Dayananda Sagar College of Engineering

Department of Electronics and Communication Engineering

Shavige Malleshwara Hills, Kumaraswamy Layout, Bengaluru – 560 078.

(An Autonomous Institute affiliated to VTU, Approved by AICTE & ISO 9001:2008 Certified)

Accredited by National Assessment and Accreditation Council (NAAC) with 'A' grade

Assignment

Program: B.E. Branch: ECE

Course: Programming in Python Semester : 5

Course Code: 18EC5DEBPP Date:

A Report on

LZ77,Huffman,Shannon Encoding and Decoding for text input, LZ77,Huffman,Shannon Encoding,decoding for Image Compression

Submitted by

USN:1DS18EC091 **NAME:Sudhamshu B N**

USN:1DS19EC426 **Name:Rajasudhan Gowda S A**

Faculty In-charge

Prof. Deepa N P

Signature of Faculty In-charge

Introduction

The LZ77 Compression Algorithm is used to analyze input data and determine how to reduce the size of that input data by replacing redundant information with metadata. Sections of the data that are identical to sections of the data that have been encoded are replaced by a small amount of metadata that indicates how to expand those sections again. The encoding algorithm is used to take that combination of data and metadata and serialize it into a stream of bytes that can later be decoded and decompressed.

Huffman coding is a data compression algorithm used to reduce data size. It uses the Greedy algorithm as its technique for implementation.Huffman code assigns codewords depending on the frequency of the characters in some data. It assigns a shorter length codeword for a character with high frequency and a longer length codeword for a character with less frequency. In this way, it reduces the amount of space that the data takes.

Shannon Fano Algorithm is an entropy coding technique used for lossless data compression. It uses the probabilities of occurrence of a character and assigns a unique variable-length code to each of them.

In this mini project we are implementing both algorithms in Python.

Algorithms

LZ77 Encoding-Algorithm

- Find the longest match in the window for the lookahead buffer.
- If a match is found, output the pointer P. Move the coding position (and the window) L bytes forward.
- If a match is not found, output a null pointer and the first byte in the lookahead buffer. Move the coding position (and the window) one byte forward.
- If the lookahead buffer is not empty, return to step 2.

LZ77 Decoding-Algorithm

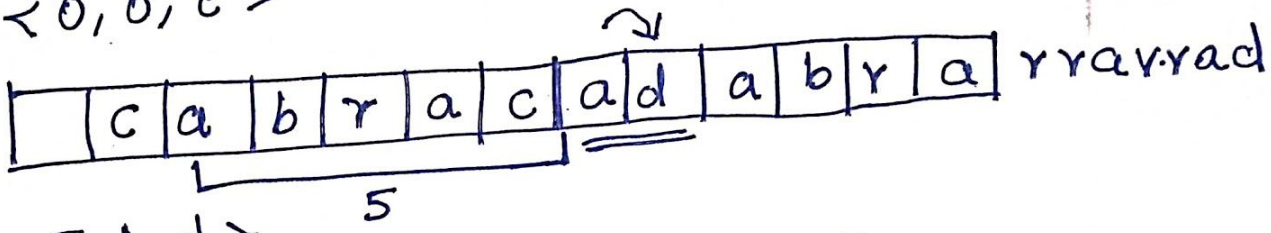
- Reverse the method used in encoding to generate the sequences.
- Do this operation using <encodedNumbers, encodedSizes, encodedLetters.>

LZ77 Encoding Example:

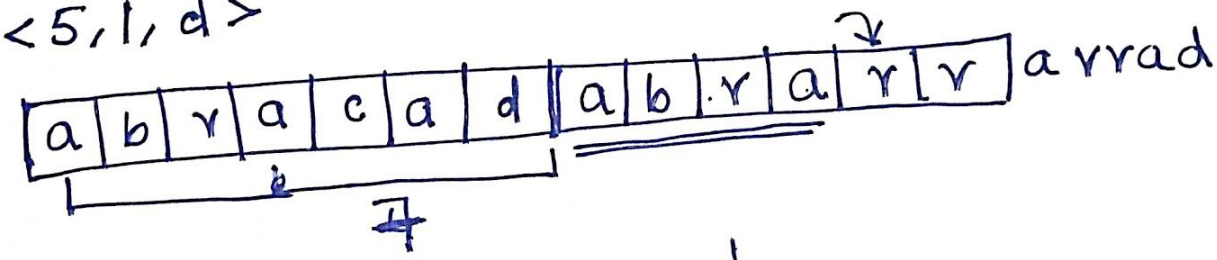
Input: “cabracadabrarrarrad”
Window Size: 13
Search Window Size: 7
Preview/Lookup Window Size: 6

LZ77 Encoding Diagram:

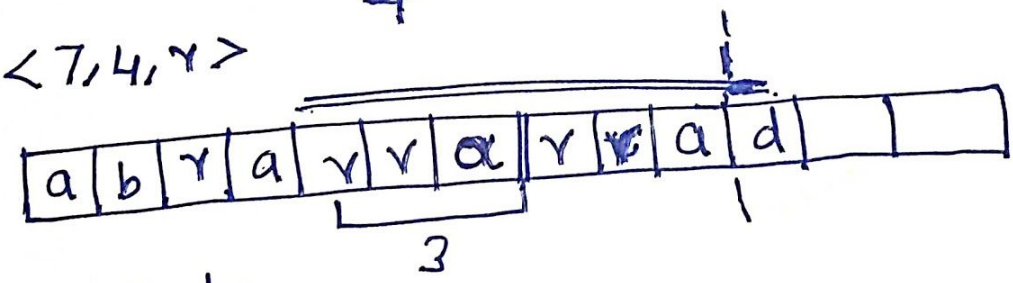
<0,0,c>
<0,0,a>
<0,0,b>
<0,0,r>
<0,0,a>
<0,0,c>



