**Assignment 3**

**Group:**BDA-1904

**Student Name:**Berik Gulina &Abdikalyk Gulnazym

**Github Link:** <https://github.com/gulina0426/InformationTheory>

**1-Description:**

Based on this assignment,we have totally 3 task to finish,in this way we have down the second and third together.As we learnt before the encoding part of the Huffman tree and in this task we are going to implement it in the way of inverse that decoding part to traverse our tree.

**2-Team working process:**

As usual we have done the whole work in the colab and teams to see each others working process and solving each problems together.We have divided equally like the first task was already implemented before in the previous assignment and the other parts we have done together.

**3-Execution part and screenshort:**

**3.1 Task-1:**

****

Figure-1 Result of the first task

**Source code-1:**

#ASSIGNMENT4 FIRST TASK

import os

import heapq

from heapq import heappop, heappush

#Firstly set our file in order to be readable

The\_text = open('Nn.txt','r',encoding='utf-8').read()

def Check\_isLeaf\_g(root):#Set our function to check is the leaf or not

    return root.gleft is None and root.gright is None#then to return none or not

    # here start the tree node

class Node:#Then is our class about the Node

    def \_\_init\_\_(self, gch, gfreq, gleft=None, gright=None):

        self.gch = gch#Here we need the para with children,frequency,left and right

        self.gfreq = gfreq#set each of them related to their

        self.gleft = gleft

        self.gright = gright

    # Here we need to override the whole function to make our node class work with priority in queue

    # such in here that the highest priority item who has the lowest frequency

    def \_\_lt\_\_(self, other):#Then in here we need to define this one

        return self.gfreq < other.gfreq #Then to return the part

# Here need to traverse the our Tree and store it in the dictionary

def encode(root, str, huffman\_code):

    if root is None:#If none then return

        return

    # In here need to found a leaf node as well

    if Check\_isLeaf\_g(root):#Check the related condtion as follow

        huffman\_code[root.gch] = str if len(str) > 0 else '1'

    encode(root.gleft, str + '0', huffman\_code)#our left child

    encode(root.gright, str + '1', huffman\_code)#our right child

# decoding part with our tree

def decode(root, index, str): #define it

        if root is None:#if none then return index

            return index

        # Here need to found a leaf node

        if Check\_isLeaf\_g(root):

            print( end='')

            return index

        index = index + 1#then the index will be added

        root = root.gleft if str[index] == '0' else root.gright

        return decode(root, index, str) #then return the decoding part

    # here need to build the tree

def buildHuffmanTree(The\_text):#The text is our file which will uploaded

    # if the empty string with condition

    if len(The\_text) == 0:

        return

    # here we need to firstly count the freq with each char,then to store it

    gfreq = {i: The\_text.count(i) for i in set(The\_text)}#counting the freq

    # here need to store the live nodes with creating the queue

    pq = [Node(k, v) for k, v in gfreq.items()]

    heapq.heapify(pq)

    while len(pq) != 1:

        # Remove the two nodes of the highest priority

        gleft = heappop(pq)

        gright = heappop(pq)

      #here we need to create the new one

        total = gleft.gfreq + gright.gfreq

        heappush(pq, Node(None, total, gleft, gright))

    # the root in here will stores the pointter of the tree

    root = pq[0]

    # traverse our\_tree and to store it in the dicc

    huffmanCode = {}

    encode(root, "", huffmanCode)

    # print the resultt

    str = ""

    for c in The\_text:

        str += huffmanCode.get(c)

    print(str)#saving it to the file

    text\_file\_save = open("outputtxt.txt", "w")

    nnn = text\_file\_save.write(str)

    text\_file\_save.close()

    if Check\_isLeaf\_g(root):

        # here is the special carses

        while root.gfreq > 0:

            print( end='')

            root.gfreq = root.gfreq - 1

    else:

        #decode and encode the string part here

        index = -1

        while index < len(str) - 1:

            index = decode(root, index, str)

# then to output the result

if \_\_name\_\_ == '\_\_main\_\_':

buildHuffmanTree(The\_text)

