Programming Assignment 3

1. Main Module cryptography.py:
    1. Has the main program
2. Steganography Module steganography.py:
    1. Has a class Steganography
3. Codec Module codec.py
    1. Has a class Codec
    2. Has a class CaesarCypher – subclass of Codec
    3. Has a class HuffmanCodes – subclass of Codec

Steganography

Steganography is the process of hiding a message within another object such as an image, audio file, movie, text, network streaming, etc. Unlike other cryptographic methods that conceals the contents of a secret message, steganography also conceals the fact that a message is communicated.

The word steganography is derived from the Greek words steganós, meaning "concealed" and grapha meaning "writing".

You will write a cryptography program that encode or decode cryptographic messages hidden in binary image files using steganography.

There are different techniques of using steganography; in this assignment, you will use the simple one, a least-significant-bit technique.

Image Files

- Original Array of an RGB-file

[[[0 0 255] [0 0 255]]

[[0 0 255] [0 0 255]]

[[0 0 255] [0 0 255]]]

- Binary Encoding

‘he’ in binary is ‘01101000 01100101’

- Steganoimage

[[[0 1 255] [0 1 254]]

[[0 0 254] [1 1 254]]

[[0 1 254] [1 0 255]]]

Module cryptography.py

```
# cryptography program
from steganography import Steganography

def main_menu():
    s = Steganography()
    menu = [ 'Encode a message       - E\n',
             'Decode a message       - D\n',
             'Print a message        - P\n',
             'Show an image          - S\n',
             'Quit the program       - Q\n']
    while True:
        print("\nChoose an operation:")
        for i in menu: print(i, end='')
        op = input().upper()
        if op == 'Q':
            break
        elif op == 'S' or op == 'E' or op == 'D':
            filein = input("Choose an image file:\n")
        if op == 'E':
            fileout = input("Choose an output image file:\n")
            s.encode(filein,fileout, get_message(), get_codec())
            s.print()
        elif op == 'D':
            s.decode(filein, get_codec())
            s.print()
        elif op == 'P':
            s.print()
        elif op == 'S':
            s.show(filein)
```

Module cryptography.py

```
def get_message():
    message = ''
    while True:
        message = input("Please type a message, use only ASCII characters:\n")
        try:
            for char in message: code = ord(char)
            if len(message) > 0: break
        except:
            print(f"The message contains not an ASCII character {char}!!!")
    return message
```

Module cryptography.py

```
def get_codec():
    while True:
        choice = input("\nChoose a codec
 method or return to the main menu:\n\
Steganography only                    - S\n\
Steganography & Caesar Cypher    - C\n\
Steganography & Huffman Codes   - H\n\
Return to the main menu          - Q\n").upper()

        if choice == 'Q':
            break
        elif choice == 'S':
            return 'binary'
        elif choice == 'C':
            return 'caesar'
        elif choice == 'H':
            return 'huffman'

if __name__ == '__main__':
    main_menu()
```

Module steganography.py

```
import cv2

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.image as mpimg

from math import ceil

from codec import Codec, CaesarCypher,
HuffmanCodes
```

```
class Steganography():
```

```
def __init__(self, delimiter = '#'):
    self.text = ''
    self.binary = ''
    self.delimiter = delimiter
    self.codec = None
```

Module steganography.py

```
def encode(self, filein, fileout, message, codec):
    image = cv2.imread(filein)
    print(image) # for debugging


    # calculate available bytes
    max_bytes = image.shape[0] * image.shape[1] * 3 // 8
    print("Maximum bytes available:", max_bytes)


    # convert into binary
    if codec == 'binary':
        self.codec = Codec()
    elif codec == 'caesar':
        self.codec = CaesarCypher()
    elif codec == 'huffman':
        self.codec = HuffmanCodes()
    binary = self.codec.encode(message + self.delimiter)


    # check if it is possible to encode the message
    num_bytes = ceil(len(binary)//8) + 1
    if  num_bytes > max_bytes:
        print("Error: Insufficient bytes!")
    else:
        print("Bytes to encode:", num_bytes)
        self.text = message
        self.binary = binary
        # your code goes here
        # you may create an additional method that modifies the image array
        cv2.imwrite(fileout, ?)
```

Module steganography.py

```
def decode(self, filein, codec):
    image = cv2.imread(filein)
    print(image) # for debugging


    # convert into text
    if codec == 'binary':
        self.codec = Codec()
    elif codec == 'caesar':
        self.codec = CaesarCypher()
    elif codec == 'huffman':
        if self.codec == None or self.codec.name != 'huffman':
            print("A Huffman tree is not set!")
            flag = False
        if flag:
            # your code goes here
            # you may create an additional method that extract bits from the image array
    binary_data = ?
    # update the data attributes:
    self.text = self.codec.decode(binary_data)
    self.binary = ?
```

