Q: What is a Python implementation of Huffman Encoding?

A:

# Calculating frequency

freq = {}

for c in string:

if c in freq:

freq[c] += 1

else:

freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

nodes = freq

while len(nodes) > 1:

(key1, c1) = nodes[-1]

(key2, c2) = nodes[-2]

nodes = nodes[:-2]

node = NodeTree(key1, key2)

nodes.append((node, c1 + c2))

nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman\_code\_tree(nodes[0][0])

print(' Char | Huffman code ')

print('----------------------')

for (char, frequency) in freq:

print(' %-4r |%12s' % (char, huffmanCode[char]))

Q: What is a Python implementation of Huffman Encoding?

A:

# Node of a Huffman Tree

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 = "AAAAAAABBCCCCCCDDDEEEEEEEEE"

print(the\_data)

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

print("Encoded output", encoding)

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

Q: What is a Python implementation of Huffman Encoding?

A:

from collections import Counter

class NodeTree(object):

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

self.left = left

self.right = right

def children(self):

return self.left, self.right

def \_\_str\_\_(self):

return self.left, self.right

def huffman\_code\_tree(node, binString=''):

'''

Function to find Huffman Code

'''

if type(node) is str:

return {node: binString}

(l, r) = node.children()

d = dict()

d.update(huffman\_code\_tree(l, binString + '0'))

d.update(huffman\_code\_tree(r, binString + '1'))

return d

def make\_tree(nodes):

'''

Function to make tree

:param nodes: Nodes

:return: Root of the tree

'''

while len(nodes) > 1:

(key1, c1) = nodes[-1]

(key2, c2) = nodes[-2]

nodes = nodes[:-2]

node = NodeTree(key1, key2)

nodes.append((node, c1 + c2))

nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

return nodes[0][0]

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

string = 'BCAADDDCCACACAC'

freq = dict(Counter(string))

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

node = make\_tree(freq)

encoding = huffman\_code\_tree(node)

for i in encoding:

print(f'{i} : {encoding[i]}')

Q: What is a Python implementation of Huffman Encoding?

A:

# A Huffman Tree Node

class Node:

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

# probability of symbol

self.prob = prob

# symbol

self.symbol = symbol

# left node

self.left = left

# right node

self.right = right

# tree direction (0/1)

self.code = ''

""" A helper function to print the codes of symbols by traveling Huffman Tree"""

codes = dict()

def Calculate\_Codes(node, val=''):

# huffman code for current node

newVal = val + str(node.code)

if(node.left):

Calculate\_Codes(node.left, newVal)

if(node.right):

Calculate\_Codes(node.right, newVal)

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

codes[node.symbol] = newVal

return codes

""" A helper function to calculate the probabilities of symbols in given data"""

def Calculate\_Probability(data):

symbols = dict()

for element in data:

if symbols.get(element) == None:

symbols[element] = 1

else:

symbols[element] += 1

return symbols

""" A helper function to obtain the encoded output"""

def Output\_Encoded(data, coding):

encoding\_output = []

for c in data:

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

encoding\_output.append(coding[c])

string = ''.join([str(item) for item in encoding\_output])

return string

""" A helper function to calculate the space difference between compressed and non compressed data"""

def Total\_Gain(data, coding):

before\_compression = len(data) \* 8 # total bit space to stor the data before compression

after\_compression = 0

symbols = coding.keys()

for symbol in symbols:

count = data.count(symbol)

after\_compression += count \* len(coding[symbol]) #calculate how many bit is required for that symbol in total

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

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

def Huffman\_Encoding(data):

symbol\_with\_probs = Calculate\_Probability(data)

symbols = symbol\_with\_probs.keys()

probabilities = symbol\_with\_probs.values()

print("symbols: ", symbols)

print("probabilities: ", probabilities)

nodes = []

# converting symbols and probabilities into huffman tree nodes

for symbol in symbols:

nodes.append(Node(symbol\_with\_probs.get(symbol), symbol))

while len(nodes) > 1:

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

nodes = sorted(nodes, key=lambda x: x.prob)

