**Huffman Encoding**

class Nodes:

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

# probability of the symbol

self.probability = probability

# the symbol

self.symbol = symbol

# the left node

self.left = left

# the right node

self.right = right

# the tree direction (0 or 1)

self.code = ''

""" A supporting function in order to calculate the probabilities of symbols in specified data """

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

""" A supporting function in order to print the codes of symbols by travelling a Huffman Tree """

the\_codes = dict()

def CalculateCodes(node, value = ''):

# a huffman code for current node

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

""" A supporting function in order to get the encoded result """

def OutputEncoded(the\_data, coding):

encodingOutput = []

for element in the\_data:

# print(coding[element], end = '')

encodingOutput.append(coding[element])

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

return the\_string

""" A supporting function in order to calculate the space difference between compressed and non compressed data"""

def TotalGain(the\_data, coding):

# total bit space to store the data before compression

beforeCompression = len(the\_data) \* 8

afterCompression = 0

the\_symbols = coding.keys()

for symbol in the\_symbols:

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

# calculating how many bit is required for that symbol in total

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 = []

# converting symbols and probabilities into huffman tree nodes

for symbol in the\_symbols:

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

while len(the\_nodes) > 1:

# sorting all the nodes in ascending order based on their probability

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

# for node in nodes:

# print(node.symbol, node.prob)

# picking two smallest nodes

right = the\_nodes[0]

left = the\_nodes[1]

left.code = 0

right.code = 1

# combining the 2 smallest nodes to create new node

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))