PA2: Cryptography

Start Assignment

Due Tuesday by 11:59pm **Points** 75 **Submitting** a file upload **File Types** py

PA 2. Cryptography

Topics

1. Arrays and NumPy
2. Bytes and codecs
3. Text compressions and encodings: ASCII and Huffman Codes
4. Image compressions and encodings (lossy and lossless: jpeg and png)
5. Cryptography: Caesar Cypher and Steganography

Prerequisites:

Installation of Matplotlib including NumPy and openCV.

To install Matplotlib you can type:

```
pip install matplotlib
```

To install openCV you can type:

```
pip install opencv-python
```

Instructions

Cryptography

Objective: practicing with classes, ADT, interfaces (API), recursions, tree ADT, NumPy, Matplotlib, and openCV modules

Description

In this assignment, you will write a cryptography program that encode or decode cryptographic messages hidden in binary image files using 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". There are different techniques of using steganography; in this assignment, you will use the simple one, a least-significant-bit technique. **Least Significant Bit** is a technique in which the last bit of each pixel in the image file is modified and replaced with the secret message data bit. First, let's review how images are created on the computer.

Digital Images. A digital image is a two-dimensional array of **pixels**, where each pixel can be composed of three colors (red, green, blue in the RGB color scheme) if image is colored, or one gray color if image is white-and-black or has a gray scale. An image can be represented as a bitmap (gray scale or white-and-black images) or a colormap (colored images) that corresponds to a 2D or a 3D array of numbers respectively, where each number means brightness or color intensity of a pixel. For example, lets create a very simple image of four red-colored pixels arranges as a 2x2 array. Then, it should have the following 3D array:

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

The third dimension is an array of three pixel colors (blue, green, and red). The second dimension is an array of columns, and the first dimension is an array of rows. If the image is black-and-white, then it should be represented as a 2D array. Let's create a black-and-white image of a letter 'x' composed of just 9 black-and-white pixels for simplicity:

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

The second dimension is an array of columns of pixels, and the first dimension is an array of rows of columns of pixels. Black-color pixels have values of 0, and white-color pixels have values of 255. Now, you should understand how image maps (bitmap or colormap) are made. However, these approaches are just a few possible ways of how real images are represented on the computer. In addition to RGB, there are **RGBA color model - Wikipedia (https://en.wikipedia.org/wiki/RGBA_color_model)** and **CMYK color model - Wikipedia. (https://en.wikipedia.org/wiki/CMYK_color_model)**

Binary Encoding. In this assignment, you will use an RGB color model without an alpha channel (not RGBA). All numbers in the RGB model should be integers that are displayed in their decimal notation on the screen when you print the array. However, all numbers are stored in their binary form in the computer memory. So, we can use this property of binary representation of integers and modify numbers in such a way that their least significant binary digit becomes either 0 or 1. In doing so, some numbers become one digit bigger or smaller or not modified depending on the encoded message that has bit values 0 and 1.

For example, we want to encode a word 'hello' in an image. First, we need to calculate how many bytes in the word. Since we are going to use a general ASCII encoding **ASCII - Wikipedia (https://en.wikipedia.org/wiki/ASCII)**, where each symbol (character) has a numeric code in the range from 0 to 255, we need 5 bytes for 5 characters in 'hello'. Each symbol requires a byte of memory because a byte has 8 bits and 2^8 = 256 that is sufficient to encode all numbers from 0 to 255.

Second, we need to have an image size that is sufficient for encoding 5 bytes or 5 x 8 bits = 40 bits or 40 pixels if the image is not colored and ceil(40//3) = 14 pixels if the image is colored (because each pixel has three numbers). Then, we need to convert each pixel number from its decimal form to its binary form. It can be done by using the method format: for example, format(num, "08b") converts num into its binary form (as a string, not bytes). If num = 9, then its binary form returned by the format is '00001001'. If a letter is 'h', then its ASCII code can be obtained using the function ord(), and its binary code returned by the format (ord ('h'), "08b") is '01101000'.

