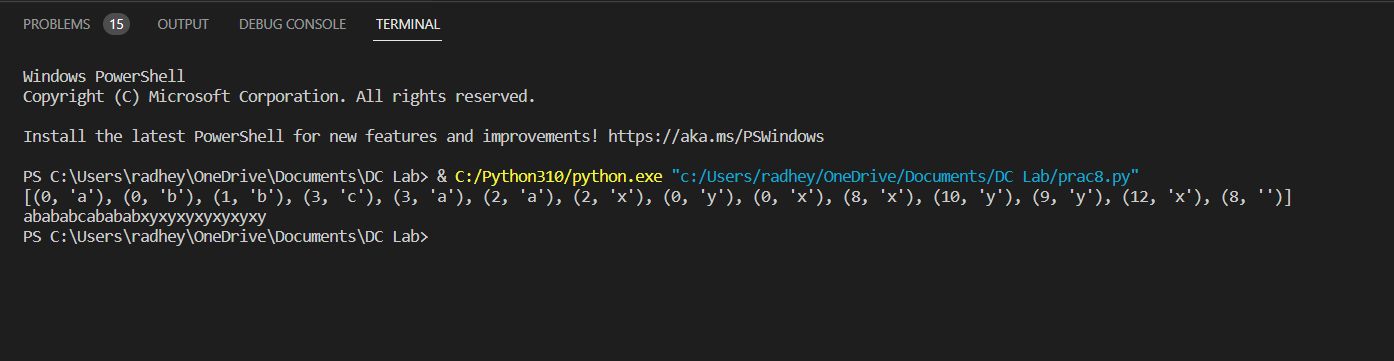
compressed\_data = lz78\_compress(text)

print(compressed\_data)

decompressed\_text = lz78\_decompress(compressed\_data)

print(decompressed\_text)

**Output:**

****

**Practical-9**

**Aim: Write a program to Implement BWT algorithm.**

**Code:**

def search(input\_char, char\_list):

for i in range(len(char\_list)):

if char\_list[i] == input\_char:

return i

def move\_to\_front(curr\_index, char\_list):

char = char\_list.pop(curr\_index)

char\_list.insert(0, char)

def mtf\_encode(input\_text, char\_list):

output\_arr = []

for char in input\_text:

output\_arr.append(search(char, char\_list))

move\_to\_front(output\_arr[-1], char\_list)

return output\_arr

if \_\_name\_\_ == '\_\_main\_\_':

input\_text = "radhey"

char\_list = list("abcdefghijklmnopqrstuvwxyz")

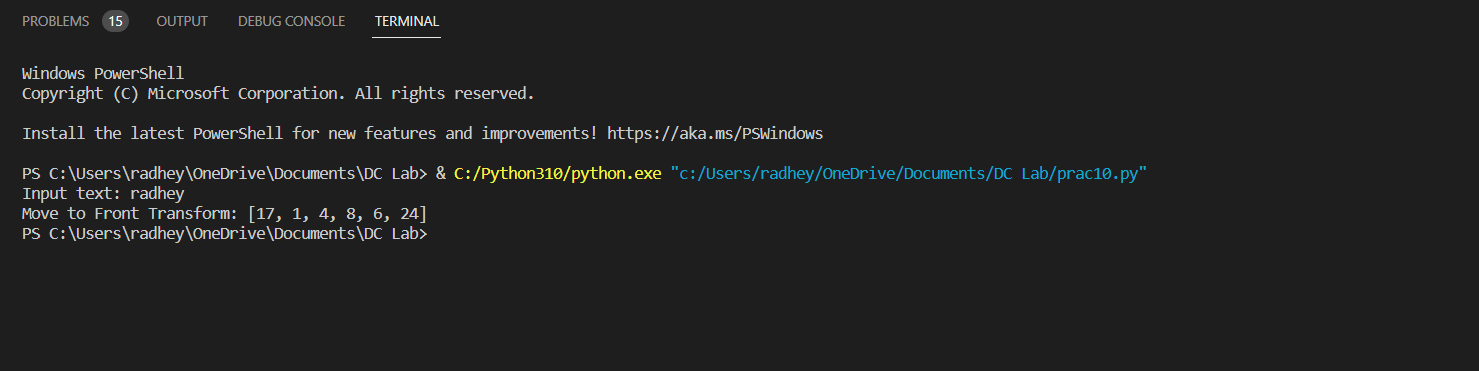
print(f"Input text: {input\_text}")

print("Move to Front Transform:", end=" ")

mtf\_transform = mtf\_encode(input\_text, char\_list)

print(mtf\_transform)

