|  |  |  |
| --- | --- | --- |
| NIM | : | 19060036 |
| Nama | : | Yan Yan Permana |
| Kelas | : | Reguler Sore |
| Jurusan | : | Teknik Informatika |

Link video mengubah gambar menjadi kode ascii: <https://youtu.be/8rdkvBf8JbI>

1. """ 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)

1. 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 = " yyyyyyaaaaaaannnnnyaaaaaaaannnnn"

# 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\_\_)

codes = dict()

def Calculate\_Codes(node, val=''):

    # huffman code for current node

    # print('node code: ',node.code)

    newVal = val + str(node.code)

    # print(node.symbol,newVal)

    # print(node.\_\_dict\_\_)

    if(node.left):

        # print("left True", node.left.\_\_dict\_\_)

        Calculate\_Codes(node.left, newVal)

    if(node.right):

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

        Calculate\_Codes(node.right, newVal)

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

        # print(codes)

        codes[node.symbol] = newVal

    # print(codes)

    return codes

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

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

""" 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()

    print('symbols: ',symbols)

    print('coding: ', coding)

    for symbol in symbols:

        count = data.count(symbol)

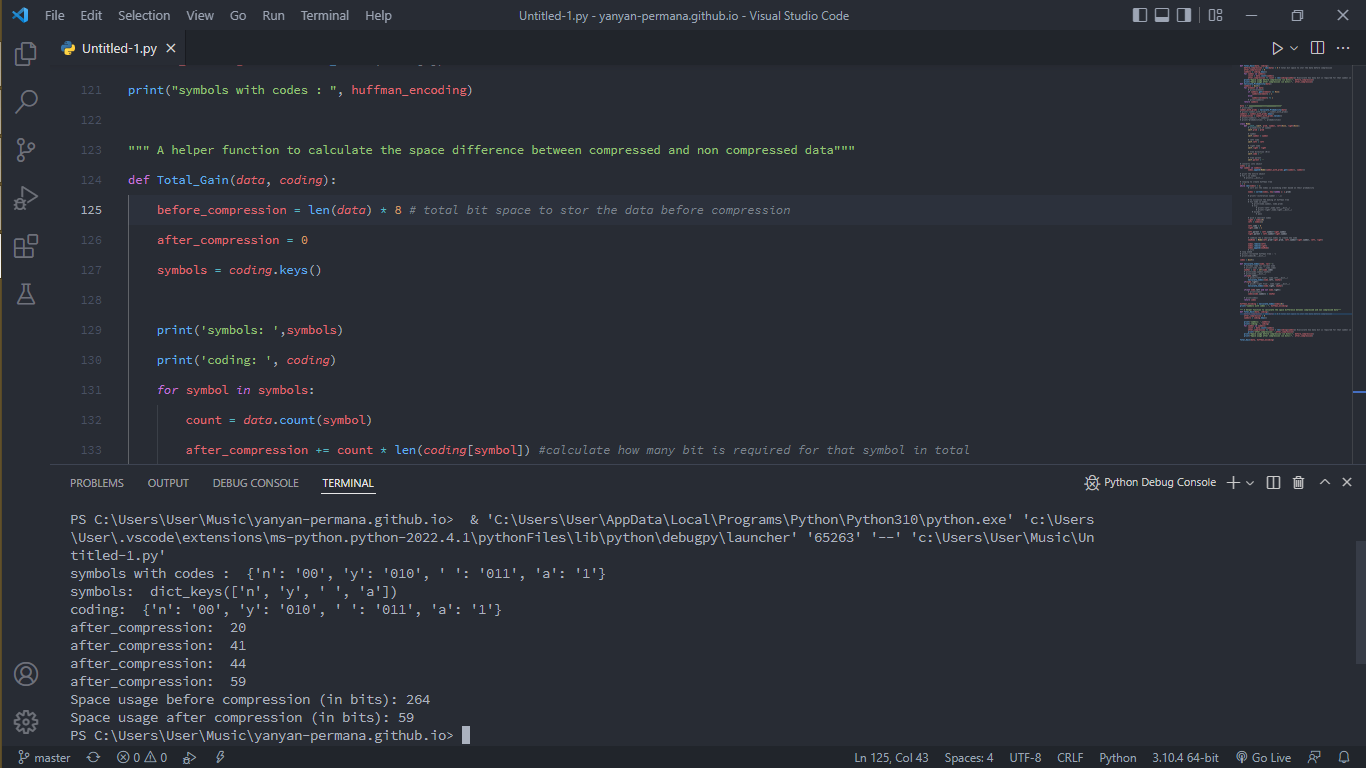
        after\_compression += count \* len(coding[symbol]) #calculate how many bit is required for that symbol in total

        print('after\_compression: ',after\_compression)