<5,1,d>

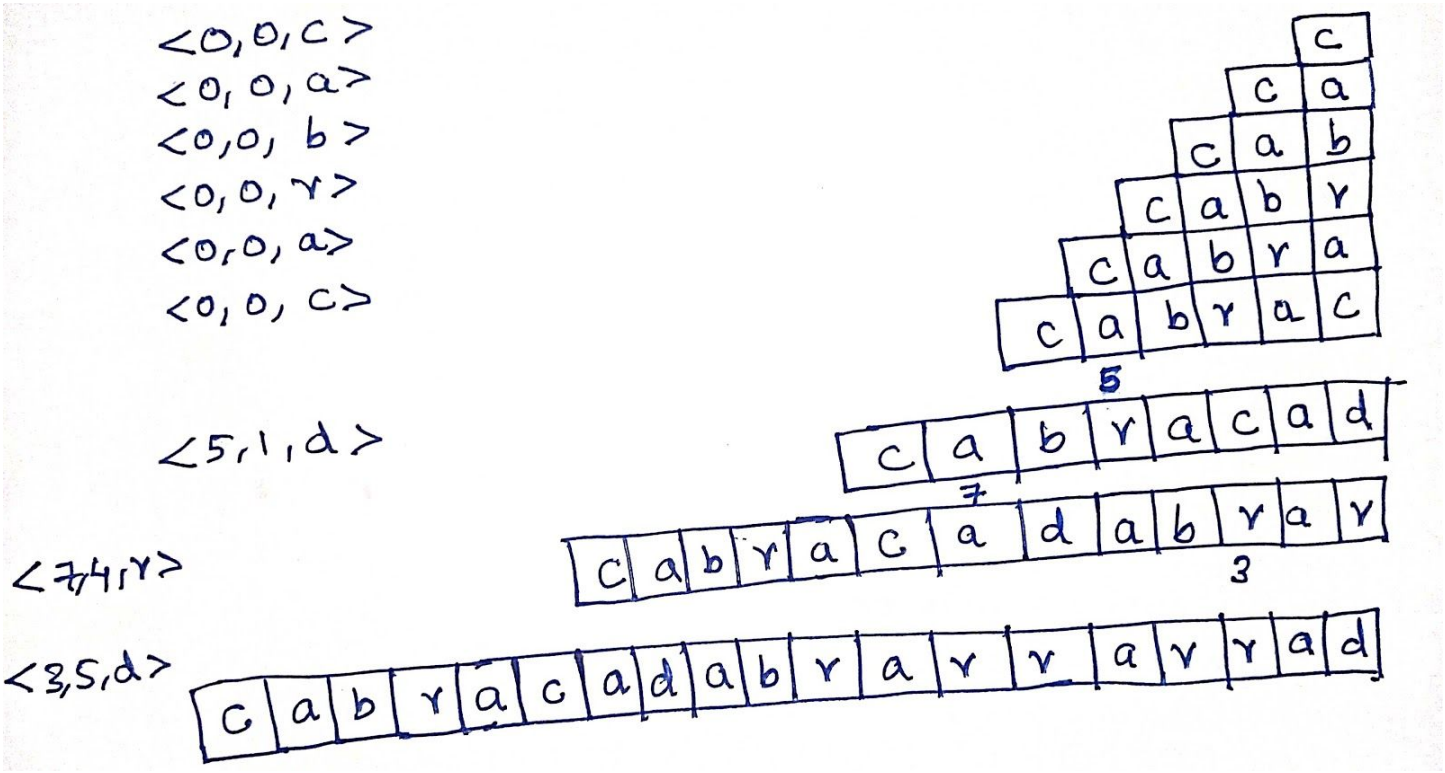


<7,4,r>



<3,5,d>

LZ77 Decoding Diagram:



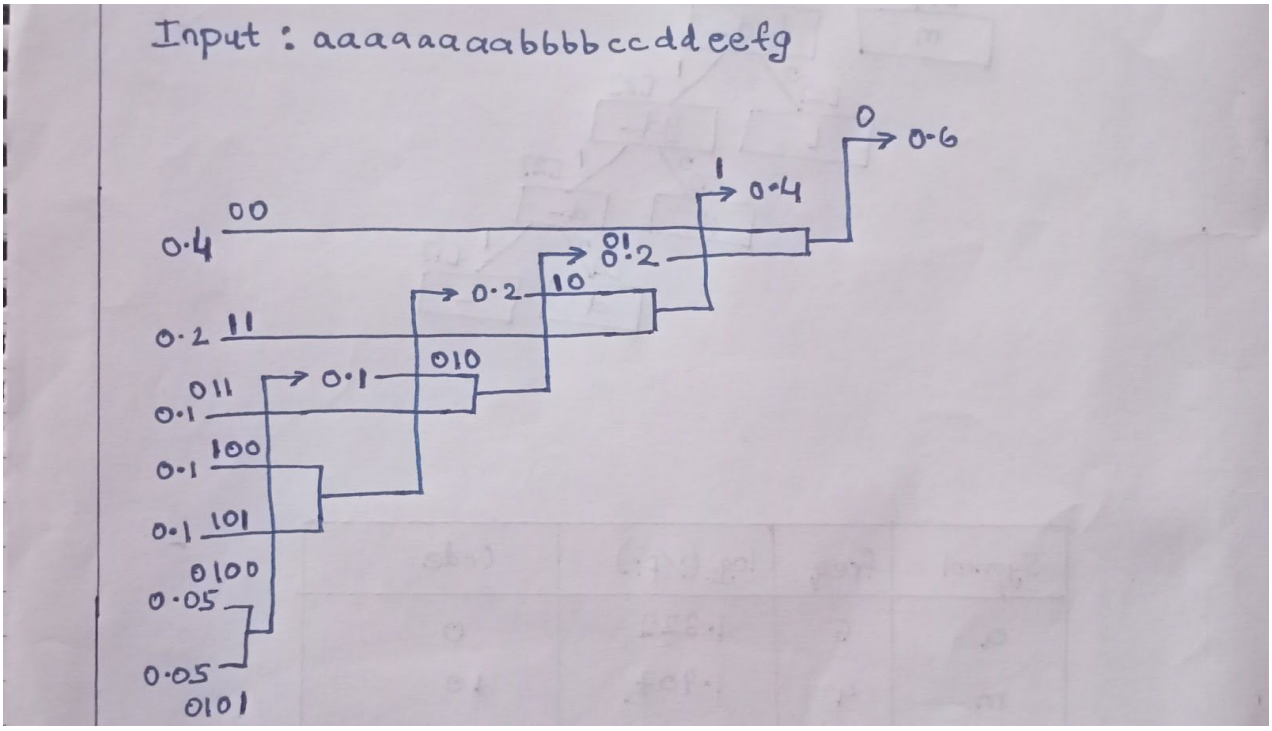
Huffman Encoding:

- Calculate the frequency of each character in the string.
- Sort the characters in increasing order of the frequency. These are stored in a priority queue
- Make each unique character as a leaf node
- Create an empty node. Assign the minimum frequency to the left child of the new node created and assign the second minimum frequency to the right child of the new node. Set the value of the new node as the sum of the above two minimum frequencies.
- Remove these two minimum frequencies from the queue and add the sum into the list of frequencies.
- Insert node new node into the tree.
- Repeat steps 3 to 5 for all the characters
- For each non-leaf node, assign 1 to the left edge and 0 to the right edge.