**Output:**

****

**Practical-10**

**Aim: Write a program to Implement BWT algorithm.**

**Code:**

def bwt(text):

text += '$'

rotations = [text[i:] + text[:i] for i in range(len(text))]

rotations.sort()

transformed\_text = ''.join([rotation[-1] for rotation in rotations])

index = rotations.index(text)

return transformed\_text, index

def inverse\_bwt(transformed\_text, index):

sorted\_chars = sorted(transformed\_text)

matrix = [''] \* len(transformed\_text)

for i, char in enumerate(sorted\_chars):

matrix[i] = char

for i in range(1, len(transformed\_text)):

for j in range(len(transformed\_text)):

matrix[j] = transformed\_text[j] + matrix[j]

matrix.sort()

return matrix[index].rstrip('$')

text = "banana"

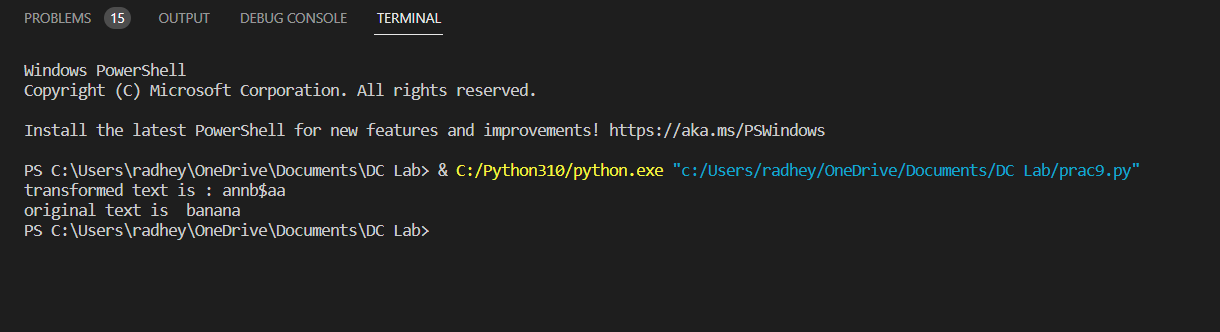
transformed\_text, index = bwt(text)

print("transformed text is :" ,transformed\_text)

original\_text = inverse\_bwt(transformed\_text, index)

print("original text is ",original\_text)

**Output:**

****

**Practical-1**

**Aim: Write a program that compress and displays uncompressed windows BMP image file.**

**Code:**

import zlib

def compress\_bmp(infile, outfile):

with open(infile, 'rb') as f\_in, open(outfile, 'wb') as f\_out:

data = f\_in.read()

compressed\_data = zlib.compress(data)

f\_out.write(compressed\_data)

def decompress\_bmp(infile, outfile):

with open(infile, 'rb') as f\_in, open(outfile, 'wb') as f\_out:

compressed\_data = f\_in.read()

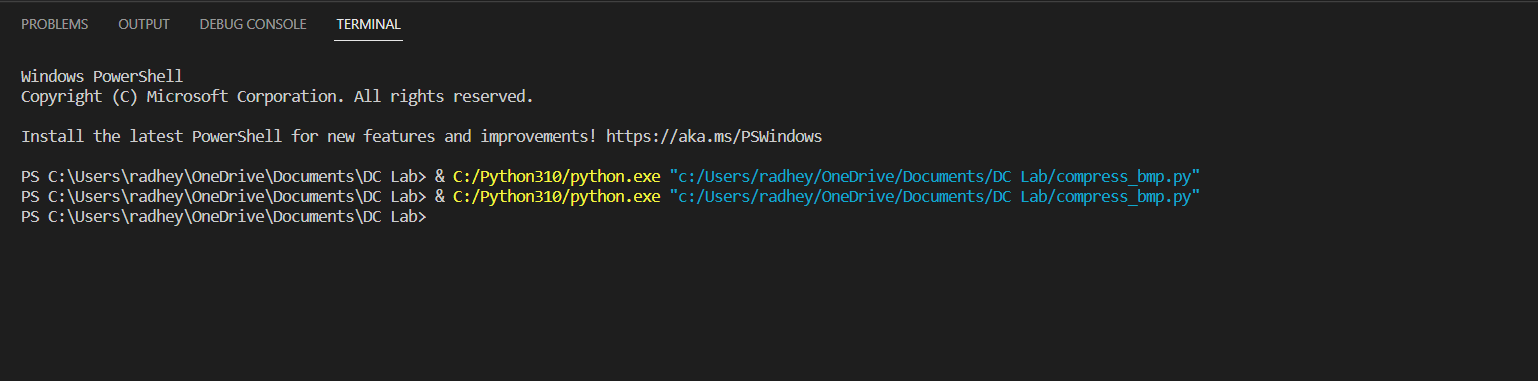
data = zlib.decompress(compressed\_data)

