Arab Academy for Science, Technology and maritime transport

College: CCIT

Department: CS

Course: Data Compression

Data-compression Project

Lecture: Dr. Saleh Mesbah

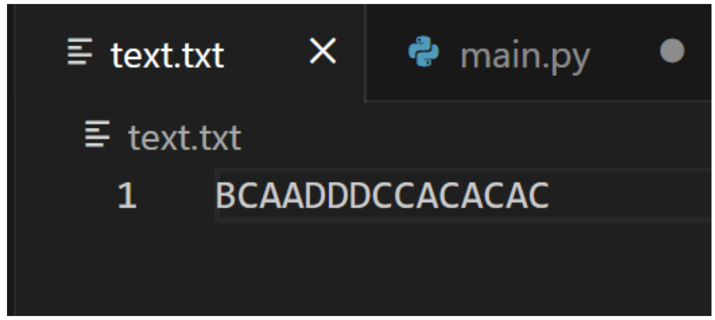
TA: Eng. Mahmoud El Morshedy

Name: Samer Ahmed Ibrahim

The programming language is python.

Introduction for Huffman for my project:

to start the Huffman coding we start with the text that will put every character in an array



so we read the text.txt file and to put it in string and

```python
text_file = open("text.txt", "r")
string = text_file.read()
text_file.close()
```

then we need to calculate the frequency so we made a dic named frequ and we made and its complexity is O(N) so when we print it the output will be like this

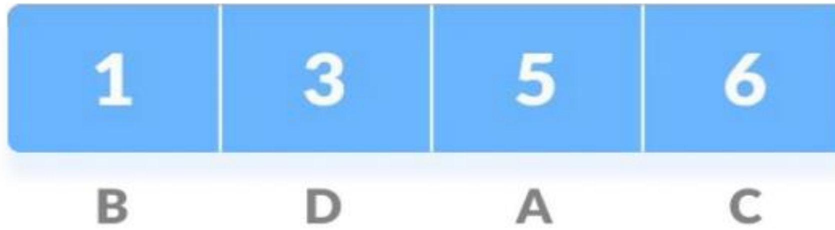
```python
frequ = {}
for x in string:
    if x in frequ:
        frequ[x] += 1
    else:
        frequ[x] = 1
```


```
{'B': 1, 'C': 6, 'A': 5, 'D': 3}
```

then we need to sort them to start making the tree so we used this

```python
frequ = sorted(frequ.items(), key=lambda x: x[1], reverse=False)
```

it will sort in and will make every char with its freq in bracket and will be printed like this




```
[('B', 1), ('D', 3), ('A', 5), ('C', 6)]
```

to start the Huffman we make a class named Node to help us do the tree

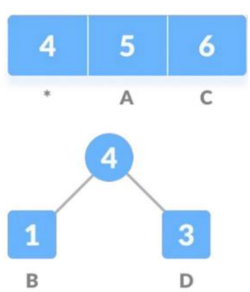
freq will represent the frequency of each character and char for character.

```python
class Node:
    def __init__(self, freq, char=None):
        self.freq = freq
        self.char = char
        self.left = None
        self.right = None
```

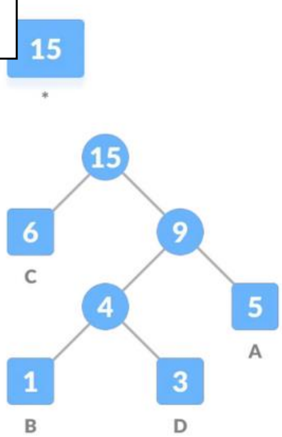
we did function named create_tree for the HUFFMAN and to know the codes of the character

Pic(1)

Pic(2)



Pic(3)



```python
def creat_tree(freq):
    nodes = [Node(freq=f[1], char=f[0]) for f in freq]
    while len(nodes) > 1:
        nodes.sort(key=lambda x: x.freq)
        new_node = Node(freq=nodes[0].freq + nodes[1].freq)
        new_node.left = nodes.pop(0)
        new_node.right = nodes.pop(0)
        nodes.append(new_node)
    return nodes[0]
```