**3.2 Task-2&3:**

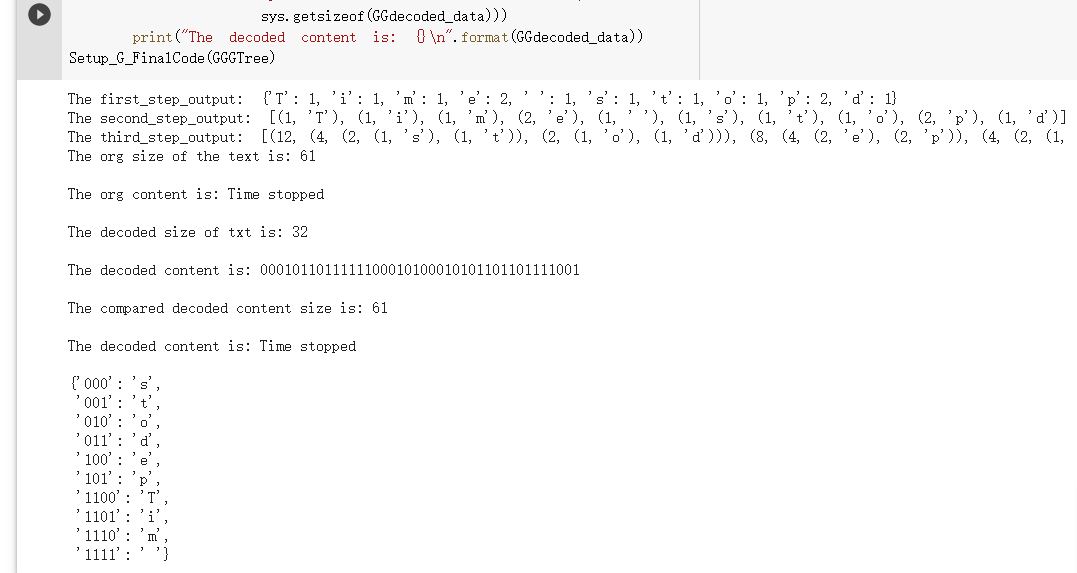


Figure-2 Result of the second and third task

**Source code-2&3:**

#Assignment 4 task 2 and 3 combination Gulina and Gulnazym

import heapq

import sys #HereWe need to import all the things what we need first

#Then we need to create our classes like nde class,tree and etc

#Then we need to create our classes like nde class,tree and etc

def AS4\_GG\_CountFrequency(string):

    G\_Total\_Frequ = dict.fromkeys(string, 0)

    for i in string:

        G\_Total\_Frequ[i] += 1

    print('The first\_step\_output: ', G\_Total\_Frequ)

    return G\_Total\_Frequ

#all the value and keys should be saved in the dictionary firstly

def AS4\_GG\_FindNodes(Saved\_Dic):

    G\_WholeNode = []

    G\_KeyListed = list(Saved\_Dic.values())

    G\_ValueListed = list(Saved\_Dic.keys())

    for i in range(len(G\_KeyListed)):

        G\_WholeNode.append((G\_KeyListed[i], G\_ValueListed[i]))

    print('The second\_step\_output: ', G\_WholeNode)

    return G\_WholeNode

def AS4\_GG\_OutTree(G\_SetNode):

#Here is our third function related to the tree list

    G\_WholeListTree = G\_SetNode

#In the tree we have its node with left and right children

    for i in range(len(G\_WholeListTree)-1):

#then the loopp

        G\_WholeListTree.sort(key=lambda tup: tup[0])

#After that we have sorted it in the desc order

#need to defining our frequency with total

        Left\_G\_Total\_Frequ = G\_WholeListTree[0][0]

        Right\_G\_Total\_Frequ = G\_WholeListTree[1][0]

        Total\_GG = Left\_G\_Total\_Frequ + Right\_G\_Total\_Frequ

#the total nodes will be the comb by left and right

        Left\_GT = G\_WholeListTree[0]

        Right\_GT = G\_WholeListTree[1]

#Here we gonna added the new node and deleted the old onee

        GTheNode = (Total\_GG, Left\_GT, Right\_GT)

        G\_WholeListTree.remove(Left\_GT)

        G\_WholeListTree.remove(Right\_GT)

        G\_WholeListTree.append(GTheNode)

#Whole part will be displayed by the output

    print('The third\_step\_output: ', G\_WholeListTree)

    return G\_WholeListTree

#Then the next function here which related to the finding list

def AS4\_GG\_ExistList(Created\_List, Listitem):

    try:

        return any(Listitem in sublist for sublist in Created\_List)#we have the sublist also

    except:

        return False

#fifth func

def AS4\_GG\_WholeCodes(Huff\_GG\_Tree, tuples):

#Which is related to the tree&tupples

    Huff\_GG\_Tree = Huff\_GG\_Tree[0]

#loop for checking it

    for i in tuples:

        CheckingProcee = True

        while CheckingProcee is True:

            if Find\_exist(Huff\_GG\_Tree[1], i) is True:

                print(i, 'is in 1')

                GTheNode11 = True

            else:

                GTheNode11 = False

            if Find\_exist(Huff\_GG\_Tree[2], i) is True:

                print(i, 'is in 2')

                GTheNode22 = True

            else:

                GTheNode22 = False

#considering the diff condition

            Huff\_GG\_Tree = Huff\_GG\_Tree[2]

            if GTheNode11 is False and GTheNode22 is False:

                CheckingProcee = False

    return 'My\_GG\_File'

#Then going to print the third result our tree

My\_GG\_File = open('Nn.txt','r',encoding='utf-8').read()

lastfreq = AS4\_GG\_CountFrequency(text)

lastNode = AS4\_GG\_FindNodes(lastfreq)

GGGTree = AS4\_GG\_OutTree(lastNode)

def Setup\_G\_FinalCode(Huff\_GG\_Tree):

    def myfuncc(GTheNode, GWord):

        if len(GTheNode) == 2:

            \_, symboll = GTheNode

            yield (GWord, symboll)

        else:

            \_, leftChildG, rightChildG = GTheNode

            yield from myfuncc(leftChildG, GWord + '0')

            yield from myfuncc(rightChildG, GWord + '1')

    SetRoot = Huff\_GG\_Tree[0]

    # convert (GWord, symboll) pairs to dictionary

    return dict(myfuncc(SetRoot, ''))

Setup\_G\_FinalCode(GGGTree)

class GG\_SetNode(object):

#Here we need to define our class that which is related to the set node

    def \_\_init\_\_(G\_Initial, Gvalue=None, Gcharacter=None):

        G\_Initial.Gcharacter = Gcharacter

        G\_Initial.Gvalue = Gvalue

        G\_Initial.GGleft = None

        G\_Initial.GGright = None

#we have each parameters with the left,right children

    def setGcharacter(G\_Initial, Gcharacter):

        G\_Initial.Gcharacter = Gcharacter

#AFTER that we need to set and get the value

    def setGvalue(G\_Initial, Gvalue):

        G\_Initial.Gvalue = Gvalue

#set is going to set down the initial value for them

    def setGGleftchild(G\_Initial, TheNode):

        G\_Initial.GGleft = TheNode

#get is going to get this from them

    def setGGrightchild(G\_Initial, TheNode):

        G\_Initial.GGright = TheNode

    def getGvalue(G\_Initial):

        return G\_Initial.Gvalue

    def getGcharacter(G\_Initial):

        return G\_Initial.Gcharacter

    def getGGrightchild(G\_Initial):

        return G\_Initial.GGright

    def getGGleftchild(G\_Initial):

        return G\_Initial.GGleft

#then we also have the has function which gonna check like the left and right child

    def hasGGleftchild(G\_Initial):

        return G\_Initial.GGleft is not None

    def hasGGrightchild(G\_Initial):

        return G\_Initial.GGright is not None

    def \_\_gt\_\_(G\_Initial, other):

        return G\_Initial.Gvalue > other.Gvalue

#here we goingto comapre the fun

# function for encoding a string

def G\_Encoding\_Tree(contentt):

#then we need to ctrea our dic to saving them

    def count\_letter(stringgs):

        G\_Count\_Word = dict()

        for i in stringgs:

            G\_Count\_Word[i] = G\_Count\_Word.get(i, 0) + 1

        G\_Count\_Word = dict(sorted(G\_Count\_Word.items(), key=lambda x: x[1]))

        return G\_Count\_Word

#Then return the count

#We also need to create the related nodes to save the frequency

    def Gheap(G\_Count\_Word):

        Gheap = list()

        for key in G\_Count\_Word:

            TheNode = GG\_SetNode(G\_Count\_Word[key], key)

            heapq.heappush(Gheap, TheNode)

        heapq.heapify(Gheap)

        return Gheap

#Then we need to merge two of them into one

    def g\_hufff\_tree(Gheap):

        GGleft\_node = GG\_SetNode

        GGright\_node = GG\_SetNode

        nodesss = list()

        while len(Gheap) > 1:#considering the situation in here

            GGleft\_node = heapq.heappop(Gheap)

            GGright\_node = heapq.heappop(Gheap)

            G\_new\_node = GG\_SetNode(#set our new node

                GGleft\_node.getGvalue() +

                GGright\_node.getGvalue(),

                GGleft\_node.getGcharacter() +

                GGright\_node.getGcharacter())

            G\_new\_node.setGGleftchild(GGleft\_node)

            G\_new\_node.setGGrightchild(GGright\_node)

            heapq.heappush(Gheap, G\_new\_node)

            nodesss.append(G\_new\_node)

        return nodesss

#Based on the logical that it will firsly visisitng the left child then the right child

    def Huff\_Traversing(root, GGcode, char\_freq, result):

        if root.hasGGleftchild() == False and root.hasGGrightchild() == False:

            char\_freq[root.getGcharacter()] = GGcode

#In here is our tree traversing which have the parameter like char

            result[root.getGcharacter()] = GGcode

            return None

        else:

#another side the the leaves stored in GGcode

            Huff\_Traversing(root.getGGleftchild(), GGcode + '0', char\_freq, result)

            Huff\_Traversing(root.getGGrightchild(), GGcode + '1', char\_freq, result)

        return result

#Then is the result and the new line

    def Gencode(code\_dict, stringgs):

        Gencode = str()

#after that the loop

        for i in range(len(stringgs)):

            Gencode += code\_dict[stringgs[i]]

#next

        return Gencode

#here we have our parameters

    nodesss = list()

    new\_stringgs = contentt

    GGcode = str()

    result = dict()

    G\_Count\_Word = dict()

    G\_Count\_Word = count\_letter(new\_stringgs)

    nodesss = g\_hufff\_tree(Gheap(G\_Count\_Word))

    return Gencode(Huff\_Traversing(nodesss[-1], GGcode,

                           G\_Count\_Word, result), new\_stringgs), nodesss[-1]

#in the decoding it gonna return the tree which we built in the previous

#decodingPart

def G\_Decoding\_Tree(GGcode, tree):

    def decoding(GGcode, tree, decode):

        root = GG\_SetNode

        root = tree

        for i in range(len(GGcode)):

            if GGcode[i] == '0':

                tree = tree.getGGleftchild()

                if tree.hasGGleftchild() == False and tree.hasGGleftchild() == False:

                    decode += tree.getGcharacter()

                    tree = root

            elif GGcode[i] == '1':

                tree = tree.getGGrightchild()

                if tree.hasGGleftchild() == False and tree.hasGGleftchild() == False:

                    decode += tree.getGcharacter()

                    tree = root

        return decode

    clear = decoding(GGcode, tree, decode=str())

    return clear

#Final to rpint the result

if \_\_name\_\_ == "\_\_main\_\_":

    codes = {}

#SET correctly pr

    my\_texxt = open('Nn.txt','r',encoding='utf-8').read()

    print("The org size of the text is: {}\n".format(

        sys.getsizeof(my\_texxt)))

    print("The org content is: {}\n".format(my\_texxt))

    GGencoded\_data, tree = G\_Encoding\_Tree(my\_texxt)

    print("The decoded size of txt is: {}\n".format(

        sys.getsizeof(int(GGencoded\_data, base=2))))

    print("The decoded content is: {}\n".format(GGencoded\_data))

    GGdecoded\_data = G\_Decoding\_Tree(GGencoded\_data, tree)

    print(

        "The compared decoded content size is: {}\n".format(

            sys.getsizeof(GGdecoded\_data)))

    print("The decoded content is: {}\n".format(GGdecoded\_data))

Setup\_G\_FinalCode(GGGTree)