f\_out.write(data)

compress\_bmp('example.bmp', 'compressed.bmp')

decompress\_bmp('compressed.bmp', 'output.bmp')

**Output:**

****

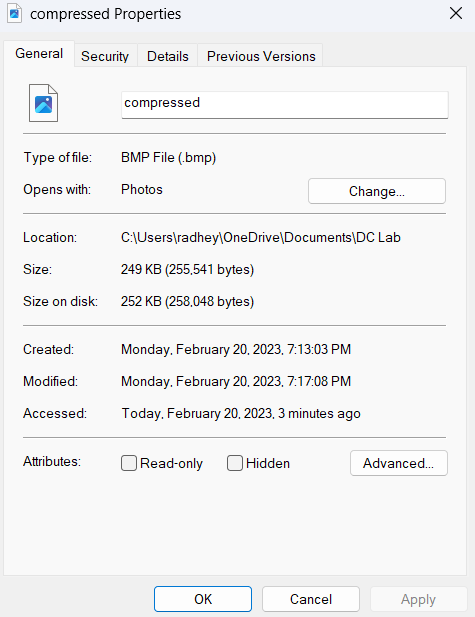
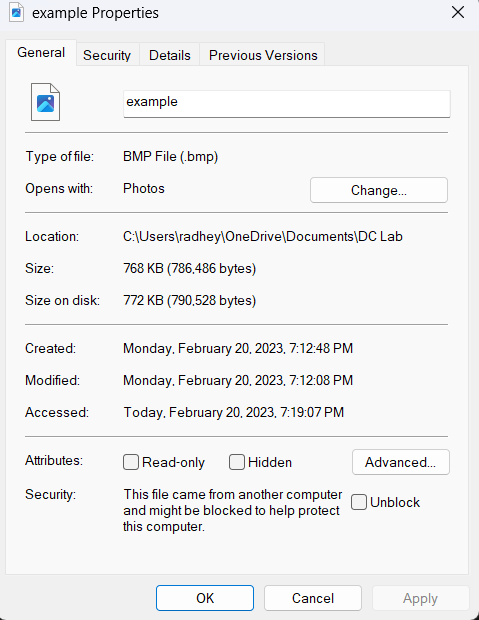


Photo Before compression Photo after compression



Output.bmp

**Practical-11**

**Aim: Write a program which performs JPEG compression, process step**

**by step for given 8x8 block and decompression also.**

**Code:**

import cv2

from scipy.fftpack import dct

import heapq

from PIL import Image

import numpy as np

# Load image as grayscale

img = Image.open("D:\\python.3.9\\python\_collage\\asd\\dc\\test.jpeg").convert("L")

# Convert image to NumPy array

img = np.array(img)

def zigzag(input):

# Returns 1D array of the input array in zig-zag order

return np.concatenate([np.diagonal(input[::-1,:], i)[::(2\*(i % 2)-1)] for i in range(1-

input.shape[0], input.shape[0])])

def rle\_encode(arr):

# RLE (Run-Length Encoding) compression for zero runs

rle = []

count = 0

for i in range(len(arr)):

if arr[i] == 0:

count += 1

else:

rle.append(count)

rle.append(arr[i])

count = 0

rle.append(count)

return rle

def huffman\_encoding(arr):