Next, we need to override the last bit (least significant bit) in each number of an RGB pixel according to the message. If the message has 0 then the overwritten bit should be 0 too. If the message has 1 then the overwritten bit should be 1. For example, let's assume that there is a 2x3 RGB image that has 6 pixels and can be used to encode 6x3 = 18 bits or 2 bytes (16 bits). The original image has the following array (it is a red rectangle):

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

The message 'he' (the first two letters of 'hello') in binary is '01101000 01100101' (I put a space between letters, so that you can identify them; however the space should be absent from the binary form!) The modified image should have the following array:

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

Look carefully: each number of the first 16 numbers has the same last bit as the corresponding bit in the message. The first bit is 0 in the message, so the last bit is 0 in the first number 0. The second bit is 1 in the message, so the last bit is 1 in the second number 1 (notice that the original number is changed from 0 to 1, it is increased by 1). The third bit is 1, so the last bit is 1 in the third number (255 is odd, that is why it is not changed!). The similar procedure is applied to all other numbers with exception of the two last numbers because we only need 16 numbers, not 18 numbers to encode 16 bits.

Finally, we just need to save the image in a binary image file. It can be done using the openCV library that is designed to manipulate images. Images can be read and written into various formats such as png, jpeg, tiff, gif, etc. Since we use a steganography least-significant bit technique, we need to save our image in a **png format** because this format preserves the numbers (it has lossless compression, not lossy compression as jpeg or other file formats). The steganoimage will appear the same as the original image because human perception of colors is not very precise. In this example, the image is so small that you may not be able to see it well. In your program, you will work with larger images and see by yourself that modified steganoimages look the same to you.

Binary Decoding. It's not fun just to encode messages. You should be able to decode messages too. Decoding a steganoimage is the reverse process of encoding. You will use an openCV method to open a steganoimage file (in a png format). Then, you will convert each number (an integer) in the image array into its binary form (as a string) and extract the least significant bit. Then, divide the extracted bits into bytes and convert each byte into an integer and then to a string using the functions int and chr, for example chr (int (byte, 2)), where byte is the binary representation of an integer (or the ASCII code of a character).

Codecs. When you done with implementation of your steganography program, you can start to improve it by using two additional techniques: Caesar Cypher and Huffman Codes. A Caesar cypher is a simple cryptographic technique used to encode text, whereas Huffman Codes is a simple compression algorithm used to compress text messages. In doing so, you will create three different codecs (**co**der - **dec**oder) for text messages. The first codec encodes and decodes a message into and from its ASCII binary form that is already described above (Binary Encoding and Decoding). The second codec encodes and decodes a message into and from its ASCII binary form by using a Caesar cypher (described below). And the third codec encodes and decodes a message into and from its ASCII binary form by using a compression algorithm Huffman Codes (also described below).

Caesar Cypher. A Caesar cypher is based on shifting a code for each symbol by some number. For example, let's choose a shift that equals to 3 and a letter from the English alphabet that is A, then the encoded symbol is a letter that follows A and is located at the position that is shifted from A by 3, so it is D. D is the fourth letter in the alphabet, whereas A is the first letter, and the distance between them is 3. In this encoding we use the right shift, so A becomes D, B becomes E, C becomes F, and so on. It is also easy to decode the cypher because we can use the left shift to reverse the encoding, so F becomes C, E becomes B, and D becomes A. The only problem is how to encode the last symbols because they are not followed by any other symbols. This problem can be easily solved if we assume that the symbols are arranged in a circle, so the last symbol is followed by the first symbol. You will use the ASCII symbols (0-255). So, if the ASCII code is 255 then it should follow by the ASCII code 0 (notice that some ASCII codes correspond to not printable characters, you will not be able to see them). To achieve this functionality, you can use the modulo operation:

code = ord(symbol) % 255, where symbol is some ASCII character and code is its corresponding ASCII code.

Your Caesar cypher should have a user-defined shift, so, the user can choose what shift to use, for example, 3, 5, or -4.

Huffman Codes. Huffman Codes are used for text data compression. They are binary encodings of text and will be used in your program as the third codec method in your cryptography program. The Huffman Codes algorithm is based on assigning a numeric binary code to a text character depending on the character frequency. Frequent characters have binary codes with small prefixes, rare characters – with long prefixes. So, overall we need to use less bits to encode the text.

The algorithm has two parts:

1. Creating a Huffman tree
2. Traversing the tree to find codes

Creating a Huffman tree is done in a greedy approach. First, each character is represented as a node of the graph; next, two nodes corresponding to the least frequent characters are selected and combined into a tree. This new tree has a root node that has frequency equal to the sum of two frequencies of its children. Then, the process is repeated, the next two nodes are combined into a tree till no nodes are left. In doing so, the generated tree is the correct tree where least frequent nodes are at the bottom and more frequent nodes are at the top.