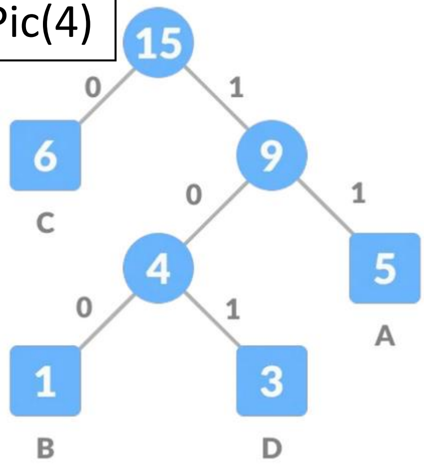
Create a leaf node for each character and assign its frequency as its weight by `nodes = [Node(freq=f[1], char=f[0]) for f in freq]` **.**

And then in the while loop that its complexity is O(1), we will Build the Huffman tree by repeatedly combining the two nodes with the lowest weights, then Sort the nodes by weight by `nodes.sort(key=lambda x: x.freq)`**, then we Create a new internal node with a weight equal to the sum of the two lowest-weight nodes like in pic(2) by**

```python
new_node = Node(freq=nodes[0].freq + nodes[1].freq)
new_node.left = nodes.pop(0)
new_node.right = nodes.pop(0)
```

and we Add the new node to the list of nodes by `nodes.append(new_node)`**, Then we Return the root node of the Huffman tree**

Pic(4)




```python
def getcode(node, code='', code_dict=None):
    if code_dict is None:
        code_dict = {}
    node.code = code
    if node.char:
        code_dict[node.char] = node.code

    else:
        getcode(node.left, code + '0', code_dict)
        getcode(node.right, code + '1', code_dict)


    return code_dict
```

to continue the Huffman coding the left will be bit 0 and the right will be 1 like in pic(4)

we defined a Function to assign bit patterns to each character in the Huffman tree , and then If no code dictionary is provided, create a new dictionary in the first if condition and then we Assign the bit pattern to the current node `node.code = code`.

Then a if condition for If the node represents a character, add it and its bit pattern to the dictionary, Otherwise, recursively assign bit patterns to the left(0) and right(1) child nodes.

`getcode(node.left, code + '0', code_dict)`

`getcode(node.right, code + '1', code_dict)`

Then we Return the dictionary

Create the Huffman tree and assign bit patterns to each character

```
root = creat_tree(frequ)
code_dict = getcode(root)
print(code_dict)
```

We requested the frequ to create the Huffman tree and put it in root , then we get every code for each character by function getcode() we did and put it in code_dict and we print it and the output will be

```
{'C': '0', 'B': '100', 'D': '101', 'A': '11'}
```

Then we encode to make the bits of the text

```
encoded_text = ''.join([code_dict[char] for char in string])
print("Encoded text :",encoded_text)
```

And then we make Function to decode a Huffman code using the character codes dictionary

```
def huffman_decompress(binary_file_path, code_dict, output_text_file_path):
    with open(binary_file_path, 'rb') as file:
        encoded_bytes = file.read()

    encoded_bits = ''.join(format(byte, '08b') for byte in encoded_bytes)
    decoded_text = ''
    current_code = ''

    for bit in encoded_bits:
        current_code += bit
        if current_code in code_dict.values():
            decoded_text += list(code_dict.keys())[list(code_dict.values()).index(current_code)]
            current_code = ''

    with open(output_text_file_path, 'w') as output_file:
        output_file.write(decoded_text)
```

For loop to Iterate over each bit in the Huffman code and then Add the current bit to the current code `current_code += bit`. Then Check if the current bit sequence is a valid Huffman code If it is, add the corresponding character to the decoded text, then Reset the current code to an empty string, then we Decode the Huffman code back to text

```
decoded_text = dec(encoded_text, code_dict)
print("Decode:",decoded_text)
```

THE FINAL OUTPUT

```
{'B': 1, 'C': 6, 'A': 5, 'D': 3}
[('B', 1), ('D', 3), ('A', 5), ('C', 6)]
{'C': '0', 'B': '100', 'D': '101', 'A': '11'}
Encoded text : 10001111101101101001101101l0
Decode: BCAADDDCCACACAC
```

TEXT.TXT:

BCAADDDCCACACAC

THE CODE:

```
class Node:
    def __init__(self, freq, char=None):
        self.freq = freq
        self.char = char
        self.left = None
        self.right = None

def creat_tree(freq):
    nodes = [Node(freq=f[1], char=f[0]) for f in freq]

    while len(nodes) > 1:
        nodes.sort(key=lambda x: x.freq)
        new_node = Node(freq=nodes[0].freq + nodes[1].freq)
        new_node.left = nodes.pop(0)
        new_node.right = nodes.pop(0)

        nodes.append(new_node)
    return nodes[0]

def getcode(node, code='', code_dict=None):
    if code_dict is None:
        code_dict = {}


    node.code = code

    if node.char:
        code_dict[node.char] = node.code

    else:
        getcode(node.left, code + '0', code_dict)
        getcode(node.right, code + '1', code_dict)

    return code_dict

def huffman_decompress(binary_file_path, code_dict, output_text_file_path):
    with open(binary_file_path, 'rb') as file:
        encoded_bytes = file.read()

    encoded_bits = ''.join(format(byte, '08b') for byte in encoded_bytes)
    decoded_text = ''
    current_code = ''

    for bit in encoded_bits:
        current_code += bit
        if current_code in code_dict.values():
```

```python
            decoded_text +=
list(code_dict.keys())[list(code_dict.values()).index(current_code)]
            current_code = ''

    with open(output_text_file_path, 'w') as output_file:
        output_file.write(decoded_text)


text_file = open("text.txt","r")
string = text_file.read()
text_file.close()

frequ = {}
for x in string:
    if x in frequ:
        frequ[x] += 1
    else:
        frequ[x] = 1


print(frequ)
frequ = sorted(frequ.items(), key=lambda x: x[1], reverse=False)
print(frequ)

root = creat_tree(frequ)
code_dict = getcode(root)
print(code_dict)

encoded_text = ''.join([code_dict[char] for char in string])
print("Encoded text :",encoded_text)

decoded_text = dec(encoded_text, code_dict)
print("Decode:",decoded_text)
```

Now in the project we will take 3 text file named (1260,1497,30360-8)
So we will add the input

```python
text_file1 = open("1260.txt","r")
string1 = text_file1.read()
text_file1.close()
####################################
text_file2 = open("1497.txt","r")
string2 = text_file2.read()
text_file2.close()
#############################
text_file3 = open("30360-8.txt","r")
string3= text_file3.read()
text_file3.close()
###############################
```

And then we will calculate the frequency and all of that like I said in the previous pages all the things we will do just add the other files and the steps are the same and in the pdf we need to

```python
# Open a text file in write mode ('w')
with open('file1,binary.txt', 'w') as file:
    # Write data to the file
    file.write(encoded_text1)
with open('file2,binary.txt', 'w') as file:
    # Write data to the file
    file.write(encoded_text2)

with open('file3,binary.txt', 'w') as file:
    # Write data to the file
    file.write(encoded_text3)
```

output a text files in it we will put the encoded text (its binary)

We will do this

Then we will calculate the size of the files in bits and the compression ratio That equal(before compression/after compression)

So we import os and then we write

```
file_size1 = (os.path.getsize('1260.txt'))*8
file_size2 = (os.path.getsize('1497.txt'))*8
file_size3 = (os.path.getsize('30360-8.txt'))*8
compr1=round(file_size1/len(encoded_text1),4)
compr2=round(file_size2/len(encoded_text2),4)
compr3=round(file_size3/len(encoded_text3),4)
```

And we need to make the output like this

	File 1	File 2	File 3
Huffman	Compression Ratio	Compression Ratio	Compression Ratio

In code its like this

```
##########################################################################
print("Huffman || file 1 || file 2 || file 3 ||")
print("        ||",compr1,"||",compr2,"||",compr3,"||")
print("-------------------------------------------------")
```

```
File original Size of file 1 is : 8562640 bits ||File Size of file 1 after compression is : 4750541 bits
File original Size of file 2 is : 9912872 bits ||File Size of file 2 after compression is : 4750541 bits
File original Size of file 3 is : 11074280 bits ||File Size of file 3 after compression is : 4750541 bits
Huffman || file 1 || file 2 || file 3 ||
        || 1.8025 || 1.8489 || 1.8029 ||
```