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

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

Total\_Gain(data, huffman\_encoding)



1. 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 = " yyyyyyyyaaaaanyyyaaan"

# 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 = ''

# 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\_\_)

codes = dict()

def Calculate\_Codes(node, val=''):

    # huffman code for current node

    # print('node code: ',node.code)

    newVal = val + str(node.code)

    # print(node.symbol,newVal)

    # print(node.\_\_dict\_\_)

    if(node.left):

        # print("left True", node.left.\_\_dict\_\_)

        Calculate\_Codes(node.left, newVal)

    if(node.right):

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

        Calculate\_Codes(node.right, newVal)

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

        # print(codes)

        codes[node.symbol] = newVal

    # print(codes)

    return codes

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

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

""" 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()

    # print('symbols: ',symbols)

    # print('coding: ', coding)

    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('after\_compression: ',after\_compression)

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

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

Total\_Gain(data, huffman\_encoding)

""" 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])

        print(c)

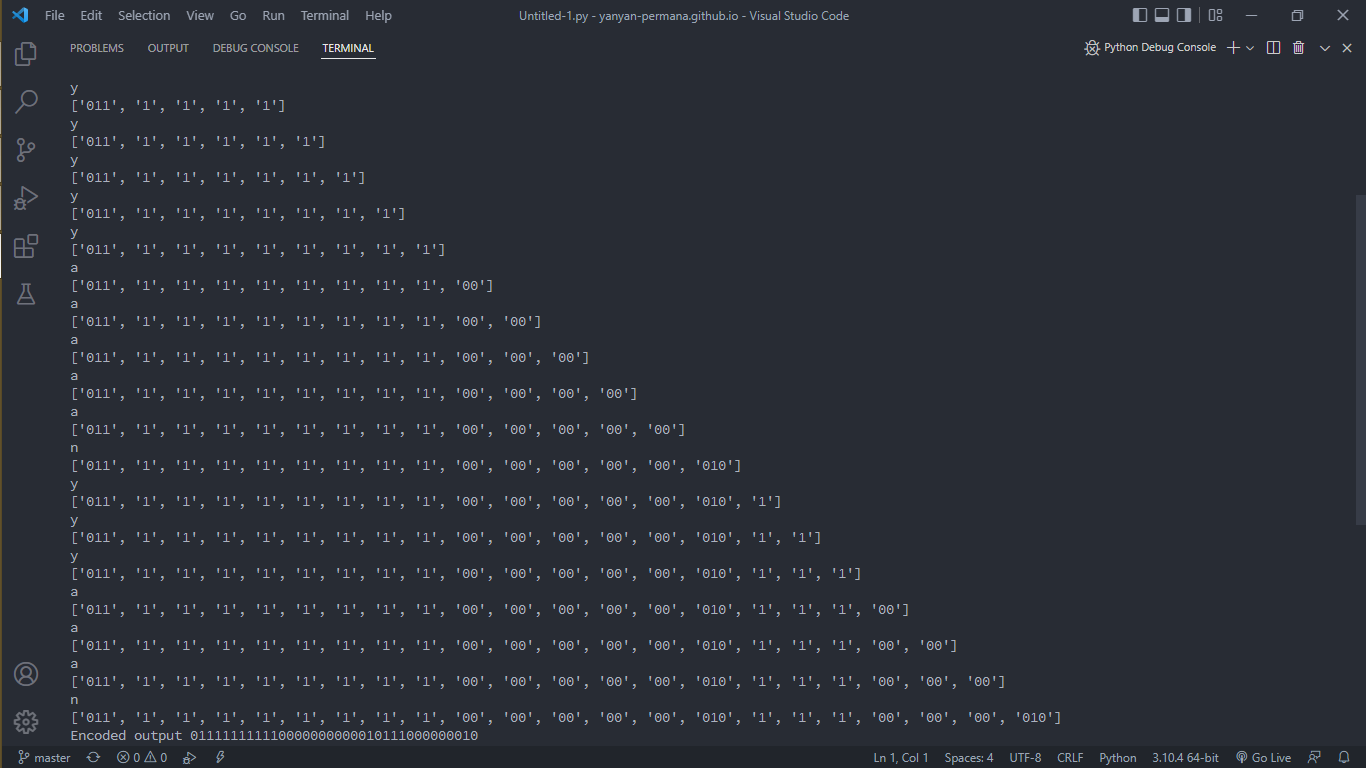
        print(encoding\_output)

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

    return string

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

print("Encoded output", encoded\_output)



1. 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

data = "yyyyyaaaaaaaaaaannnnnnnnn"

print(data)

encoding, tree = Huffman\_Encoding(data)

print("Encoded output", encoding)

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