Module codec.py: Binary Codec

```
# codecs
import numpy as np

class Codec():

    def __init__(self, delimiter='#'):
        self.name = 'binary'
        self.delimiter = delimiter

    # convert text or numbers into binary form
    def encode(self, text):
        if type(text) == str:
            return ''.join([format(ord(i), "08b") for i in text])
        else:
            print('Format error')

    # convert binary data into text
    def decode(self, data):
        binary = []
        for i in range(0,len(data),8):
            byte = data[i: i+8]
            if byte == self.encode(self.delimiter):
                break
            binary.append(byte)
        text = ''
        for byte in binary:
            text += chr(int(byte,2))
        return text
```

Module codec.py: Caesar Cypher

```
class CaesarCypher(Codec):

    def __init__(self, shift=3, delimiter='#'):
        self.name = 'caesar'
        self.delimiter = delimiter
        self.shift = shift
        self.chars = 256        # total number of characters

    # convert text into binary form
    # your code should be similar to the corresponding code used for Codec
    def encode(self, text):
        data = ''
        # your code goes here
        return data

    # convert binary data into text
    # your code should be similar to the corresponding code used for Codec
    def decode(self, data):
        text = ''
        # your code goes here
        return text
```

Huffman Codes

Huffman codes is a lossless data compression algorithm developed by David A. Huffman

If we use fixed code (not Huffman codes), then each character has length of 8 bits, and to encode a file 10^6 characters, we need to use $8^* 10^6$ bits. If we use Huffman codes, we need 10 times less memory

Compression is 20 – 90%

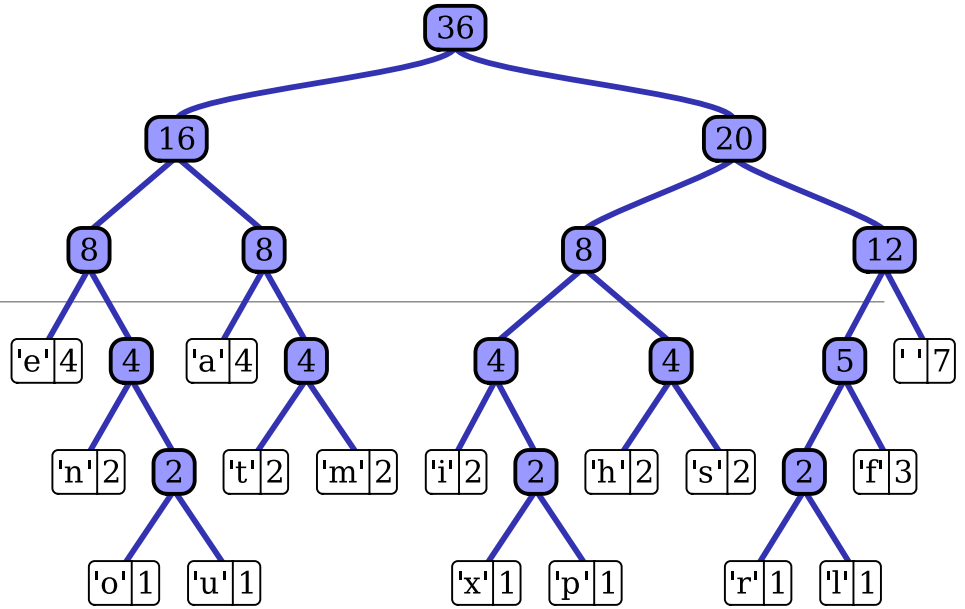
The algorithm is based on assigning a numeric binary code to a text character depending on the character frequency. Frequent characters have codes with small prefixes, rare characters – with long prefixes

The algorithm has two parts:

- Creating a Huffman tree
- Traversing the tree to find codes

Character	Frequency	Binary Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Huffman Codes: Example