That's it for the Huffman part AND that's the code so far before we add lzw

The code

```
import os

#########################
# Define a class to represent nodes in the Huffman tree
class Node:
    def __init__(self, freq, char=None):
        self.freq = freq
        self.char = char
        self.left = None
        self.right = None


# Function to create the Huffman tree given a list of character frequencies
def creat_tree(freq):
    # Create a leaf node for each character and assign its frequency as its weight
    nodes = [Node(freq=f[1], char=f[0]) for f in freq]

    # Build the Huffman tree by repeatedly combining the two nodes with the lowest weights
    while len(nodes) > 1:
        # Sort the nodes by weight
        nodes.sort(key=lambda x: x.freq)

        # Create a new internal node with a weight equal to the sum of the two lowest-weight
nodes
        new_node = Node(freq=nodes[0].freq + nodes[1].freq)
        new_node.left = nodes.pop(0)
        new_node.right = nodes.pop(0)

        # Add the new node to the list of nodes
        nodes.append(new_node)

    # Return the root node of the Huffman tree
    return nodes[0]
```

```
# Function to encode bit patterns to each character in the Huffman tree
def getcode(node, code='', code_dict=None):
    # If no code dictionary is provided, create a new dictionary
    if code_dict is None:
        code_dict = {}

    # Assign the bit pattern to the current node
    node.code = code

    # If the node represents a character, add it and its bit pattern to the dictionary
    if node.char:
        code_dict[node.char] = node.code
    # Otherwise, recursively assign bit patterns to the left and right child nodes
    else:
        getcode(node.left, code + '0', code_dict)
        getcode(node.right, code + '1', code_dict)

    # Return the dictionary
    return code_dict

# Function to decode a Huffman code using the character codes dictionary
def huffman_decompress(binary_file_path, code_dict, output_text_file_path):
    with open(binary_file_path, 'rb') as file:
        encoded_bytes = file.read()

    encoded_bits = ''.join(format(byte, '08b') for byte in encoded_bytes)
    decoded_text = ''
    current_code = ''

    for bit in encoded_bits:
        current_code += bit
        if current_code in code_dict.values():
            decoded_text +=
list(code_dict.keys())[list(code_dict.values()).index(current_code)]
            current_code = ''

    with open(output_text_file_path, 'w') as output_file:
        output_file.write(decoded_text)

# Read the input text file and count the frequency of each character
text_file1 = open("1260.txt","r")
string1 = text_file1.read()
text_file1.close()
###################################
text_file2 = open("1497.txt","r")
string2 = text_file2.read()
text_file2.close()
############################
text_file3 = open("30360-8.txt","r")
string3= text_file3.read()
```

```
text_file3.close()
##############################
frequ1 = {}
for x in string1:
    if x in frequ1:
        frequ1[x] += 1
    else:
        frequ1[x] = 1
###############################
frequ2 = {}
for x in string2:
    if x in frequ2:
        frequ2[x] += 1
    else:
        frequ2[x] = 1
####################################
frequ3 = {}
for x in string3:
    if x in frequ3:
        frequ3[x] += 1
    else:
        frequ3[x] = 1
####################################
# Sort the character frequencies in ascending order
#print(frequ1)
ffrequ1 = sorted(frequ1.items(), key=lambda x: x[1], reverse=False)
ffrequ2 = sorted(frequ2.items(), key=lambda x: x[1], reverse=False)
ffrequ3 = sorted(frequ3.items(), key=lambda x: x[1], reverse=False)
#print(frequ1)
# Create the Huffman tree and assign bit patterns to each character
root1 = creat_tree(ffrequ1)
code_dict1 = getcode(root1)
##print(code_dict)
###################################
root2 = creat_tree(ffrequ2)
code_dict2 = getcode(root2)
#####################################
root3 = creat_tree(ffrequ3)
code_dict3 = getcode(root3)
##############################################
# Encode the input text using the Huffman code
encoded_text1 = ''.join([code_dict1[char] for char in string1])
encoded_text2 = ''.join([code_dict2[char] for char in string2])
encoded_text3 = ''.join([code_dict3[char] for char in string3])

# Convert the binary-encoded text into bytes
encoded_bytes1 = bytes(int(encoded_text1[i:i+8], 2) for i in range(0, len(encoded_text1), 8))
encoded_bytes2 = bytes(int(encoded_text2[i:i+8], 2) for i in range(0, len(encoded_text2), 8))
encoded_bytes3 = bytes(int(encoded_text3[i:i+8], 2) for i in range(0, len(encoded_text3), 8))
```