While decoding, **you assume that the Huffman tree is already built** and decode text by traversing it. For example, if the encoded message is 01101, your algorithm should take the first symbol 0 and traverse the tree to the left node of the root (0 means go left, 1 means go right). Then, it should check if it is a leaf node, if yes then it should record its value (a letter) and start the process of traversing from the root of the Huffman tree. Otherwise, it should proceed further to the next tree node. Since the next symbol is 1, it should go to the right node of the current root, and perform the same operations as before.

Delimiter. When you work with a stream of data like extraction of least significant bits from bytes in a file, you need to know when to begin and to stop the process. By default, you can assume that you can start encoding and decoding from the first byte, but you cannot assume when to end the encoding and decoding by default. You need to use a delimiter for this purpose. For example, you can use a hash symbol '#' as a delimiter that stops the encoding and decoding processes.

Programming Approaches

In this assignment you are free to create your original code. However, all implementations should comply with the following abstract data type interfaces and application programming interface (API) for codecs:

Steganography ADT:

1. Steganography() - creates an object of class Steganography
2. print() - prints encoded and decoded messages in text and binary forms
3. show(filein) - displays an image file in a window using a Matplotlib (or openCV) method
4. encode(filein, fileout, message, **codec**) - save a message in an image from a given file into another file using a specified codec
5. decode(filein, **codec**) - returns a decoded text message hidden in a given file using a specified codec
6. **codec** is a name (string) used for identification of a codec class, it should have values 'binary' for the Codec class, 'caesar' for the CaesarCypher class, or 'huffman' for the HuffmanCodes.

Your steganography class should be saved in a module (file) called **steganography.py**. You are encouraged to use a template file steganography.py that contains some additional information. It is available in Files/Programming Assignments/PA2.

Codec API:

1. Codec() - creates an object of class Codec
2. encode(text) - converts text into the binary format and returns the binary data
3. decode(data) - converts binary data into the text format and returns the text

Caesar Cypher ADT (Subclass of Codec):

1. CaesarCypher(shift) - creates an object of class CaesarCypher with a specified shift, the default shift should be set to 3
2. encode(text) - returns the encoded binary message
3. decode(data) - returns the decoded text message

Huffman Codes ADT (Subclass of Codec):

1. HuffmanCodes(Codec) - creates an object of class HuffmanCodes
2. encode(text) - returns the encoded binary message
3. decode(data) - returns the decoded text message

All Codec classes (Codec, CaesarCypher, HuffmanCodes) should be saved in one module (file) called **codec.py**. You are encouraged to use a template file codec.py that contains some additional information. It is available in Files/Programming Assignments/PA2.

Testing

To test your codec implementations, use the following driver program that you can add to your codec.py:

```
if __name__ == '__main__':
    text = 'hello' # or 'Casino Royale 10:30 Order martini'
    print('Original:', text)
    c = Codec()
    binary = c.encode(text)
    print('Binary:', binary)
    data = c.decode(binary)
    print('Text:', data)

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

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

You can check your steganography program using the cryptography.py file as a main program. It is available in Files/Programming Assignments/PA2. You can use either fractal.jpg or redbox.jpg image file as input files. Do not forget to save steganoimages in png files!!!

```
# 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('\n').upper()
        if op == 'Q':
            break
        if op != 'P':
            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)

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

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()
```

Grading Rubric

All supplementary files are located in the **Files**/ Programming Assignments/ PA2 folder on Canvas. Your program will be manually graded according to the following rubric:

Criteria	Points	Excellent	Good	Satisfactory	Unsatisfactory
Submitted on time	75 possible points	0% deduction submitted on time	5-10% deduction 1 - 3 days late	20% deduction 7 days late	40% deduction more than 7 days late
Program Performance (program execution description)	25	runs without errors for all four classes 25 points	runs without errors for three classes 20 points	runs without errors for two classes 15 points	runs without errors for one class 10 points
Program Functionality (program produces the desired results)	30	correct output for all four classes 30 points	correct output for three classes 25 points	correct output for two classes 15 points	correct output for one class 10 points
Program Style (neatness, readability, conciseness, comments, inventiveness)	10	all characteristics are present 10 points	one of the characteristics is absent 8 points	two-three characteristics are absent 4-6 points	more than two characteristics are absent 0-2 points
Program Specifications	10	all required methods are implemented	one-two of the required methods	three-four of the required methods	more than four of the required

(how program is implemented)		10 points	are not implemented 8-9 points	are not implemented 6-7 points	methods are not implemented 0-5 points

What to turn in

Submit your program files steganography.py and codec.py here on Canvas before the due date. As always start early and ask questions in lab sessions, office hours, and on Canvas.