# Huffman Encoding for RLE (Run-Length Encoding) compressed values

freq\_dict = {}

for i in range(len(arr)):

if arr[i] not in freq\_dict:

freq\_dict[arr[i]] = 1

else:

freq\_dict[arr[i]] += 1

heap = [[freq, [val, ""]] for val, freq in freq\_dict.items()]

heapq.heapify(heap)

while len(heap) > 1:

left = heapq.heappop(heap)

right = heapq.heappop(heap)

for pair in left[1:]:

pair[1] = '0' + pair[1]

for pair in right[1:]:

pair[1] = '1' + pair[1]

heapq.heappush(heap, [left[0] + right[0]] + left[1:] + right[1:])

huff\_dict = dict(heapq.heappop(heap)[1:])

huff\_encoded = ""

for i in range(len(arr)):

huff\_encoded += huff\_dict[arr[i]]

return huff\_encoded, huff\_dict

# Pad the image if necessary to ensure it is divisible by 8 in both dimensions

height, width = img.shape

if height % 8 != 0:

pad\_height = 8 - (height % 8)

else:

pad\_height = 0

if width % 8 != 0:

pad\_width = 8 - (width % 8)

else:

pad\_width = 0

img = np.pad(img, ((0, pad\_height), (0, pad\_width)), mode='constant')

# Divide the image into 8x8 blocks and perform DCT (Discrete Cosine Transform) on each

block

dct\_blocks = np.zeros(img.shape)

for i in range(0, img.shape[0], 8):

for j in range(0, img.shape[1], 8):

dct\_blocks[i:i+8, j:j+8] = dct(dct(img[i:i+8, j:j+8].T).T)

# Quantization matrix

quant = np.array([[16, 11, 10, 16, 24, 40, 51, 61],

[12, 12, 14, 19, 26, 58, 60, 55],

[14, 13, 16, 24, 40, 57, 69, 56],

[14, 17, 22, 29, 51, 87, 80, 62],

[18, 22, 37, 56, 68, 109, 103, 77],

[24, 35, 55, 64, 81, 104, 113, 92],

[49, 64, 78, 87, 103, 121, 120, 101],

[72, 92, 95, 98, 112, 100, 103, 99]])

quant\_blocks = np.zeros(img.shape)

for i in range(0, img.shape[0], 8):

for j in range(0, img.shape[1], 8):

quant\_blocks[i:i+8, j:j+8] = np.round(dct\_blocks[i:i+8, j:j+8] / quant)

rle\_compressed = []

for i in range(0, img.shape[0], 8):

for j in range(0, img.shape[1], 8):

rle\_block = rle\_encode(zigzag(quant\_blocks[i:i+8, j:j+8]))

rle\_compressed.extend(rle\_block)

huff\_encoded, huff\_dict = huffman\_encoding(rle\_compressed)

with open("huff\_dict.txt", "w") as f:

for key, value in huff\_dict.items():

f.write(str(key) + ":" + str(value) + "\n")

binary\_encoded = ""

for char in huff\_encoded:

binary\_encoded += huff\_dict[int(char)]

while len(binary\_encoded) % 8 != 0:

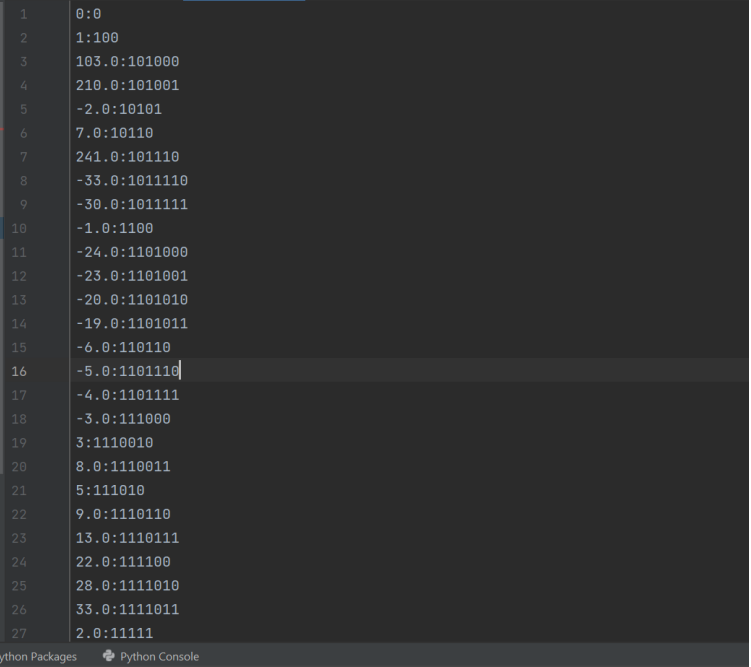
binary\_encoded += "0"

with open("output.bin", "wb") as f:

for i in range(0, len(binary\_encoded), 8):

f.write(bytes([int(binary\_encoded[i:i+8], 2)]))

**Output:**

****