# for node in nodes:

# print(node.symbol, node.prob)

# pick 2 smallest nodes

right = nodes[0]

left = nodes[1]

left.code = 0

right.code = 1

# combine the 2 smallest nodes to create new node

newNode = Node(left.prob+right.prob, left.symbol+right.symbol, left, right)

nodes.remove(left)

nodes.remove(right)

nodes.append(newNode)

huffman\_encoding = Calculate\_Codes(nodes[0])

print("symbols with codes", huffman\_encoding)

Total\_Gain(data, huffman\_encoding)

encoded\_output = Output\_Encoded(data,huffman\_encoding)

return encoded\_output, nodes[0]

def Huffman\_Decoding(encoded\_data, huffman\_tree):

tree\_head = huffman\_tree

decoded\_output = []

for x in encoded\_data:

if x == '1':

huffman\_tree = huffman\_tree.right

elif x == '0':

huffman\_tree = huffman\_tree.left

try:

if huffman\_tree.left.symbol == None and huffman\_tree.right.symbol == None:

pass

except AttributeError:

decoded\_output.append(huffman\_tree.symbol)

huffman\_tree = tree\_head

string = ''.join([str(item) for item in decoded\_output])

return string

""" First Test """

data = "AAAAAAABCCCCCCDDEEEEE"

print(data)

encoding, tree = Huffman\_Encoding(data)

print("Encoded output", encoding)

print("Decoded Output", Huffman\_Decoding(encoding,tree))

Q: How do we fix this Python implementation of Huffman Encoding without library functions?

class Node:

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

self.freq = freq

self.symbol = symbol

self.left = left

self.right = right

self.code = None

def \_\_str\_\_(self):

return f"{self.symbol}: {self.freq} ({self.code})"

def calculate\_frequency(data):

frequency = {}

for symbol in data:

if symbol not in frequency:

frequency[symbol] = 0

frequency[symbol] += 1

return frequency

def print\_tree(node):

if node is None:

return

print(str(node))

print\_tree(node.left)

print\_tree(node.right)

def build\_huffman\_tree(data, table=False):

if table:

frequencies = table

else:

frequencies = calculate\_frequency(data)

nodes = [Node(freq, symbol) for symbol, freq in frequencies.items()]

nodes = sorted(nodes, key=lambda x: (x.freq, x.symbol.lower()))

while len(nodes) > 1:

left = nodes[0]

right = nodes[1]

new\_node = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)

nodes.remove(left)

nodes.remove(right)

nodes.append(new\_node)

nodes = sorted(nodes, key=lambda x: (x.freq, x.symbol.lower()))

return nodes[0]

def build\_tree\_from\_table(table):

nodes = [Node(freq, symbol) for symbol, freq in table.items()]

nodes = sorted(nodes, key=lambda x: (x.freq, len(x.symbol) > 1, x.symbol.lower()))

while len(nodes) > 1:

left = nodes[0]

right = nodes[1]

new\_node = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)

nodes.remove(left)

nodes.remove(right)

nodes.append(new\_node)

nodes = sorted(nodes, key=lambda x: (x.freq, len(x.symbol) > 1, x.symbol.lower()))

return nodes[0]

def huffman\_encoding(node, code=''):

if node is None:

return

if node.left is None and node.right is None:

node.code = code

huffman\_encoding(node.left, code + '0')

huffman\_encoding(node.right, code + '1')

def huffman\_decode(encoded\_data, table):

root = build\_tree\_from\_table(table)

decoded\_data = ''

current\_node = root

for bit in encoded\_data:

if bit == '0':

current\_node = current\_node.left

elif bit == '1':

current\_node = current\_node.right

if current\_node.left is None and current\_node.right is None:

decoded\_data += current\_node.symbol

current\_node = root

return decoded\_data

def print\_encoding(data, tree):

encoding = {}

huffman\_encoding(tree)

def fill\_encoding(node):

if node is None:

return

if node.left is None and node.right is None:

encoding[node.symbol] = node.code

fill\_encoding(node.left)

fill\_encoding(node.right)

fill\_encoding(tree)

encoded\_data = ''

for symbol in data:

if symbol != ' ':

encoded\_data += encoding[symbol]

return encoded\_data # Add this line to return the encoded data

A: