YAYASAN ISLAM BOJONG

SEKOLAH TINGGI TEKNOLOGI YBSI TASIKMALAYA

KompleksMayasari Plasa Telp/Facs 0265-231607 Tasikmalaya

Website :[www.sttybsi.ac.id](http://www.sttybsi.ac.id/)

MataKuliah : MULTIMEDIA

Dosen : Rudi Hermawan, ST,. MOS

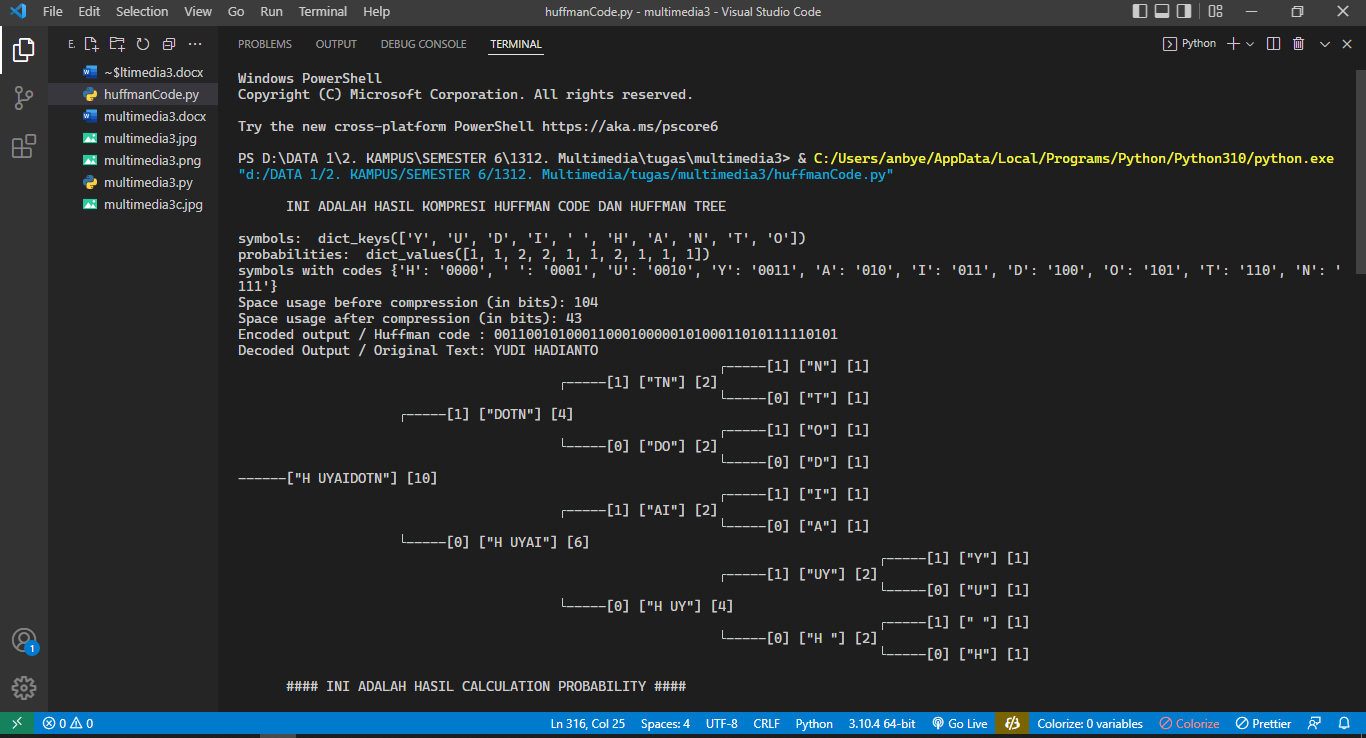
Nama : **Yudi Hadianto**  
NIM : 19060006

**LINK YOUTUBE**

**Quiz 3**

[**https://www.youtube.com/watch?v=g9l2isk9p\_8**](https://www.youtube.com/watch?v=g9l2isk9p_8)

**HUFMAN CODE**

****

1. **Space usage before compression (in bits): 104 bits**
2. **Space usage after compression (in bits): 43 bit**

# 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 = ''

# tree parent

self.parent = ''

""" 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 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 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):

# total bit space to stor the data before compression

before\_compression = len(data) \* 8

after\_compression = 0

symbols = coding.keys()

for symbol in symbols:

count = data.count(symbol)

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

after\_compression += count \* len(coding[symbol])

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:

# pick 2 smallest nodes

right = nodes[0]

left = nodes[1]

left.code = 0

right.code = 1

left.parent = left.symbol+right.symbol

right.parent = left.symbol+right.symbol

# 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:

# print(x)

if x == '1':

# print('before',huffman\_tree.symbol)

huffman\_tree = huffman\_tree.right

# print('after',huffman\_tree.symbol)

elif x == '0':

# print('before',huffman\_tree.symbol)

huffman\_tree = huffman\_tree.left

# print('after',huffman\_tree.symbol)

try:

# print('left.symbol',huffman\_tree.left.symbol)

# print('right.symbol',huffman\_tree.right.symbol)

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

pass

except AttributeError:

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

# print(decoded\_output)

huffman\_tree = tree\_head

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

return string

""" Function to visualize huffman tree """

# It does reverse inorder traversal

def print\_tree\_util(root, space, markers):

# Base case

if (root == None):

return

# Increase distance between levels

space += markers[0]

# Process right child first

print\_tree\_util(root.right, space, markers)

# Print current node after space

print(''.rjust((space), ' '), end="")

if 1 in {root.code}: # if node is on the right

print(f'┌-----[{root.code}] ["{root.symbol}"] [{len(root.symbol)}]')

elif 0 in {root.code}: # if node is on the left

print(f'└-----[{root.code}] ["{root.symbol}"] [{len(root.symbol)}]')

else: # if node is root

print(f'------["{root.symbol}"] [{len(root.symbol)}]')

# Process left child

print\_tree\_util(root.left, space, markers)

# Wrapper over print\_tree\_util()

def print\_huffman\_tree(root, spaces=None):

if not spaces:

# sets the distances between levels by the length of the root

markers = [(len(f'[ ] ["{root.symbol}"] [{len(root.symbol)}]'))]

space = 0 - markers[0]

else:

# users sets the space

markers = [spaces]

space = 0 - spaces

print\_tree\_util(root, space, markers)

# DRIVERCODE

print('''

INI ADALAH HASIL KOMPRESI HUFFMAN CODE DAN HUFFMAN TREE

''')

data = "YUDI HADIANTO"

encoding, huffman\_trees = Huffman\_Encoding(data)

print("Encoded output / Huffman code :", encoding)

print("Decoded Output / Original Text:",

Huffman\_Decoding(encoding, huffman\_trees))

print\_huffman\_tree(huffman\_trees, 20)

# ENCODING

def Calculate\_Probability(data):

symbols = dict()

for element in data:

print(element)

if symbols.get(element) == None:

symbols[element] = 1

else:

symbols[element] += 1

print(symbols)

return symbols

print('''

#### INI ADALAH HASIL CALCULATION PROBABILITY ####

''')

data = "YUDI HADIANTO"

print(data)

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

# CALCULATE PROBABILITY

def Calculate\_Probability(data):

symbols = dict()

for element in data:

# print(element)

if symbols.get(element) == None:

symbols[element] = 1

else:

symbols[element] += 1

# print(symbols)

return symbols

print('''

#### INI ADALAH HASIL CALCULATION PROBABILITY ####

''')

data = "YUDI HADIANTO"

print(data)

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

print('symbol\_with\_probs :', symbol\_with\_probs)

symbols = symbol\_with\_probs.keys()

probabilities = symbol\_with\_probs.values()

print("symbols: ", symbols)

print("probabilities: ", probabilities)

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 = ''

# tree parent

self.parent = ''

# converts into object

nodes = []

for symbol in symbols:

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

# print the entire object

for i in nodes:

print(i.\_\_dict\_\_)

def Calculate\_Probability(data):

symbols = dict()

for element in data:

# print(element)

if symbols.get(element) == None:

symbols[element] = 1

else:

symbols[element] += 1

# print(symbols)

return symbols

data = "YUDI HADIANTO"

print(data)

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

print('symbol\_with\_probs :', symbol\_with\_probs)

symbols = symbol\_with\_probs.keys()

probabilities = symbol\_with\_probs.values()

print("symbols: ", symbols)

print("probabilities: ", probabilities)

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 = ''

# tree parent

self.parent = ''

# converts into object

nodes = []

for symbol in symbols:

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

# print the entire object

# for i in nodes:

# print(i.\_\_dict\_\_)

# looping to create huffman tree

i = 1

while len(nodes) > 1:

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

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

print('\niteration number : ', i)

# to visualize the making of huffman tree

for node in nodes:

print(node.symbol, node.prob)

try:

print('left', node.left.\_\_dict\_\_)

print('right', node.right.\_\_dict\_\_)

except:

pass

# pick 2 smallest nodes

right = nodes[0]

left = nodes[1]

left.code = 0

right.code = 1

left.parent = left.symbol+right.symbol

right.parent = left.symbol+right.symbol

# 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)

i += 1

# loop ended

print('\ncreated huffman tree : ')

print(nodes[0].\_\_dict\_\_)