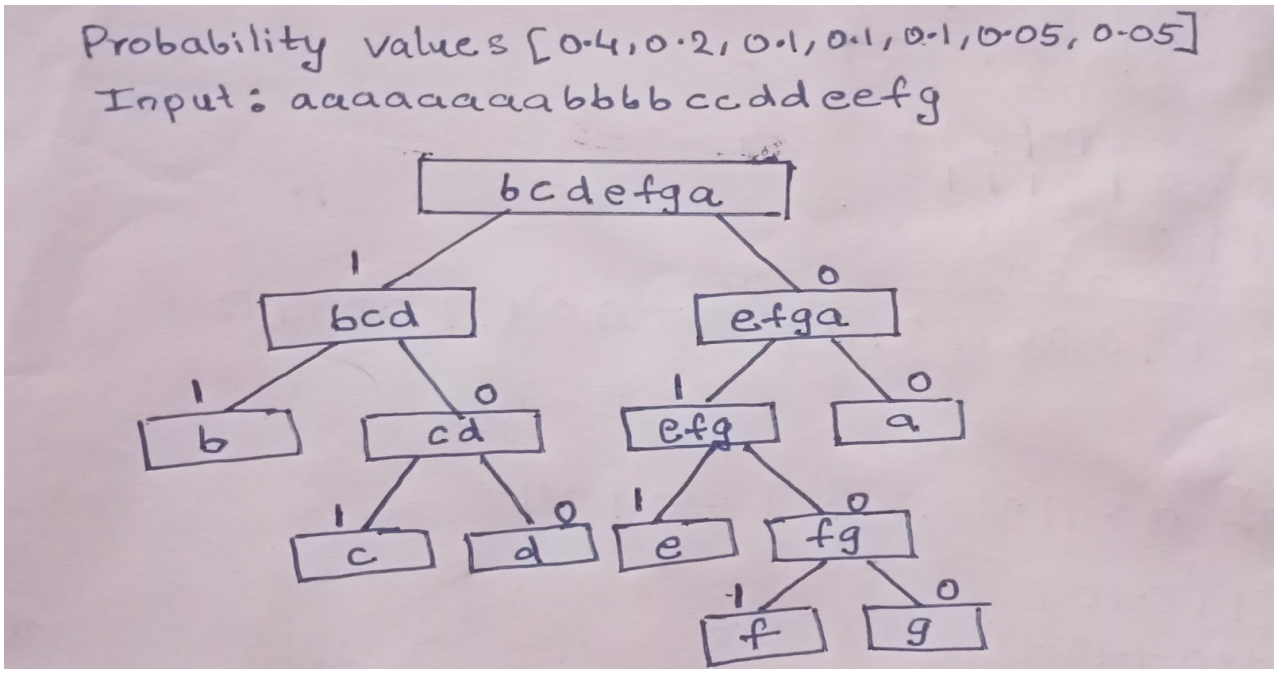
Huffman Decoding:

- To decode the encoded data we require Huffman tree and compressed code.
- Iterate through the binary encoded data,to find the corresponding encoded data.

Manual Calculation



Algorithm



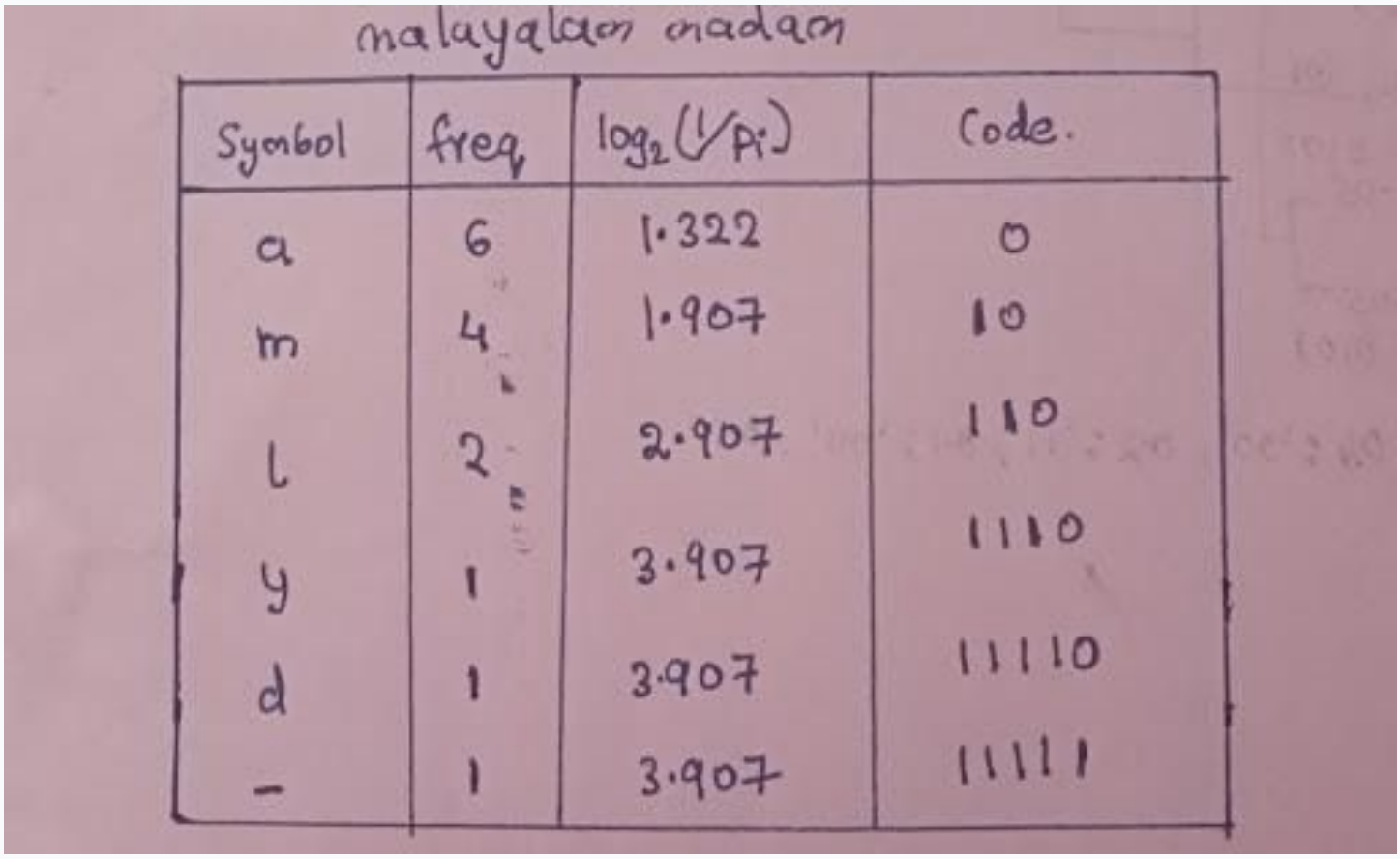
Shannon Encoding:

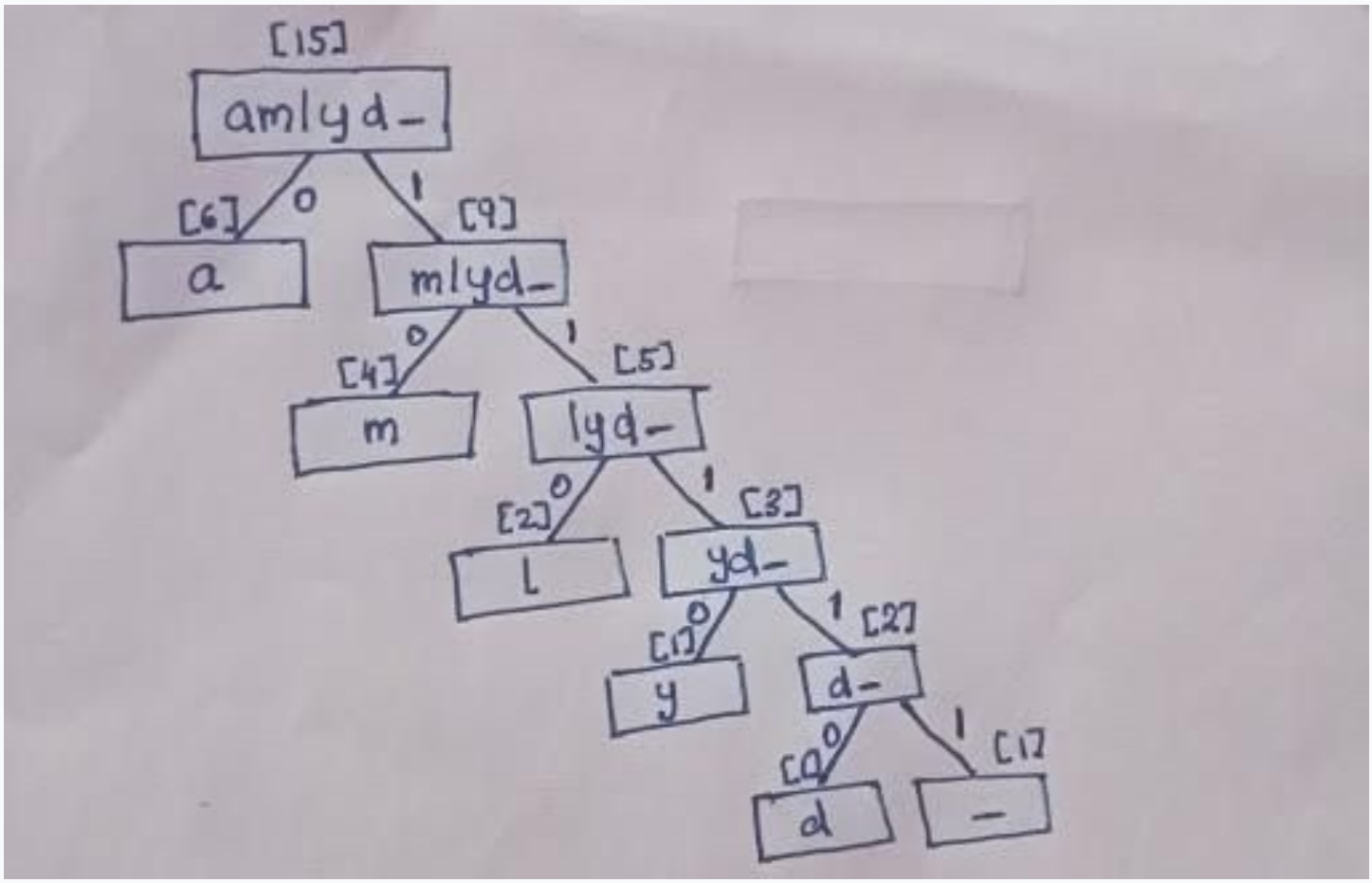
- Calculate the frequency of each character in the string.
- Sort the characters in increasing order of the frequency. These are stored in a priority queue
- Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.
- Assign values to the nodes (at the left of branch 0 and at the right 1)

Shannon Decoding:

- To decode the data the same code as Huffman decoding is used.

- To decode the encoded data we require Shannon tree and compressed code.
- Iterate through the binary encoded data,to find the corresponding encoded data.





Algorithm-Image Compression

- Convert the image file to numpy array values.
- Convert a numpy array to list and list to a string.
- Perform the Compression algorithms and get the uncompressed string back.
- Use regex to convert strings with brackets'[ ]' to only probability values with the same dimension of input image.
- Convert the string values to numpy array values.
- Convert numpy array to image file.

Huffman Code

Click here:sudhamshu091/Huffman-Encoding-and-Decoding (github.com)

```
print("Huffman Compression Program")
print("=============================================================")
h = int(input("Enter 1 if you want to enter in command window, 2 if you are using some file:"))
if h == 1:
    my_string = input("Enter the string you want to compress:")
elif h == 2:
    file = input("Enter the filename:")
    with open(file, 'r') as f:
        my_string = f.read()
else:
    print("You entered invalid input")
len_my_string = len(my_string)                    # taking user input
print ("Entered string is:",my_string)
print("Your data is ",len_my_string * 7 , "bits long")

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:]
nodes.sort()                                  # sorting according to frequency
```

```
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])
        newnode.append(combined_node1)
        newnode.append(combined_node2)              # combining the nodes to generate
pathways
        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("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:")
print(binary)                           # 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")

print("Compressed file generated as compressed.txt")
output = open("compressed.txt","w+")
print("Decoding.....")
output.write(binary)
```

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

print("Your UNCOMPRESSED data is:")         # uncompressed string
print(uncompressed_string)
```

Shannon Code

Click here:[sudhamshu091/Shannon-Encoding-and-Decoding (github.com)]

```
print("Shannon Compression Program")
print("==========================================================================")
import collections
h = int(input("Enter 1 if you want to enter in command window, 2 if you are using input as file :"))
if h == 1:
    message = input("Enter the string you want to compress:")
elif h == 2:
    file = input("Enter the filename:")
    with open(file, 'r') as f:
        message = f.read()
else:
    print("You entered invalid input")
print("Entered string is:",message)                    #taking input from user

c = {}
```

```
def create_list(message):
    list = dict(collections.Counter(message))
    for key, value in list.items():
        print(key, ' : ', value)                    #creating the sorted list according to the
probablity
    list_sorted = sorted(iter(list.items()), key = lambda k_v:(k_v[1],k_v[0]),reverse=True)
    final_list = []
    for key,value in list_sorted:
        final_list.append([key,value,''])
    return final_list

print("Shannon tree with merged pathways:")
def divide_list(list):
    if len(list) == 2:
        print([list[0]],[list[1]])              #printing merged pathways
        return [list[0]],[list[1]]
    else:
        n = 0
        for i in list:
            n+= i[1]
        x = 0
        distance = abs(2*x - n)
        j = 0
        for i in range(len(list)):              #shannon tree structure
            x += list[i][1]
            if distance < abs(2*x - n):
                j = i
    print(list[0:j+1], list[j+1:])              #printing merged pathways
    return list[0:j+1], list[j+1:]


def label_list(list):
    list1,list2 = divide_list(list)
    for i in list1:
        i[2] += '0'
        c[i[0]] = i[2]
    for i in list2:
        i[2] += '1'
        c[i[0]] = i[2]
```