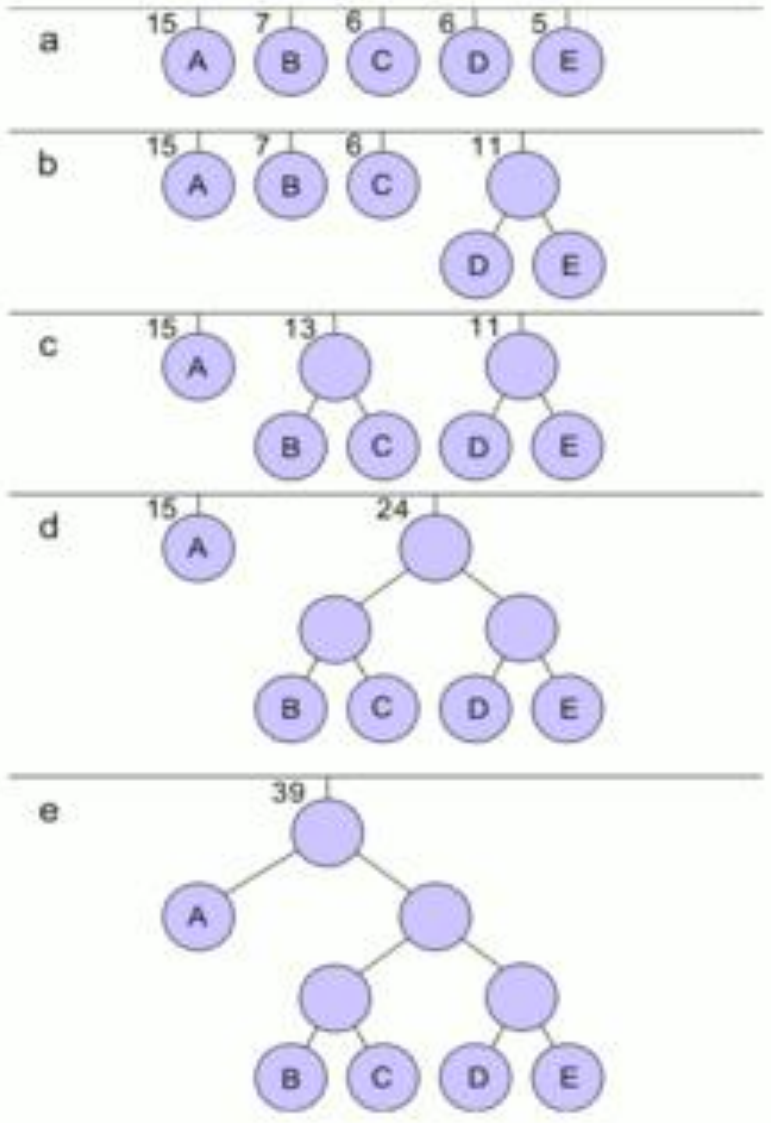
Text: “this is an example of a huffman tree”

Pseudocode:

- Calculate frequency of letters and arrange them in order: ‘ ‘ -7, e-4, a-4, f-3, t-2, etc.
- Make nodes and combine nodes with low frequencies into a binary tree where the root node has the frequency equals to the sum of frequencies of its child nodes
- Do the same process recursively until no nodes are left
- Traverse to each leaf node and find its code by combining prefixes. Each edge on the left has 0 and the right has 1

Complexity O(nlogn)

Character	Frequency	Binary Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111



Huffman Tree Construction

Module codec.py : Node

```
# a helper class used for class HuffmanCodes that implements a Huffman tree (Node)

class Node:

    def __init__(self, freq, symbol, left=None, right=None):

        self.left = left

        self.right = right

        self.freq = freq

        self.symbol = symbol

        self.code = ''
```

Module codec.py : Huffman Codes

```
class HuffmanCodes(Codec):
    def __init__(self):
        self.nodes = None
        self.name = 'huffman'
    # make a Huffman Tree
    def make_tree(self, data):
        # make nodes
        nodes = []
        for char, freq in data.items():
            nodes.append(Node(freq, char))
        # assemble the nodes into a tree
        while len(nodes) > 1:
            # sort the current nodes by frequency
            nodes = sorted(nodes, key=lambda x: x.freq)
            # pick two nodes with the lowest frequencies
            left = nodes[0]
            right = nodes[1]
            # assign codes
            left.code = '0'
            right.code = '1'
            # combine the nodes into a tree
            root = Node(left.freq+right.freq, left.symbol+right.symbol,
                        left, right)
            # remove the two nodes and add their parent to the list of nodes
            nodes.remove(left)
            nodes.remove(right)
            nodes.append(root)
        return nodes
```

Module codec.py : Huffman Codes

```
# traverse a Huffman tree
    def traverse_tree(self, node, val):
        next_val = val + node.code
        if(node.left):
            self.traverse_tree(node.left, next_val)
        if(node.right):
            self.traverse_tree(node.right, next_val)
        if(not node.left and not node.right):
            print(f"{node.symbol}->{next_val}") # this is
for debugging
```

```
# you need to update this part of the code
# or rearrange it so it suits your need
```

Module codec.py: Codec Driver

```
# driver program for codec classes
if __name__ == '__main__':
    text = 'hello'
    print('Original:', text)

    c = Codec()
    binary = c.encode(text + c.delimiter)
    print('Binary:',binary)
    data = c.decode(binary)
    print('Text:',data)
```

```
    cc = CaesarCypher()
    binary = cc.encode(text + cc.delimiter)
    print('Binary:',binary)
    data = cc.decode(binary)
    print('Text:',data)

    h = HuffmanCodes()
    binary = h.encode(text + h.delimiter)
    print('Binary:',binary)
    data = h.decode(binary)
    print('Text:',data)
```