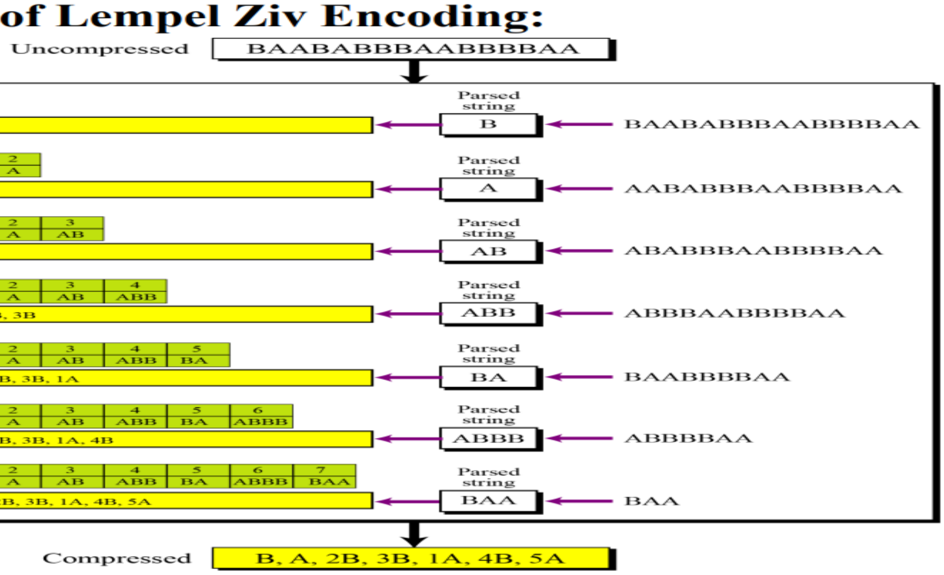
```
# Write bytes to binary files
with open('file1_binary.bin', 'wb') as file:
    file.write(encoded_bytes1)

with open('file2_binary.bin', 'wb') as file:
    file.write(encoded_bytes2)

with open('file3_binary.bin', 'wb') as file:
    file.write(encoded_bytes3)

#############################################################
#calculate size
file_size1 = (os.path.getsize('1260.txt'))*8
file_size2 = (os.path.getsize('1497.txt'))*8
file_size3 = (os.path.getsize('30360-8.txt'))*8
compr1=round(file_size1/len(encoded_text1),4)
compr2=round(file_size2/len(encoded_text2),4)
compr3=round(file_size3/len(encoded_text3),4)
print("Huffman")
print("File original Size of file 1 is :", file_size1, "bits","||File Size of file 1 after
compression is :",len(encoded_text1),"bits")
print("File original Size of file 2 is :", file_size2, "bits","||File Size of file 2 after
compression is :",len(encoded_text1),"bits")
print("File original Size of file 3 is :", file_size3, "bits","||File Size of file 3 after
compression is :",len(encoded_text1),"bits")
```

In lzw

```python
def compress(text):
    dictionary = {chr(i): i for i in range(256)}
    result = []
    p = text[0]

    for c in text[1:]:
        pc = p + c
        if pc in dictionary:
            p = pc
        else:
            result.append(dictionary[p])
            dictionary[pc] = len(dictionary)
            p = c

    result.append(dictionary[p])
    return result
```

This function for lzw compression we make a dictionary and then check the string with the next one if its in the dictionary or not if yes it will add it to the dictionary if not go for adding the next string to it example------