```
    if len(list1)==1 and len(list2)==1:        #assigning values to the tree
        return
    label_list(list2)
    return c

code = label_list(create_list(message))
print("Shannon's Encoded Code:")
output = open("compressed.txt","w+")          # generating output binary
letter_binary = []
for key, value in code.items():
    print(key, ' : ', value)
    letter_binary.append([key,value])
print("Compressed file generated as compressed.txt")

for a in message:
    for key, value in code.items():
        if key in a:
            print(key, ' : ', value)
            output.write(value)
output = open("compressed.txt","r")
intermediate = output.readlines()
bitstring = ""
for digit in intermediate:
    bitstring = bitstring + digit
uncompressed_string =""
code =""
for digit in bitstring:
    code = code+digit
    pos=0
    for letter in letter_binary:              # decoding the binary and genrating original data
        if code ==letter[1]:
            uncompressed_string=uncompressed_string+letter_binary[pos] [0]
            code=""
        pos+=1

print("Your UNCOMPRESSED data is:")
print(uncompressed_string)
```

LZ77 Code

Click here:[sudhamshu091/LZ77-Encoding-and-Decoding (github.com)]

```
searchWindowSize = 0
previewWindowSize = 0

def longest_common_substring(s1, s2):
    maxLongest = 0
    offset = 0
    for i in range(0, len(s1)):
        longest = 0
        if ((i == len(s1) - len(s2) - 2)):
            break
        for j in range(0, len(s2)):
            if (i+j < len(s1)):
                if s1[i+j] == s2[j]:
                    longest = longest + 1
                    if (maxLongest < longest):
                        maxLongest = longest
                        offset = i
                else:
                    break
            else:
                break
    return maxLongest, offset

def encode_lz77(text, searchWindowSize, previewWindowSize):
    encodedNumbers = []
    encodedSizes = []
    encodedLetters = []
    i = 0
    while i < len(text):
        if i < previewWindowSize:
            encodedNumbers.append(0)
            encodedSizes.append(0)
            encodedLetters.append(text[i])
            i = i + 1
```

```
        else:
            previewString = text[i:i+previewWindowSize]
            searchWindowOffset = 0
            if (i < searchWindowSize):
                searchWindowOffset = i
            else:
                searchWindowOffset = searchWindowSize
            searchString = text[i - searchWindowOffset:i]
            result = longest_common_substring(searchString + previewString, previewString)
# searchString + prevString, prevString
            nextLetter = ''
            if (result[0] == len(previewString)):
                if (i + result[0] == len(text)):
                    nextLetter = ''
                else:
                    nextLetter = text[i+previewWindowSize]
            else:
                nextLetter = previewString[result[0]]
            if (result[0] == 0):
                encodedNumbers.append(0)
                encodedSizes.append(0)
                encodedLetters.append(nextLetter)
            else:
                encodedNumbers.append(searchWindowOffset - result[1])
                encodedSizes.append(result[0])
                encodedLetters.append(nextLetter)
            i = i + result[0] + 1
    return encodedNumbers, encodedSizes, encodedLetters

def decode_lz77(encodedNumbers, encodedSizes, encodedLetters):
    i = 0
    decodedString = []
    while i < len(encodedNumbers):
        if (encodedNumbers[i] == 0):
            decodedString.append(encodedLetters[i])
        else:
            currentSize = len(decodedString)
            for j in range(0, encodedSizes[i]):
                decodedString.append(decodedString[currentSize-encodedNumbers[i]+j])
```

```
            decodedString.append(encodedLetters[i])
        i = i+1
    return decodedString


print("LZ77 Compression Algorithm")
print("================================================================
======")
h = int(input("Enter 1 if you want to enter input in command window, 2 if you are using
some file:"))
if h == 1:
    stringToEncode = input("Enter the string you want to compress:")
elif h == 2:
    file = input("Enter the filename:")
    with open(file, 'r') as f:
        stringToEncode = f.read()
else:
    print("You entered invalid input")
print ("Enetered string is:",stringToEncode)
searchWindowSize = int(input("Enter the Search Window Size:"))
previewWindowSize = int(input("Enter the Preview Window Size:"))
[encodedNumbers, encodedSizes, encodedLetters] = encode_lz77(stringToEncode,
searchWindowSize, previewWindowSize)
a =[encodedNumbers, encodedSizes, encodedLetters]
print("Compressed file generated as compressed.txt")
output = open("compressed.txt","w+")
output.write(str(a))
print("Encoded string: ", end="")
i = 0
while i < len(encodedNumbers):
    print ("{",encodedNumbers[i],":", encodedSizes[i],":", encodedLetters[i],"}",end = " ")
    i = i + 1
print("\n")
decodedString = decode_lz77(encodedNumbers, encodedSizes, encodedLetters)
print("Decoded string:", "".join(decodedString))
```

Huffman Encoding and Decoding for Image Compression

[sudhamshu091/Huffman-Encoding-Decoding-For-Image-Compression (github.com)]

```
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")
```

Shannon Encoding and Decoding for Image Compression

[sudhamshu091/Shannon-Fano-Encoding-and-Decoding-for-Image-Compression (github.com)]

```
import re
import numpy as np
from PIL import Image
print("Shannon Image Compression Program")
print("==========================================================================")
import collections
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)
    sudhi = my_string
    shape = my_string.shape
    print ("Enetered string is:",my_string)
    message = 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)
    sudhi = my_string
    message = str(my_string.tolist())
else:
    print("You entered invalid input")                    #taking input from user

c = {}
```

```
def create_list(message):
    list = dict(collections.Counter(message))
    for key, value in list.items():
        print(key, ' : ', value)                    #creating the sorted list according to the
probablity
    list_sorted = sorted(iter(list.items()), key = lambda k_v:(k_v[1],k_v[0]),reverse=True)
    final_list = []
    for key,value in list_sorted:
        final_list.append([key,value,''])
    return final_list

print("Shannon tree with merged pathways:")
def divide_list(list):
    if len(list) == 2:
        print([list[0]],"::",[list[1]])             #printing merged pathways
        return [list[0]],[list[1]]
    else:
        n = 0
        for i in list:
            n+= i[1]
        x = 0
        distance = abs(2*x - n)
        j = 0
        for i in range(len(list)):                  #shannon tree structure
            x += list[i][1]
            if distance < abs(2*x - n):
                j = i
    print(list[0:j+1],"::",list[j+1:])              #printing merged pathways
    return list[0:j+1], list[j+1:]


def label_list(list):
    list1,list2 = divide_list(list)
    for i in list1:
        i[2] += '0'
        c[i[0]] = i[2]
    for i in list2:
        i[2] += '1'
        c[i[0]] = i[2]
```

```
    if len(list1)==1 and len(list2)==1:        #assigning values to the tree
        return
    label_list(list2)
    return c

code = label_list(create_list(message))
print("Shannon's Encoded Code:")
output = open("compressed.txt","w+")          # generating output binary
letter_binary = []
for key, value in code.items():
    print(key, ' : ', value)
    letter_binary.append([key,value])
print("Compressed file generated as compressed.txt")

for a in message:
    for key, value in code.items():
        if key in a:
            print(key, ' : ', value)
            output.write(value)
output = open("compressed.txt","r")
intermediate = output.readlines()
bitstring = ""
for digit in intermediate:
    bitstring = bitstring + digit
uncompressed_string =""
code =""
for digit in bitstring:
    code = code+digit
    pos=0
    for letter in letter_binary:              # decoding the binary and genrating original data
        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 sudhi.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")
```

LZ77 Encoding and Decoding for Image Compression

[sudhamshu091/LZ77-Encoding-and-Decoding-for-Image-Compression (github.com)]

```
import re
import numpy as np
from PIL import Image
searchWindowSize = 0
previewWindowSize = 0

def longest_common_substring(s1, s2):
    maxLongest = 0
```

```
    offset = 0
    for i in range(0, len(s1)):
        longest = 0
        if ((i == len(s1) - len(s2) - 2)):
            break
        for j in range(0, len(s2)):
            if (i+j < len(s1)):
                if s1[i+j] == s2[j]:
                    longest = longest + 1
                    if (maxLongest < longest):
                        maxLongest = longest
                        offset = i
                else:
                    break
            else:
                break
    return maxLongest, offset

def encode_lz77(text, searchWindowSize, previewWindowSize):
    encodedNumbers = []
    encodedSizes = []
    encodedLetters = []
    i = 0
    while i < len(text):
        if i < previewWindowSize:
            encodedNumbers.append(0)
            encodedSizes.append(0)
            encodedLetters.append(text[i])
            i = i + 1
        else:
            previewString = text[i:i+previewWindowSize]
            searchWindowOffset = 0
            if (i < searchWindowSize):
                searchWindowOffset = i
            else:
                searchWindowOffset = searchWindowSize
            searchString = text[i - searchWindowOffset:i]
            result = longest_common_substring(searchString + previewString, previewString)
# searchString + prevString, prevString
```

```
            nextLetter = ''
            if (result[0] == len(previewString)):
                if (i + result[0] == len(text)):
                    nextLetter = ''
                else:
                    nextLetter = text[i+previewWindowSize]
            else:
                nextLetter = previewString[result[0]]
            if (result[0] == 0):
                encodedNumbers.append(0)
                encodedSizes.append(0)
                encodedLetters.append(nextLetter)
            else:
                encodedNumbers.append(searchWindowOffset - result[1])
                encodedSizes.append(result[0])
                encodedLetters.append(nextLetter)
            i = i + result[0] + 1
    return encodedNumbers, encodedSizes, encodedLetters

def decode_lz77(encodedNumbers, encodedSizes, encodedLetters):
    i = 0
    decodedString = []
    while i < len(encodedNumbers):
        if (encodedNumbers[i] == 0):
            decodedString.append(encodedLetters[i])
        else:
            currentSize = len(decodedString)
            for j in range(0, encodedSizes[i]):
                decodedString.append(decodedString[currentSize-encodedNumbers[i]+j])
            decodedString.append(encodedLetters[i])
        i = i+1
    return decodedString


print("LZ77 Compression Algorithm")
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)
    sudhi = my_string
    shape = my_string.shape
    print ("Enetered string is:",my_string)
    stringToEncode = 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)
    sudhi = my_string
    stringToEncode = str(my_string.tolist())
else:
    print("You entered invalid input")                    # taking user input


print ("Enetered string is:",stringToEncode)
searchWindowSize = int(input("Enter the Search Window Size:"))
previewWindowSize = int(input("Enter the Preview Window Size:"))
[encodedNumbers, encodedSizes, encodedLetters] = encode_lz77(stringToEncode,
searchWindowSize, previewWindowSize)
a =[encodedNumbers, encodedSizes, encodedLetters]
print("Compressed file generated as compressed.txt")
output = open("compressed.txt","w+")
output.write(str(a))
print("Encoded string: ", end="")
i = 0
while i < len(encodedNumbers):
    print ("{",encodedNumbers[i],":", encodedSizes[i],":", encodedLetters[i],"}",end = " ")
    i = i + 1
print("\n")
decodedString = decode_lz77(encodedNumbers, encodedSizes, encodedLetters)
uncompressed_string ="".join(decodedString)
print("Decoded string:", "".join(decodedString))

if h == 1:
    temp = re.findall(r'\d+', uncompressed_string)
    res = list(map(int, temp))
```

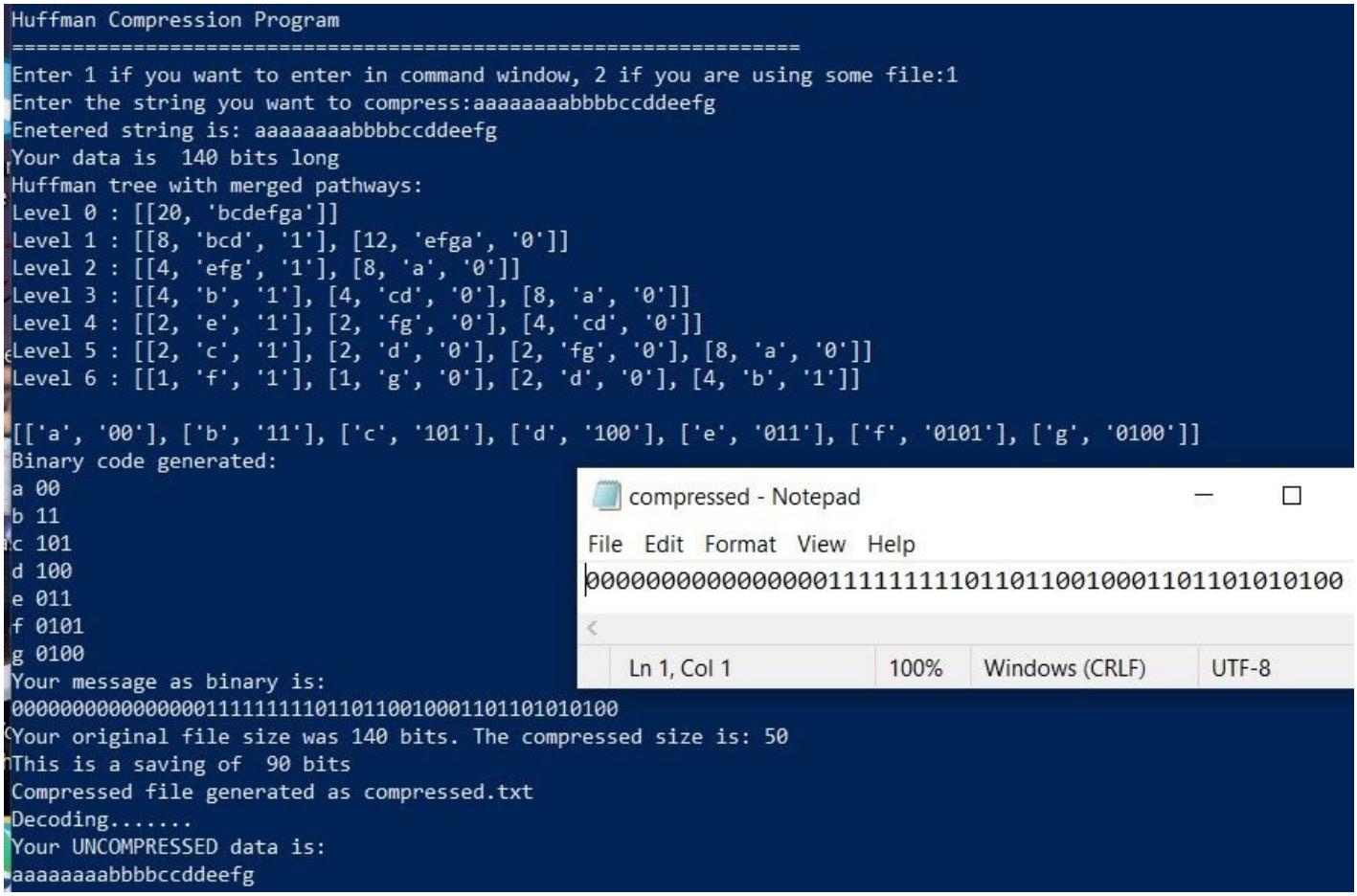
```
    res = np.array(res)
    res = res.astype(np.uint8)
    res = np.reshape(res, shape)
    print(res)
    print("Observe the shapes and input and output arrays are matching or not")
    print("Input image dimensions:",shape)
    print("Output image dimensions:",res.shape)
    data = Image.fromarray(res)
    data.save('uncompressed.png')
    if sudhi.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")
```

Huffman Results

Executed Huffman Encoding and decoding program for probability values

[0.4, 0.2, 0.1, 0.1, 0.1, 0.05, 0.05] generated codes 00,11,101,100,011,0101,0100
[8*a,4*b,2*c,2*d,2*e, 1*f, 1*g]
compressed file generated as
00000000000000001111111110110110010001101101010100

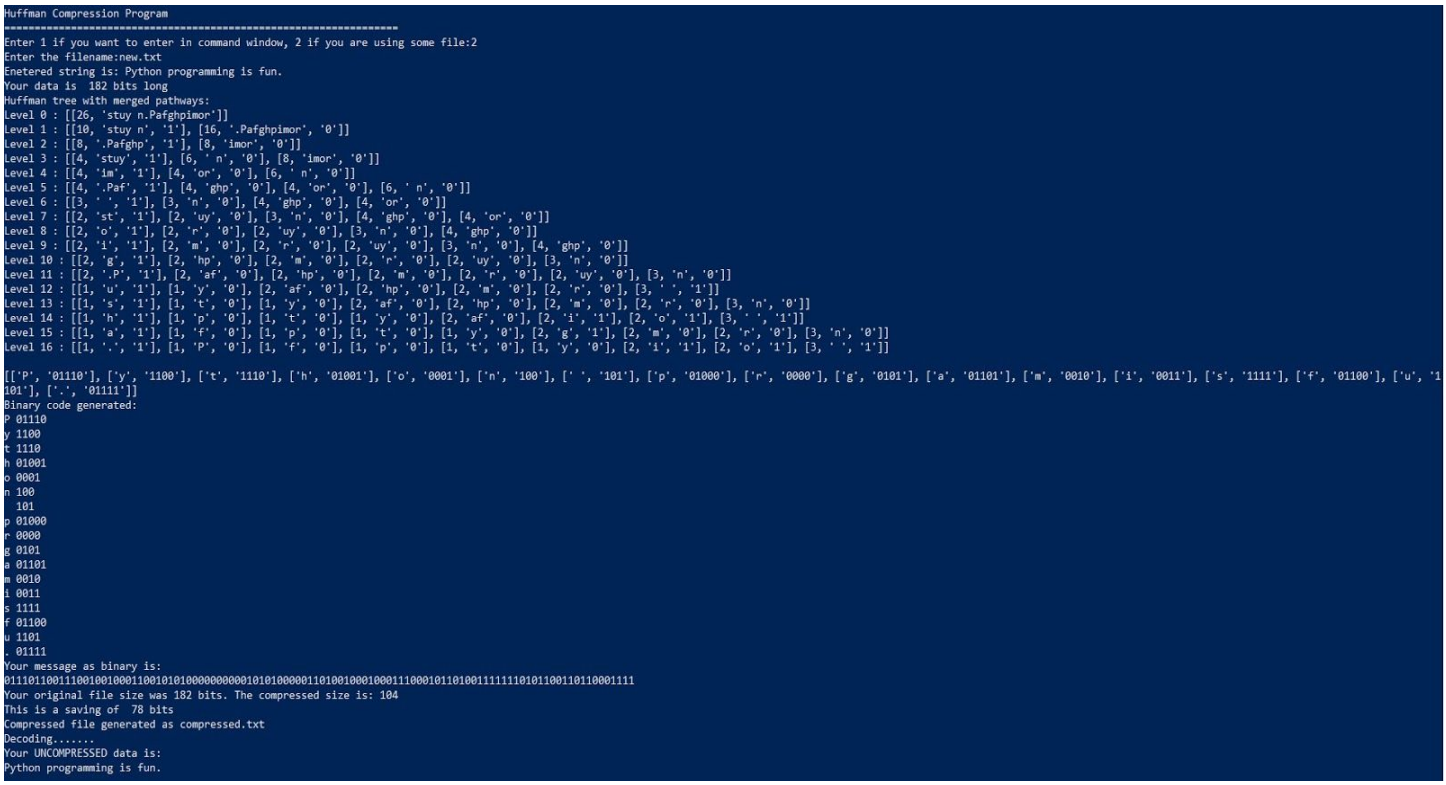
Decoded data:aaaaaaaabbbbccddeefg



Executed Huffman Encoding and decoding program for the following input: ”Python programming is fun.”

[['P', '01110'], ['y', '1100'], ['t', '1110'], ['h', '01001'], ['o', '0001'],['n', '100'], [' ', '101'], ['p', '01000'], ['r', '0000'], ['g', '0101'], ['a', '01101'],['m', '0010'], ['i', '0011'], ['s', '1111'], ['f',
'01100'], ['u', '1101'], ['.', '01111']]
compressed file generated as:
01110110011100100100011001010100000000001010100000110100100010001110001011010011111111010110011011000111 1

Decoded data:Python programming is fun.



Shannon Fano Results

Executed Shannon Encoding and decoding for input “malayalam madam”

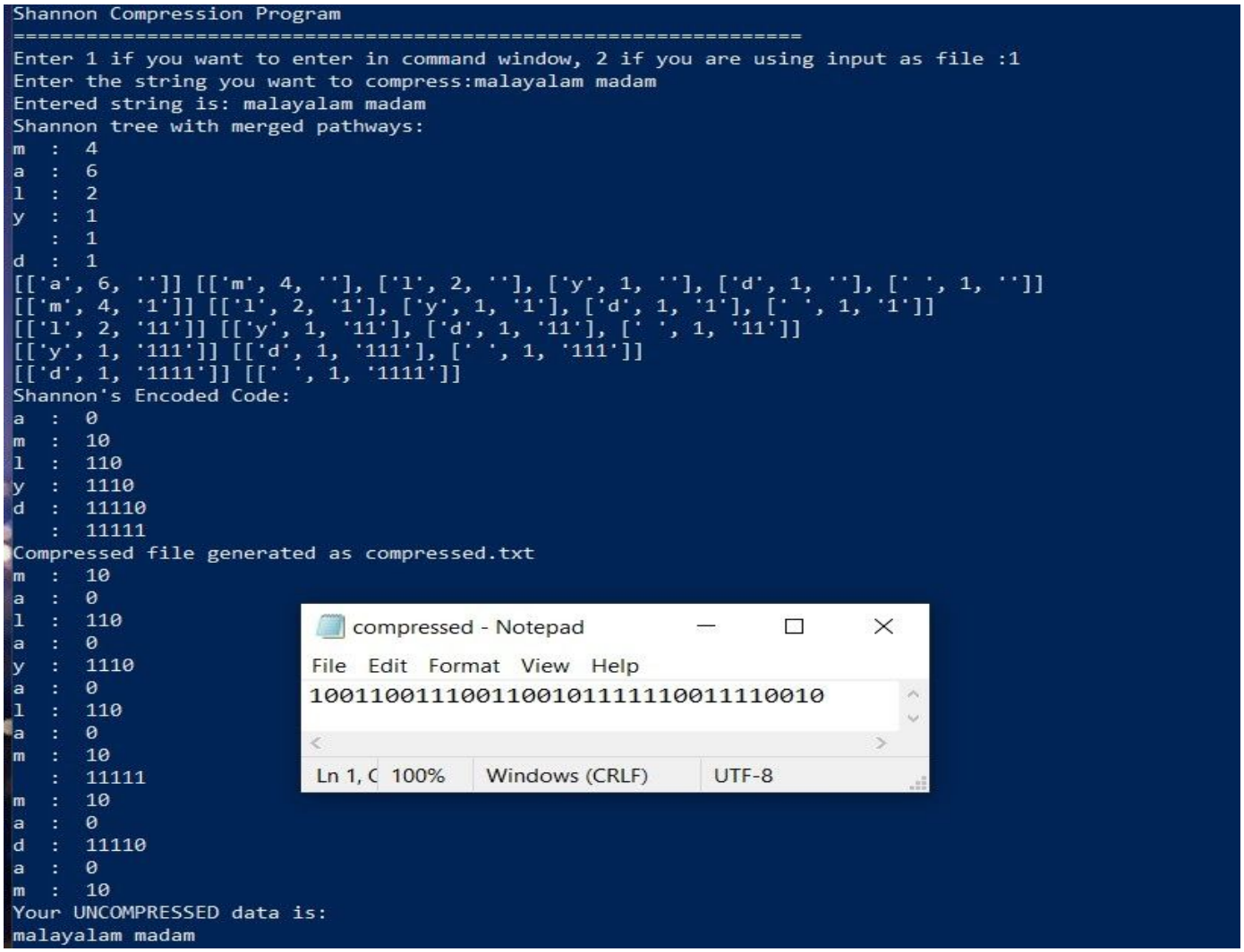
for probability values [0.266, 0.4, 0.1333, 0.0666,0.0666,0.0666]

[a , m , l , y , d , “ “]

Generated Codes: 0,10,110,1110,11110,11111

Binary code :1001100111001100101111110011110010

Decoded data: malayalam madam



LZ77 Results

```
LZ77 Compression Algorithm
================================================================
Enter 1 if you want to enter input in command window, 2 if you are using some file:1
Enter the string you want to compress:cabracadabrarrarrad
Enetered string is: cabracadabrarrarrad
Enter the Search Window Size:7
Enter the Preview Window Size:6
Compressed file generated as compressed.txt
Encoded string: { 0 : 0 : c } { 0 : 0 : a } { 0 : 0 : b } { 0 : 0 : r } { 0 : 0 : a } { 0 : 0 : c } { 5 : 1 : d }
 { 7 : 4 : r } { 3 : 5 : d }

Decoded string: cabracadabrarrarrad
```

compressed - Notepad

File Edit Format View Help

```
[[0, 0, 0, 0, 0, 0, 5, 7, 3], [0, 0, 0, 0, 0, 0, 1, 4, 5],
['c', 'a', 'b', 'r', 'a', 'c', 'd', 'r', 'd']]
```

Image Compression Results

Input

Output

Input

Output

Input

Output

Discussions

- LZ77 is much better compared to Huffman and Shannon in terms of speed
- Huffman is advantageous Compared to Shannon's Algorithm
- The Image Compression algorithm used by us is not the most efficient one,but we have just tried to compress image files.
- LZ77 has proven better results for image compression
- LZ78 algorithm is also a great method for data compression

Applications

- Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.
- Multimedia codecs like JPEG, PNG and MP3 use Huffman encoding.
- Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.
- Brotli compression algorithm by Google compresses data using a combination of a modern variant of the LZ77 algorithm, Huffman coding and 2nd order context modeling.
- Image Compression