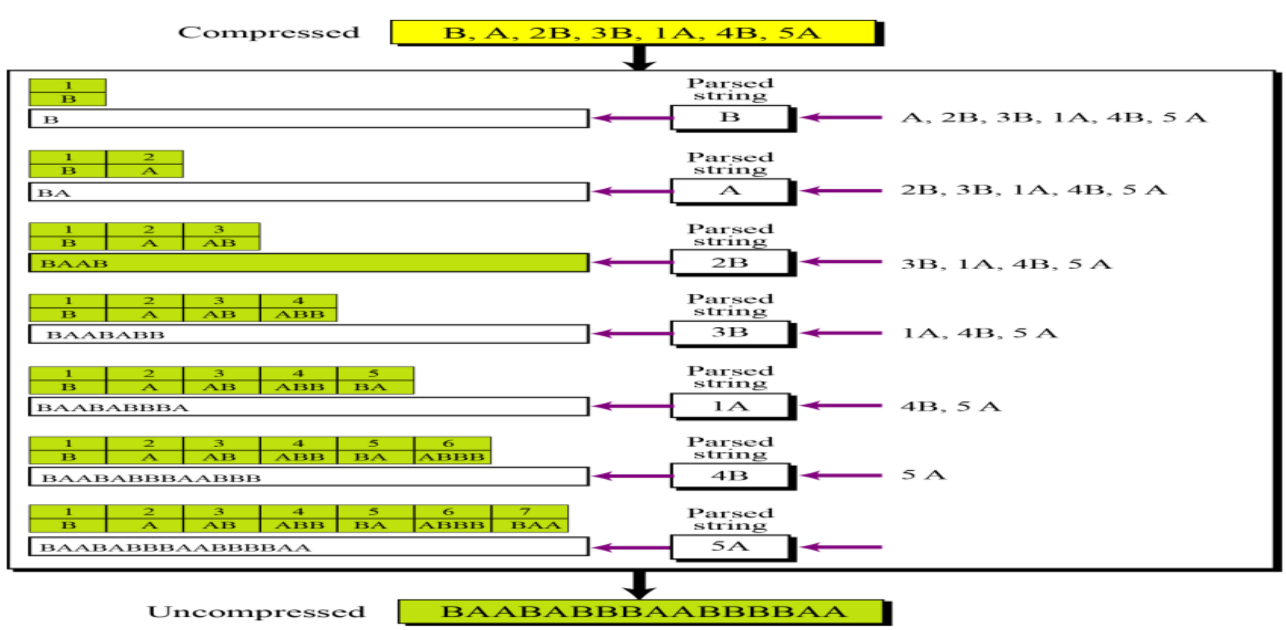
An example of Lempel Ziv Encoding:



```python
def decompress(compressed):
    dictionary = {i: chr(i) for i in range(256)}
    result = [dictionary[compressed[0]]]
    p = dictionary[compressed[0]]

    for code in compressed[1:]:
        if code in dictionary:
            entry = dictionary[code]
        elif code == len(dictionary):
            entry = p + p[0]
        else:
            raise ValueError("Invalid code: " + str(code))

        result.append(entry)
        dictionary[len(dictionary)] = p + entry[0]
        p = entry

    return ''.join(result)
```

The lzw decompression we will make the opposite of the decompression we will the the encoded for the strings if its in the dictionary we will replace with its pattern example for it -----------

An example of Lempel Ziv Decoding:



```python
# Test the LZW compression and decompression
file_name1 = "1260.txt"
with open(file_name1, "r") as file:
    text1 = file.read()
file_name2 = "1497.txt"
with open(file_name2, "r") as file:
    text2 = file.read()
file_name3 = "30360-8.txt"
with open(file_name3, "r") as file:
    text3 = file.read()

# Compress the text using LZW algorithm
compressed_result1 = compress(text1)
compressed_result2 = compress(text2)
compressed_result3 = compress(text3)

# Convert the compressed LZW data into bytes
# Convert the compressed LZW data into bytes (clamp values to the range 0 to 255)
compressed_bytes1 = bytes(min(max(code, 0), 255) for code in compressed_result1)
compressed_bytes2 = bytes(min(max(code, 0), 255) for code in compressed_result2)
compressed_bytes3 = bytes(min(max(code, 0), 255) for code in compressed_result3)
```

In code we will do the same as the Huffman in this part , we will read the text files and the compressed_result(n) Will request the compress function and and the the encoded like this

B, A, 2B, 3B, 1A, 4B, 5A

And we will convert the compressed lzw data into bytes to upload it to binary file as requested in the project pdf

```python
# Write compressed data to binary files
with open('file1_lzw.bin', 'wb') as file:
    file.write(compressed_bytes1)

with open('file2_lzw.bin', 'wb') as file:
    file.write(compressed_bytes2)

with open('file3_lzw.bin', 'wb') as file:
    file.write(compressed_bytes3)
```

```python
comprl1=round(file_size1/len(compressed_result1),4)
comprl2=round(file_size2/len(compressed_result2),4)
comprl3=round(file_size3/len(compressed_result3),4)
print("LZW")
print("File original Size of file 1 is :", file_size1, "bits","||File Size of file 1 after compression is :",len(compressed_result1),"bits")
print("File original Size of file 2 is :", file_size2, "bits","||File Size of file 2 after compression is :",len(compressed_result2),"bits")
print("File original Size of file 3 is :", file_size3, "bits","||File Size of file 3 after compression is :",len(compressed_result3),"bits")

print("Huffman || file 1 || file 2 || file 3 ||")
print("        ||",compr1,"||",compr2,"||",compr3,"||")
print("---------------------------------------------")
print("LZW     |",comprl1,"|",comprl2,"|",comprl3,"||")
print("---------------------------------------------")
```

Here we will calculate the compression ratio and compare it with the Huffman compression ratio like requested in the pdf

```
Huffman
File original Size of file 1 is : 8562640 bits ||File Size of file 1 after compression is : 4750541 bits
File original Size of file 2 is : 9912872 bits ||File Size of file 2 after compression is : 4750541 bits
File original Size of file 3 is : 11074280 bits ||File Size of file 3 after compression is : 4750541 bits
LZW
File original Size of file 1 is : 8562640 bits ||File Size of file 1 after compression is : 201283 bits
File original Size of file 2 is : 9912872 bits ||File Size of file 2 after compression is : 210474 bits
File original Size of file 3 is : 11074280 bits ||File Size of file 3 after compression is : 233965 bits
Huffman || file 1 || file 2 || file 3 ||
        || 1.8025 || 1.8489 || 1.8029 ||
-----------------------------------------------
LZW     | 42.5403 | 47.0978 | 47.3331 ||
-----------------------------------------------
```

This is the comparison between the compression ratio of lzw and Huffman

And the after compression it's the same as the binary file of them so its correct and after compression it returned as the original files

1	file1_binary.bin	8/23/2023 3:12 AM	BIN File	580 KB
	file1_lzw.bin	8/23/2023 3:12 AM	BIN File	197 KB
2	file2_binary.bin	8/23/2023 3:12 AM	BIN File	655 KB
	file2_lzw.bin	8/23/2023 3:12 AM	BIN File	206 KB
3	file3_binary.bin	8/23/2023 3:12 AM	BIN File	750 KB
	file3_lzw.bin	8/23/2023 3:12 AM	BIN File	229 KB
	compress.py	8/20/2023 7:09 PM	Python Source File	10 KB
4	1260.txt	8/10/2023 4:22 AM	Text Document	1,046 KB
	1497.txt	8/10/2023 4:22 AM	Text Document	1,211 KB
	30360-8.txt	8/10/2023 4:22 AM	Text Document	1,352 KB
5	file1_decompressed_huffman.txt	8/23/2023 3:12 AM	Text Document	1,046 KB
	file1_decompressed_lzw.txt	8/23/2023 3:13 AM	Text Document	1,046 KB
6	file2_decompressed_huffman.txt	8/23/2023 3:13 AM	Text Document	1,211 KB
	file2_decompressed_lzw.txt	8/23/2023 3:13 AM	Text Document	1,211 KB
7	file3_decompressed_huffman.txt	8/23/2023 3:13 AM	Text Document	1,352 KB
	file3_decompressed_lzw.txt	8/23/2023 3:13 AM	Text Document	1,352 KB

In 1 it's the compressed files using lzw and Huffman as 2 and 3 , as we see in the original size in 4 are the same of the decompressed files in 5,6 and 7

```python
# Decompress using Huffman decompression and save to a text file

huffman_decompress('file1_binary.bin', code_dict1,'file1_decompressed_huffman.txt')
huffman_decompress('file2_binary.bin', code_dict2,'file2_decompressed_huffman.txt')
huffman_decompress('file3_binary.bin', code_dict3,'file3_decompressed_huffman.txt')
##############################################################################################

output_texts = [decompress(compressed_result1),decompress(compressed_result2),decompress(compressed_result3)]
file_names = ['file1_decompressed_lzw.txt', "file2_decompressed_lzw.txt", "file3_decompressed_lzw.txt"]

for i, file_name in enumerate(file_names):
    with open(file_name, "w") as file:
        file.write(output_texts[i])
##############################################################################################
```

And those the decompression of the binary files , that's all

THANK YOU